NoC-based hardware software co-design framework for dataflow thread management

Somnath Mazumdar, Alberto Scionti, Stéphane Zuckerman, Antoni Portero

The Journal of Supercomputing (2023) 79:17983–18020
https://doi.org/10.1007/s11227-023-05335-8

NoC-based hardware software co-design framework for dataflow thread management

Somnath Mazumdar[1] · Alberto Scionti[2] · Stéphane Zuckerman[3] · Antoni Portero[4]

Accepted: 22 April 2023 / Published online: 11 May 2023

Abstract
Applications running in a large and complex manycore system can significantly benefit from adopting the dataflow model of computation. In a dataflow execution environment, a thread can run only if all its required inputs are available. While the potential benefits are large, it is not trivial to improve resource utilization and energy efficiency by focusing on dataflow thread execution models (i.e., the ways specifying how the threads adhering to a dataflow model of computation execute on a given compute/communication architecture). This paper proposes and implements a hardware-software co-design-based dataflow threads management framework. It works at the Network-on-Chip (NoC) level and consists of three stages. The first stage focuses on a fast and effective thread distribution policy. The next stage proposes an approach that adds reconfigurability to a 2D mesh NoC via customized instructions to manage the dataflow thread distribution. Finally, a 2D mesh and ring-based hybrid NoC is proposed for better scalability and higher performance. This work can be considered a primary reference framework from which extensions can be carried out.

Keywords Co-Design · Dataflow · Framework · Hardware · Hashing · NoC · Software · Thread

1 Introduction

Emerging extreme-scale machines[1], and their successors are expected to manage many threads that are orders of magnitude greater than in current petascale machines, with more stringent power and resiliency constraints [1, 2]. In recent

[1] ORNL's Frontier supercomputer officially executed more than an ExaFLOP/s. See https://top500.org/news/ornls-frontier-first-to-break-the-exaflop-ceiling/.

✉ Somnath Mazumdar
sma.digi@cbs.dk

Extended author information available on the last page of the article

years, communication-centric applications have evolved towards using massively dynamic and distributed programming models. Such transformations have forced computer architects to change the hardware accordingly to the requirements imposed by the applications running at such large scales [3]. Recent CPU and application-specific accelerator designs favour the integration of a vast number of simple cores (e.g., single-issue, in-order cores) [4–6] to easily reach the trade-off between performance and energy consumption, being also able to increase the number of threads that can be executed in parallel.

A large part of a processor's power budget is used to transport data packets from and to cores and memory modules. Manycore systems aim to improve energy efficiency by increasing parallelism and resource utilization. However, improving resource utilization without creating contention and workload imbalance is not trivial. Such a solution should be holistic and must follow a hardware-software co-design approach. Here, the software should provide flexibility while exploring specific hardware enhancements that allow for higher performance. Specifically, hardware-based support for explicit multithreading can facilitate the smooth execution of applications spawning a massive number of threads and offer advantages over managing them at the software level.

An application consists of multiple sub-tasks which generate multiple threads. Threads can communicate with each other and may be dependent on input data. Such scenarios may cause I/O or memory stalls, significantly reducing overall performance. For the smooth and efficient execution of applications, the underlying system architecture needs to dynamically adapt to the 'ever-changing' situations with minimal commotion to their functionality. In general, the program execution flow is governed by control or data dependency. In control dependency-based execution, threads are instantiated when some conditions are met (i.e., generally expressed with conventional, high-level programming languages). In the data dependency model, threads start to run when the required input data is available (i.e., this model is mainly data-driven). Current popular program execution models (PXMs) rely on von Neumann architectures and exhibit large thread synchronisation overheads. Their inherent sequential nature makes it tough to guarantee the correctness and race condition freedom when one considers the execution of multithreaded programs [7].

Since the beginning of the 2010s, and with interest in big data and machine learning-based applications, dataflow and event-driven models have regained popularity in high-performance computing (HPC) communities. In particular, as the on-chip core count has been steadily growing, expected issues come with memory latency, synchronisation, and the natural expression of parallelism both at the application and hardware levels. Dataflow models tend to propose simple and efficient mechanisms to tackle these issues [8]. Dataflow PXMs can achieve good performance by reducing energy consumption by up to 40% [9]. In a dataflow PXM, if input data is available, the independent portion of the application can be executed in parallel. Such an approach helps to achieve better spatial parallelisation because the dynamic form of dataflow can support a higher level of parallelism by supporting data repetitions [10]. Overall, managing threads at low-level have several advantages, such as fewer hotspots, better performance, and efficient power utilisation. Whilst both HPC and high-throughput computing applications can significantly benefit from the

adoption of dataflow-oriented PXMs, most of the current implementations are either generic but software-based or rely on hardware acceleration, but with a specific type of application in mind (e.g., graphs, neural networks).

Application mapping on manycores is challenging when multiple constraints, such as limited energy (power) consumption, latency, and throughput, must be satisfied. The increased core count offers a higher degree of parallelism. As the chip's core count rises, each core can generate enormous (data) traffic inside the chip. Then, stable performance becomes dependent on the contention of other crucial functional components, such as interconnections. In this context, the interconnection subsystem becomes vital for achieving better performance. There is no easy way to categorise the traffic generated by the applications in the interconnect subsystem of manycore systems (i.e., also including smaller chip-multiprocessors—CMPs) [11]. However, an inefficient interconnection subsystem can reduce overall system performance and consume a significant portion of the area and power budget of the chip [12]. Today's popular interconnection subsystems for manycore systems are known as Networks-on-Chip (NoCs).

An NoC is an embedded packet-switching network supporting the (data packet) communication between the processing elements (such as general-purpose cores and application-specific accelerators) and the primary memory. NoCs have the potential of offering better scalability and more deterministic performance [13]. NoCs have a simple organization and mechanism for controlling the data traffic but can consume a significant fraction of energy while connecting many cores. For instance, Vangal et al. showed that on the Intel TeraFLOPS chip, the NoC uses up to 28% of the chip's power [14]. Its routing mechanism and topology influence the performance and power consumption of an NoC. Topology impacts the overall network latency and power consumption. To reduce such issues, for connecting very high core counts, hierarchical topologies (e.g., bus and 2D mesh based [15], mesh and ring based [16]) have also been proposed.

Paper contribution In this article, we propose a dataflow-based thread management framework (see Sect. 4) which relies on specific features embedded in the NoC to facilitate the management of a massive number of threads within a manycore architecture by easing the control of data packets across the interconnection. As such, the proposed framework aims to improve the energy efficiency of a manycore system executing applications spawning a very high number of threads. A dataflow-based PXM is used to efficiently distribute dataflow threads across the available processing cores by limiting the rise of single hot spots in the communication patterns. It is worth noting that this framework can also be customised for conventional control-flow programming models. The framework consists of three stages from top to bottom (refer to Fig. 1). The top stage focuses on fast dataflow thread distribution on the available cores. In the middle stage, a software-controlled reconfiguration mechanism is added on top of a custom 2D mesh router (actually based on a custom configuration of interconnection rings). Finally, a hybrid NoC topology (bottom stage) is explored to gain performance when the number of cores to connect grows. Indeed, the combination of rings and a 2D mesh using routers equipped with conventional crossbar-based switches provide better performance than standard 2D mesh architectures. It is worth mentioning that the framework is flexible so that the

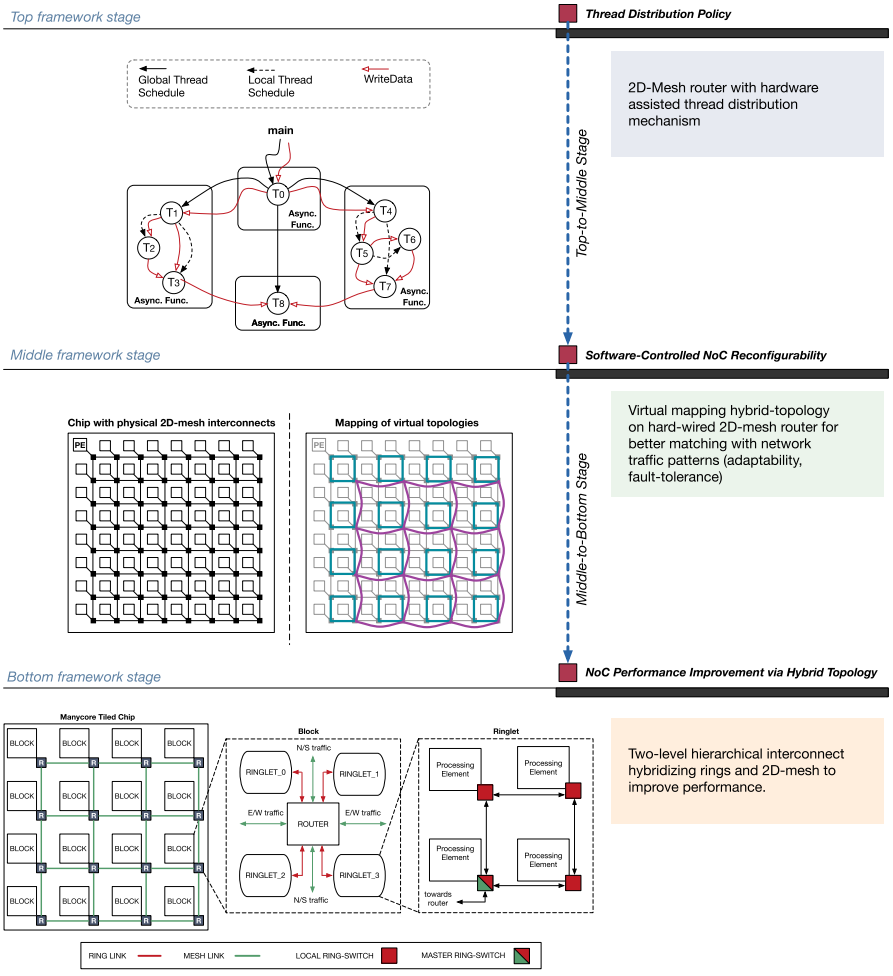


Fig. 1 The proposed framework with three sequential stages

user does not need to use all three stages: any stage can be skipped based on requirements. Overall, the contributions of this paper are as follows:

- *Top stage (thread distribution policy)* We proposed a hash-based thread distribution scheme that can be very efficient with simple hardware modification to support the (energy-efficient) distribution of a massive number of concurrent threads over the available cores (refer to Sect. 4.1) [17].
- *Middle stage (software-controlled noc reconfigurability)* To let the software layer control the interconnection topology, we proposed a lightweight mechanism to extend the router microarchitecture and make the actual NoC topology (i.e., called *virtual topology*) directly controllable through dedicated core instructions (see Sect. 4.2) [18]. Whilst demonstrated on top of an NoC architecture based on

multiple rings, the proposed mechanism remains general enough to be adapted to other interconnections, e.g., routers with traditional crossbar switching elements.
- *Bottom stage NoC performance improvement via hybrid topology)* To improve the scalability and performance of the NoC infrastructure, a hybrid architecture combining a 2D mesh and rings is proposed. The results show that it is more efficient compared to the traditional pure 2D mesh topology (see Sect. 4.3) [19]. The result shows it performs well by efficiently processing local (rings) and global (2D mesh) traffic. Such configuration allows us to exploit network traffic localisation better, thus facilitating traffic management at a low energy cost.

2 Background and related work

This section presents the two foundations of this work: dataflow-inspired program execution models (PXMs), and Network-on-Chip (NoC) architectures. Whilst the large reviewed literature demonstrates some foundational idea captured by our proposed hardware-software co-design framework, no one of the reported works covers all the necessary aspects, i.e., providing an effective mechanism for managing the distribution of a large number of threads, while improving performance, adaptability, energy efficiency and scaling up capability. Section 3 and Sect. 4 aim to fill this gap.

2.1 Dataflow models

We will first introduce the concept of dataflow models, and then discuss the use of a modern dataflow-inspired PXM for our contribution.

2.1.1 Dataflow models: principles

Dataflow models have a long history, from both hardware [20, 21], data structures, system software, and software perspectives [22–24]. It could be argued they hide behind superscalar processors' out-of-order engines; are still actively used for digital signal processing circuits in embedded systems thanks to variations around dataflow, such as Synchronous Data Flow [25]; and more recently, explicitly dataflow-inspired constructs have been added to programming languages, e.g., OpenMP [26].

The original dataflow PXM specifies a *firing rule*: a dataflow *actor* is *ready* to fire when all of its input data tokens are present, and there are no remaining output data tokens to be consumed. This rule is relaxed in further iterations of dataflow PXMs, e.g., to allow for general recursion. A dataflow program can be represented as a directed graph, called a dataflow graph (DFG). Its vertices are dataflow actors, and its arcs indicate in which direction data tokens *flow* from one actor to the other. DFGs naturally expose parallelism, which both hardware and software systems can then exploit.

Since the 1990s, several hybrid von Neumann-dataflow PXMs have been proposed to take advantage of more traditional off-the-shelf high-performance

microprocessors, such as e.g., EARTH-MANNA [27, 28], which combines a compiler to write dataflow-inspired programs into C combined with a low-level run-time system. These hybrid systems provide a 'macro-dataflow' programming environment to express computations, and then map the resulting generated code to the underlying hardware thanks to a combination of compiler-generated code and run-time system calls.

The last dataflow-inspired architectures include explicit multi-threading (XMT) architecture, which introduces an abstract execution model, where switching from serial to parallel execution is made through explicit spawn/join instructions [29]. Another dataflow-inspired high-performance hardware system is the Maxeler processing platform [30]. Readers can refer to the survey for more [31].

2.1.2 Codelet model

With the rise of massively parallel chip multiprocessing systems, i.e., manycore systems, the early 2010s saw a renewed interest in examining dataflow-based PXMs to exploit the increasing amount of parallelism contained in a single chip, but also between CMPs. One of the resulting PXMs is the Codelet Model [32, 33]. Much like EARTH [28], it is a hybrid von Neumann-dataflow model designed for massively parallel multicore systems. However, it is *event-driven*: on top of the data tokens of dataflow, hardware and software resources may also be expressed as dependencies to help the system decide how to map the computation on the underlying hardware. Hence, a codelet may only *fire* if all of its data *and* resource dependencies are met.

While they are the *quantum of execution* of the PXM, codelets cannot be called directly by the programmer. Instead, they are stored in a *Threaded Procedure*. Threaded Procedures are essentially asynchronous functions, and act as a container for a small *codelet graph*, and a *data frame* to hold inputs, outputs, and intermediate values shared by the threaded procedure's codelets. A threaded procedure descriptor can be sent to any cluster of cores, but once it has been assigned to a given cluster to be executed, its codelets can only run on the cores of the selected clusters. This helps maintain data locality, e.g., if the cluster features some cache or scratchpad memory available to all its cores. Codelets have a read-write access to its threaded procedure's frame, and rely on the programmer and compiler to have produced a *well-behaved* codelet graph.

On top of defining a way to represent a unit of computation, the Codelet model also defines an abstract machine model. In the model, the target system is comprised of hundreds of cores, grouped into clusters. Each cluster comprises several compute units, and a single synchronisation unit. Each compute unit is equipped with a codelet pool, which can hold one or more codelets ready to run, whereas synchronisation units are tasked with distributing such codelets to the various compute units of their cluster. At the same time, synchronisation units are also in charge of tracking local thread procedure invocations, and must then manage the memory pools tied to them. They must be software-managed to deal with dynamic memory allocation issues. An unknown number of thread procedures (in particular, the data frame sizes vary with inputs, outputs, and intermediate values) will be stored in the pools and can be 'fed' with additional thread procedure descriptors from other synchronisation units.

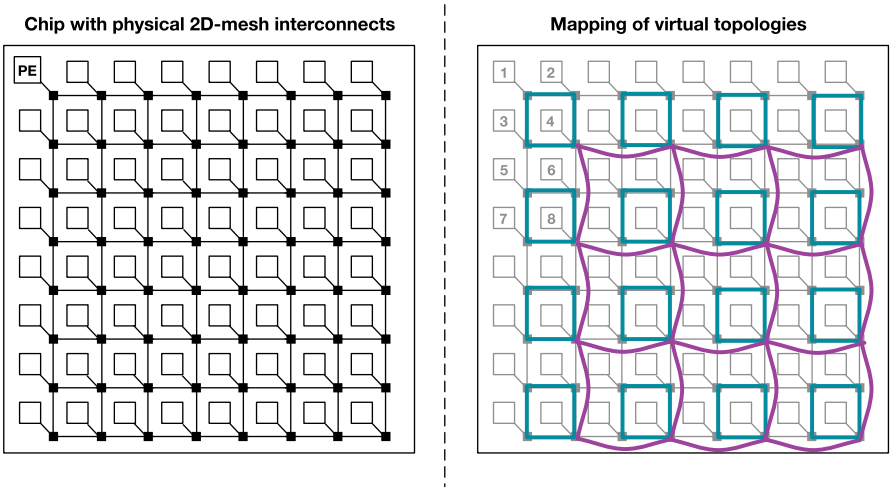


Fig. 2 Mapping between the physical network with a 2D mesh topology and a multi-level virtual topology. Links to the physical network are organised into local rings (blue lines) and a global 2D mesh among rings (purple lines)

2.2 Network-on-Chip architectures

In this sub-section, we will first introduce the concept of 'virtual' topology mapping used in the context of customizing the system interconnection, and later discuss the related literature.

2.2.1 Primer to virtual mapping

The 2D mesh topology has become popular due to its advantages compared to other alternatives regarding wiring area, power cost, and fault tolerance [34]. The left side of Fig. 2 depicts the typical organisation of 2D mesh-based CMP: processing elements (PEs—white squares) communicate with routers (black squares) which are connected through a 2D mesh (black lines). In such a topology, the active communication elements (i.e., the routers) have five input/output channels each, which map to the four communication directions of the mesh (i.e., north, south, east and west links) and the local communication link with the PE. The right side of Fig. 2 shows an example of a *virtual topology* mapping, with groups of adjacent PEs communicating through virtual rings (blue lines), which are further connected through a virtual mesh (purple lines). Mapping the physical topology with the desired virtual one requires the underlying routers' architecture to expose additional functionalities. In particular, links should be enabled/disabled or bypassed, i.e., the traffic flowing in it is directly injected in another link without passing the switching/routing process. In other words, data packets are directly forwarded to the following router. In the right side of Fig. 2, for instance, routers of PE 3 and PE 5 (PEs are numbered in the top-left corner) can disable respectively south and north links, while the router

of PE 6 can configure the north link in bypass mode so that its traffic is directly injected in the south link of PE 8. An example of an NoC architecture supporting this kind of dynamic mapping is Panther [35]. Although it allows significant power savings in principle, the entire NoC architecture uses routers based on a conventional microarchitecture, which becomes power hungry (i.e., the majority of the dissipated power is due to the crossbar switch and link drivers) when the number of PEs to connect grows as in manycore. [36] proposed a method to perform high-level mapping of cores onto various NoC topologies based on multiple objective functions. Readers can refer to the survey paper [37], which reviews existing routing and selection techniques for 2D mesh topology considering various NoC-related performance parameters.

2.2.2 Hybridizing NoCs

For a small number of cores, ring topology is demonstrated to be very effective, requiring a low-radix router (i.e., a packet-switching element with a low number of input/output ports connecting to the other elements of the network). Interestingly, a ring topology can outperform a mesh topology for moderate to high memory access locality-based workloads [38]. Various efforts have been made to connect local and global rings via special routers called bridges to improve scalability together with performance and better energy consumption [16, 39]. Due to scalability and a high level of fault tolerance, 2D mesh topology has become very popular to efficiently connect a moderate number of cores (e.g., Polaris chip [12]). However, 2D mesh topology suffers from space and power trade-offs for a vast number of connected cores [12].

Hybridization has been explored as an effective way to exploit the benefits of each topology at different scales (i.e., when a different number of cores to connect varies). Kim et al. propose using ring-based routers to implement a scalable 2D mesh topology [40]. Despite the relatively lower cost of the proposed packet-switching elements, the network's physical topology remains fixed, with links kept active in all of these elements, thus leading to low resource usage for lightly used connections and larger overall power consumption.

In [41], the authors proposed a hybrid NoC architecture with high-speed transmission lines. It uses an adaptive routing technique to send long-distance packets. Manzoor et al. proposed a deadlock-free congestion-aware routing algorithm for a 2D mesh network while not using any virtual channels [42]. The proposed algorithm fuses deterministic and partially adaptive algorithms. [43] presents a routing algorithm based on a four-step method for a 2D mesh network. The proposed method collects the congestion information that enhances the regional traffic information. [44] proposed an application-to-cores mapping along with a modified 2D mesh NoC. Next, the proposed approach implements techniques to determine the congestion level of the NoC links dynamically and thus adjust the application-to-cores mapping accordingly. In another work, a Chisel NoC generator is proposed for enabling design exploration and evaluation of heterogeneous NoCs [45].

Hierarchical rings with deflection is a hierarchical ring-based NoC design for improved energy efficiency, where buffers within individual rings are avoided while

needing up to four levels of hierarchy to connect a modest number of cores [16]. CSquare proposes a way of clustering routers so that clusters adopt an internal tree-like organisation [46]. The authors showed that this topological design improves throughput while lowering the average latency over mesh-like topologies under the uniform traffic pattern. Transportation network-inspired NoC is another hierarchical ring topology that employs a hybrid packet-flit, credit-based flow control mechanism for better scalability and priority-based arbitration for better performance [47]. It allocates channels (i.e., links) with flit granularity, while buffers are allocated with packet granularity to reduce buffer counts. 2D-HERT is a hierarchical expansion of a ring topology, focusing on optical NoCs [48]. Kilo-NoC is a topology-aware quality-of-service (QoS)-oriented architecture, adopting a low-diameter (i.e., the largest, minimal hop count over all pairs of cores) topology [49]. It provides a service guarantee for applications with reduced power and area costs. It reduces the extent of hardware support to portions of the die, reducing router complexity to support large core counts. In [50], authors present a hybrid architecture where a 2D mesh network is partitioned into several smaller sub-meshes. Next, the sub-meshes are connected using a hierarchical ring interconnect for delivering global traffic. This work uses a bridge module to drive traffic to the different levels of the hierarchy. The addressing and routing scheme has also been modified to support the proposed topology.

3 Overview of the proposed framework

In recent times energy-awareness has become paramount to keep the pace of gaining performance from one generation of HPC systems to another. Consequently, the design of microarchitectures considers energy efficiency as one of the main requirements. Energy efficiency can be achieved: by increasing the parallelism via increasing the core count or improving the computing capabilities via integrating application-oriented features. In manycore platforms, the interconnection and memory modules are the critical subsystems as they are shared by all cores for data exchange. With applications exploiting such parallelism, the capability of adequately distributing threads (which may refer to a hierarchy where OS-level threads map onto low-level hardware-assisted threads) on the system resources becomes of primary importance. In particular, the way threads are dynamically spawned and associated with cores, and how the communication (data exchanged) happens among running threads are critical aspects to be addressed in the (co-)design of new high-performance energy-efficient processors. That said, here the primary research question is: *how can a hybrid NoC-based dataflow-oriented thread distribution mechanism improve the run-time performance?*

Dataflow is mainly intended to explore ‘spatial parallelism’ offered by many-core CMPs, meaning that threads can execute in parallel if enough resource exists and all their inputs are available. The proposed framework incorporates a flexible dataflow thread management layer (i.e., resembling the Codelet model that efficiently distributes dataflow threads across the available processing cores. We consider the dataflow model as it has fewer management issues, and the available general-purpose hardware can be used to exploit the inherent parallelism of the

dataflow applications. Worth mentioning here is the fact that the threads targeted by this approach are considered the ones lying at the lowest level of the software stack. In other words, these threads are directly managed with the assistance of the underlying hardware; as such, higher-level threads [(e.g., OS threads (similar to [36])] may be mapped on top of them without impacting the operations performed by the proposed approach or on the performance. On the other hand, the proposed hybrid NoC can support control-driven and data-driven execution models with better scalability, throughput and improved latency, lower power consumption and area cost.

To answer the primary research question, this paper focuses on a hardware-software co-design framework (refer to Fig. 1), which aims at providing mechanisms to (i) seamlessly distributing (low-level) threads on the available cores, (ii) allow the interconnection to adapt to map a specific virtual topology to better serves the application's data traffic, and (iii) improve the overall performance by exploiting the hybrid topology of the underlying interconnection. The proposed framework leverages the implementation of a full-fledged dataflow PXM. Three main goals are tailored to provide software support for exposing a data-driven thread spawning mechanism connected to a hardware-assisted system that automatically distributes threads on the available cores while avoiding creating communication hot spots. Also, to better comply with the application communication patterns, interconnection topology can be adapted and controlled via a simple software interface. Such an approach advocates for better energy efficiency and scalability. More specifically, the proposed framework consists of three stages to answer three sub-research questions.

1. *Thread distribution policy How to efficiently distribute the massive number of concurrent (low-level) threads among available cores in such a way being energy efficient?* To answer this research question, we propose a hardware-assisted hash-based thread distribution scheme that can be efficiently implemented and integrated into a conventional 2D mesh router microarchitecture. Specifically, the proposed scheme targets a dataflow execution model and offers abstraction and flexibility.
2. *Software-controlled NoC reconfigurability How to improve the adaptability of the underlying NoC interconnection to the application communication patterns?* To counter this issue, we propose a scalable software-defined NoC (SDNoC) architecture, which allows mapping different (virtual) topologies upon a fixed physical 2D mesh NoC. In the proposed approach, the software layer can directly control the network topology to accommodate various application requirements and communication patterns.
3. *NoC performance improvement via hybrid topology How to improve network performance by exploiting traffic localisation?* To address this question, a hybrid, scalable and efficient (physical) NoC topology is designed, which fuses rings and 2D mesh topology to provide higher performance whilst efficiently processing local (rings) and global (mesh) traffic. The results show that it is indeed efficient compared to the more traditional full flattened 2D mesh topology, which is diffused in modern HPC processors. Also, it keeps the complexity of the overall solution lower than the widely used 2D mesh topology.

4 Hardware-software co-design framework

As already mentioned in Sect. 3, the proposed hardware-software co-design-based NoC framework consists of three stages, each of which is presented below.

4.1 Thread distribution policy

We use a PXM directly derived from the Codelet model, where assisting hardware embedded in the packet-switching element (i.e., the NoC router) provides performance improvements. The applications are divided into a set of fine-grain threads, each totalling no more than a few tens or hundreds of instructions. Threads represent the quantum of execution and can exchange data with each other (if needed).

4.1.1 Added software support for mapping threads

Employed PXM allows the construction of the DFG at compile time, explicitly showing data dependencies among threads. The thread context includes the (data) frame and a unique thread identifier. Each thread holds a local storage space (*frame*) used to receive input data from producer threads and to write intermediate results. It also contains a *scheduling slot* (SS) counting the number of inputs still required for the execution. To preserve locality and allow for better latency hiding, threads are grouped into *asynchronous functions* (AFs). As part of the hashing mechanism, the framework dynamically groups cores to form virtual nodes (VNs), forcing threads within an AF to be executed on the same VN. To expose these characteristics at the programming level, the proposed architecture extends the core instruction set architecture (ISA) with a reduced set of dedicated instructions (possibly wrapped by high-level programming language functions, e.g., C/C++) as follows:

- `CreateThread(*code,SS,frame)` Creates a new thread context, i.e., the SS, the type of the thread, a unique identifier, and the required space to hold the thread's data frame and allows to schedule the execution of threads belonging to the same thread pool (TP) within the same VN.
- `CreateAF(*code,SS,frame)` As the above instruction, but it allows a new asynchronous function (i.e., a new thread spawned outside the VN).
- `ReadData(offset)` Reads data from a thread's frame at a specific offset within the frame.
- `WriteData(TID,frame,offset,data)` writes data to a thread's frame at a specific offset within the frame (both within and outside the current VN).
- `DecreaseSS(TID,dep_cnt)`: Allows decreasing the SS of a thread by the number of resolved dependencies.
- `DeleteThread()` Removes the context of a thread that has completed the execution.

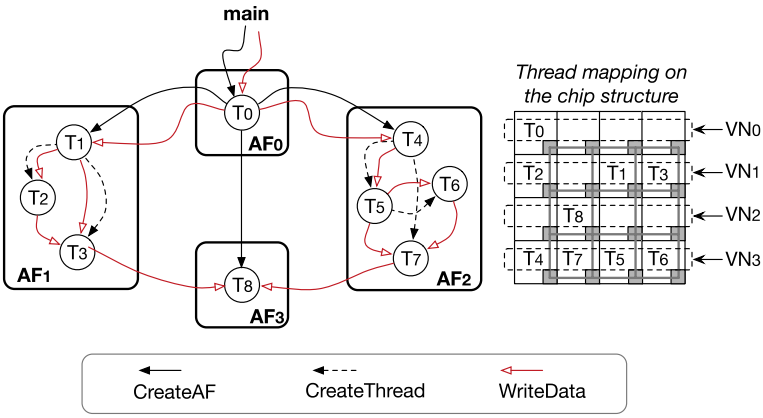


Fig. 3 Application adhering with the proposed PXM and a possible mapping of threads on the PEs

- `SetVN(N_pe)` Sets the number of cores being part of each VN and sends a broadcast message to all indicating the number of cores (PEs) composing each VN.
- `ConfigRouter(*config,Rd,B)` Allows configuring routers by specifying the memory address where the configuration is stored. The destination router identifier is contained in the `Rd` variable, while flag `B` indicates if the configuration is broadcast to all routers.

The compiler can aggregate multiple writes (e.g., dealing with large loops) and use a single `DecreaseSS` signalling operation to update the corresponding SS field to optimise the execution.

A PE within the current VN is automatically selected and signalled whenever a new thread is spawned. Similarly, whenever a thread creates a new AF, the destination PE is selected within the whole chip. Hence, the creation of a new AF is led back to the scheduling of the root thread of the DFG contained in the AF. Figure 3 shows an example of a simple kernel application consisting of four AFs, each with its DFG. Both AFs and threads are directly managed by the compiler, which is responsible for mapping high-level programming constructs (e.g., `#pragma omp for` when using OpenMP) with the correct sequence of `CreateAF` and `CreateThread` instructions. Figure 3 also shows that AF scheduling requests and writing operations remain well confined to the local VN. By monitoring scheduling slots, the hardware unit automatically fires threads that become runnable without explicit instruction. On the contrary, the execution completion is signalled by the `DeleteThread` instruction that allows freeing resources held by the thread.

4.1.2 Hardware support for thread distribution

The target CMP-based platform comprises many cores tied to lightweight routers supporting a 2D mesh topology. A core-router complex is referred to as a tile. Figure 4 shows how a 2D mesh NoC connects a large group of tiles covering the entire

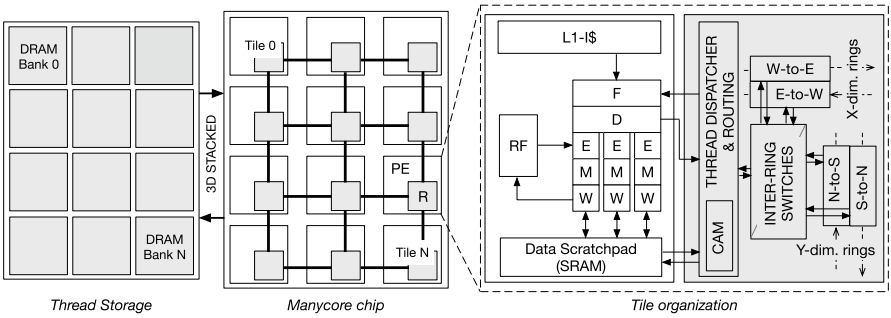


Fig. 4 Chip organization: tiles contain a PE (white box) and router (grey box). The scratchpad substitutes the traditional L1-data cache

chip area. Routers are augmented with our (fine-grain) dataflow thread distribution hardware support: a local unit called *Thread Dispatcher* (TD) manages the threads during their lifetime. In particular, it allocates internal space for storing thread contexts every time new threads are created, removes previously allocated resources whenever threads are complete, and reads from (respectively writes to) associated frames. Since the software controls the behaviour of threads through the extended instruction set, the TD directly interacts with the decoding stage of the PE. We assume that only one thread at a time can be executed in each PE.

We used hashing for thread distribution in a resource-constrained environment. The hashing mechanism must avoid creating hot spots (i.e., assigning a large number of threads to a single PE that can create large resource contention and generate large traffic towards other PEs) in the chip. An imbalance in the load distribution quickly and significantly increases the i.e.power and temperature of the more stressed portion of the chip, thus contributing to a decrease in the overall reliability, which may accelerate device ageing. Load imbalance can also create congestion in the network since some links drive more traffic than others. On the other hand, the hashing mechanism used to distribute the threads must guarantee the locality of computations. To this end, our hashing scheme allows placing a group of dependent threads (i.e., asynchronous function) on the same group of PEs (i.e., VN), while still preserving a fair thread distribution within the VN by randomly selecting PEs. Similarly, the hashing scheme allows the scheduling of AFs randomly on different VNs across the whole chip, avoiding the need to explicitly find an allocation of PEs that minimise the communication distance between producer and consumer threads (in a minimal amount of clock cycles). Furthermore, our PXM implies more than one producer thread can generate input data for a single thread, and producers can be scheduled and executed at different points in time.

Each thread in the system is identified by a unique *thread identifier* (T_{id}) over the whole application execution. It is composed of three main fields: the *source field*, the *destination field*, and a local counter (*CNT*). The source and destination fields are in turn, formed by two sub-fields representing the *core identifier* of the PE (C_{id}), and the *virtual node identifier* (N_{id}). While the content of the source field is fixed for each tile (once the number of cores in each VN has been selected), the content

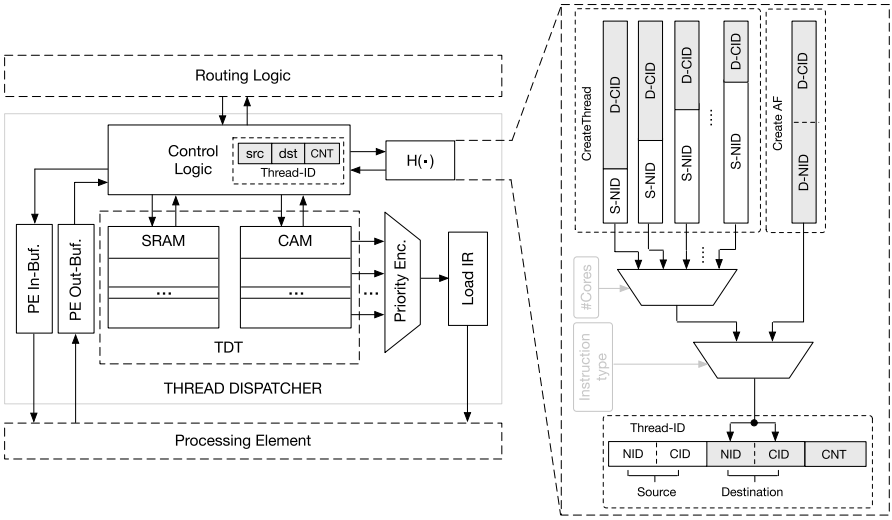


Fig. 5 Thread Dispatcher module organisation (left) with the internal structure of the $H(\cdot)$ function (right)

of the destination field is produced at run-time by the hashing function $H(\cdot)$. These fields allow the system to generate unique identifiers that are conveniently stored in a 64-bit register.

The organisation of the T_{id} is illustrated in Fig. 5. By passing to the $H(\cdot)$ module both the size of VNs (N_{pe}) and the indication of the executed instruction I_{ex} (i.e., the `CreateThread` or the `CreateAF` instruction), it uniquely identifies the PE responsible for the execution of the newly generated thread (i.e., $\langle N_{id}, C_{id} \rangle_{dst} = H(N_{pe}, I_{ex})$). Once selected, the PE is signalled by sending a message over the network. Since the destination is encoded in the T_{id}, any subsequent operation on the thread can easily be forwarded to its corresponding PE without any calculation. This contributes to the overall speedup of the system.

The basis of our system is a 2D mesh NoC implemented with lightweight ring-based routers (see Sect. 4.2.2). Such routers allow the implementation of a flexible 2D mesh topology on top of four unidirectional rings. An internal table describes how traffic flows in the links. The internal crossbar switch is substituted with four ring stations, each capable of driving network traffic in the same direction or steering it to the opposite dimension. The ring stations are coupled with two additional modules (*inter-ring switch* – IRS) that are responsible for ejecting traffic travelling in one dimension or injecting traffic in the opposite dimension.

Thread descriptor table (TDT) is a data structure used to manage threads organised in two fixed-size local memory arrays. The total area is comparable to an L1 instruction cache (see Sect. 5.2). It is interesting to note that we may 'lose' some information while having the TDT, but we can reclaim some of it because we use scratchpads and not actual caches for the thread frames. Input and intermediate data are stored in the scratchpad memory. Every time the context of a thread is updated, the T_{id} is used as a search key within the content-addressable memory. In the case

of a match, the returned base address of the frame F_b is added to an offset F_o to determine the location access (i.e., $l = F_b + F_o$). Finally, a priority encoder selects the thread with the lowest T_{id} among those runnable (SS = 0, see Fig. 5). Every time the selected PE is devoid of free resources, it can access a larger but slower and less energy-efficient memory area called *Thread Storage*, which is implemented as a 3D stacked DRAM layer[2]. It is organised by banks (one for each PE) representing a larger TDT structure. When a PE receives a new thread, it first selects the entry in the local TDT and compares the SS value of the new thread with the one currently stored. The thread with the highest SS will be swapped to the DRAM memory bank.

Hash scheduling function $H(\cdot)$ is used to map new threads to PEs for their efficient execution and contains a set of maximum-length linear-feedback shift registers. In our case, the hardware module assigns to the newly created thread the tuple $\langle N_{id}, C_{id} \rangle_{\text{dst}}$, depending on the VN size and the executed instruction. To be effective, the scheduling function $H(\cdot)$ has to distribute `CreateAF` and `CreateThread` requests among the available resources fairly. The effectiveness of the hashing function derives from the ability to avoid collisions, i.e., to limit the number of times two distinct input values result in the same output value for the hashing. In our distributed scheduling scheme, this translates into avoiding different PEs selecting the same destination, given two different T_{id}. In that case, the PEs' load (i.e., the number of threads to execute) is balanced, thus avoiding the formation of hot spots and increasing the overall system reliability.

A good hash function must provide *determinism*, meaning that it has to provide the same set of hash values for the same set of input keys. More importantly, hash functions must exhibit *uniformity*: given a set of n input keys and m output buckets, each bucket must show a load $\lambda = \frac{n}{m}$. It directly translates to the same probability for each output bucket to be selected, thus limiting the number of collisions. Finally, since we are not considering cryptographic applications, it is not strictly required for the function being *non-invertible*. Indeed, it is more desirable to keep hardware implementation efficient regarding area and power consumption.

The mentioned hashing scheme offers good results while maintaining a low area overhead and preserving the capability of dynamically changing VN sizes (see Sect. 5 for more details). Another important aspect of our scheme is that it works in an entirely distributed fashion, meaning that there is no single point of failure, as is desired in a system potentially equipped with thousands of PEs.

4.2 Software-controlled NoC reconfigurability

Here, we propose a microarchitecture that tries to merge the benefits of ring-based NoCs (i.e., performance and energy efficiency) with those brought by dynamic reconfiguration (i.e., adaptation, fault tolerance) while keeping the hard-wired 2D mesh topology fixed. To accommodate different application requirements and communication patterns, the proposed interconnect maps different types of topologies

[2] Alternatively, 2.5D implementation is possible using a chiplet-based design of the entire target system, while getting even slower accesses due to the availability of few shared memory controllers.

(or virtual topologies) over the physical ones. A few customised instructions are added to the core ISA to allow the software layer to control the network's topology during execution directly. Thanks to the flexibility in managing the underlying interconnection subsystem at the application level [51], the software can control every single link with fine granularity using ring-based interconnects as the building block for implementing the physical 2D mesh topology. As a result, physical connections can be switched off when not in use. Link usage is monitored using internal hardware counters and accessible through dedicated instructions. This information can be exploited by the programmer and the optimisation tools or compiler to better adapt the virtual topology to the application's communication patterns. It is worth saying here that, although demonstrated targeting a ring-based physical topology, the underlying physical layer can be constructed using more conventional routers based on crossbar switch elements or a mix of the two.

4.2.1 Software interface

The ISA extension we presented in Sect. 4.1.1 has been further extended with a few instructions to manage the reconfiguration phases and to monitor the links' traffic. The combination of both proposed ISA extensions allows us to deal with a SDNoC. As in the case of instructions used to manage the dataflow thread distribution. Here, each instruction can be conveniently wrapped by a function in standard high-level languages (e.g., C/C++) as follows:

- `SetRouterCfg(Rd,Rs,B)` Sends a configuration request to the routers by specifying the memory address where the configuration is stored. Variable `Rd` specifies the destination router, variable `Rs` contains the memory address where the configuration is stored, and flag `B` (unsigned immediate value) indicates if the request is sent in broadcast to all routers (`B`>0), or not (`B`=0);
- `ReadCounter(Rd,Rs1,Rs2)` Reads the content of a link's counter, by specifying the link to read (one of the four bits starting from the LSB position in the variable `Rs1` must be set), the destination router (variable `Rs2`), and the variable where the counter content will be stored (`Rd`);
- `ResetCounter(Rd,Rs,B)` Allows to reset traffic statistics by specifying which links' counters to reset (four bits starting from the LSB position in the variable `Rs` are set if the corresponding links' counter must be reset), the destination router (variable `Rd`), and the flag `B` that indicates if the request is sent in broadcast to all routers (`B`>0), or not (`B`=0).

Each router is equipped with a 16×1 SRAM containing the *switch table* (ST). It describes how traffic entering a link can flow into other links. More precisely, the memory content is logically arranged as a 4×4 matrix. Each row of the matrix corresponds to an input link, while the four columns in a row represent output links. An element $s_{i,j} \in ST$ is set to one if the traffic travelling in the direction i (e.g., west) can be forwarded on the direction j (e.g., north). The content of the ST, and the values of bits enabling the bypass and power-gating logic (respectively BP and PG bits), are stored in memory in the form of a bitstream. Every time the `SetRouterCfg`

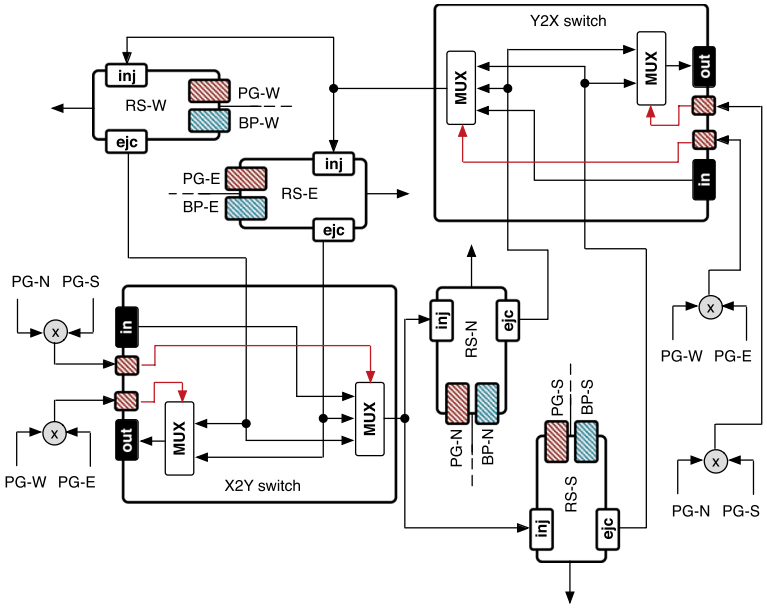


Fig. 6 Lightweight router microarchitecture: ring stations (RSs) have injection and ejection ports, as well as bypass (blue squares) and power-gating (red squares) ports which are controlled by relative bits. Inter-ring switches are power-gated depending on the state of the RS (grey circles are AND gates)

instruction is executed, it generates a corresponding message sent to a specific router. The destination router directly accesses the memory location (actually, the PE performs this operation), where the configuration is stored, by adding a fixed offset to the basic address (`Rs`). The operation can be parallelized if multiple routers need to be configured.

4.2.2 Hardware support for virtual mapping

Figure 6 merges the benefits of lightweight and scalable router microarchitecture with those of dynamic reconfiguration [40, 52]. Four separate rings allow communication to flow in the north, south, east and west directions, while specific bits control the status of each link. The communication in each of the four directions (i.e., north (N), south (S), east (E), and west (W)) is ensured by dedicated *ring stations* (RSs), which are essentially responsible for forwarding the traffic coming from the injection port (inj) to the output port or the ejection port (ejc). An RS manages the traffic from a physical ring in the same direction. During the configuration phase via the customised instruction, the links can be set in three different modes: *normal*, *bypass* (BP), or *power-gated* (PG). Bypass and power-gated modes disable links partially or fully, thus allowing the network to save power and implement different topologies.

The original router design uses a single *inter-ring switch* (IRS) to switch traffic travelling horizontally (i.e., E-to-W, or from W-to-E) to one of the vertical directions (i.e., N-to-S, or from S-to-N), but prevent the opposite traffic steering (i.e., from

vertical to the horizontal direction) [52]. This choice was imposed by implementing the X-Y routing algorithm on top of the ring-based architecture. Although this organisation ensures chip-level communication within the physical mesh, it does not support mapping virtual topologies.

To compensate, we provide a more flexible architecture: two symmetrical IRS, i.e., an X2Y switch and a Y2X switch allow the traffic to spill from horizontal directions to vertical ones, and vice-versa. To this end, the design uses the X-Y deterministic routing algorithm without the restriction of forwarding traffic first horizontally and then vertically, i.e., packets can be routed alternatively on the X and Y dimensions as a first routing step. Each IRS is composed of two multiplexers (MUXs). One is responsible for selecting which of the inputs (e.g., traffic coming from the local PE, E-to-W direction, or W-to-E direction) has to be transferred to the injection ports of the RS in the opposite dimension (e.g., the traffic is injected in the N-to-S or the S-to-N direction). The second one selects which traffic to present to the output port (i.e., this is the spill point, where traffic coming from the network is presented to the PE). For instance, the X2Y switch can present to the output port of the traffic coming from the ejection port of the RS-W and RS-E (see Fig. 6). The other two separate ports in each IRS (highlighted in red) control their state, thus allowing for selectively power-gating one of the two internal MUXs or both. Considering the X2Y switch (similar to the Y2X switch), the MUX connected to the output port can be disabled only when the RS-W and RS-E are in power-gated mode. The MUXs are responsible for forwarding the traffic in the opposite dimension are power-gated when both RS-N and RS-S are in power-gated mode. Similarly to IRS, each RS is provided with two additional ports controlling the station's state. A power-gated port (highlighted in red) entirely disables the RS, dropping traffic at the input port. A bypassed port (highlighted in blue) partially disables the RS, allowing the traffic arriving at the port of entry to be directly injected into the output link. These ports are controlled by two corresponding configuration bits: PG and BP. While the configuration of the IRS depends on the state of the RS controlling them, the state of an RS can be set independently from the others. Furthermore, the power-gated mode is dominant in the bypass mode, meaning that the configuration $\langle BP, PG \rangle = \langle 1, 1 \rangle$ leads the RS to be set with the power-gated mode.

The internal organisation of an RS is depicted in Fig. 7. Traffic arriving at the input port (IN) can continue travelling in the same direction or be extracted. This decision is implemented within the demultiplexer (DMUX) as part of the routing logic and requires only analysis of the destination field of incoming packets. Every time input traffic has to be ejected (i.e., extracted by the local PE or deflected in the opposite dimension), it is temporarily stored in a local buffer (S-BUF). Similarly, the buffer R-BUF temporarily stores packets that continue to flow in the same direction. It is worth noting that both buffers can be tiny compared to input buffers in conventional routers. The limited storage space required by the buffer is mainly due to the adoption of a traffic prioritisation policy, which allows the RS to privilege traffic that flows in the same direction as the one coming from the injection port. This policy is implemented as part of the selection mechanism of the output MUX. It is also worth noting that multiple R-BUF and S-BUF buffers can be used to support virtual channels (VCs). Another MUX (i.e., BP-MUX) selects which traffic to

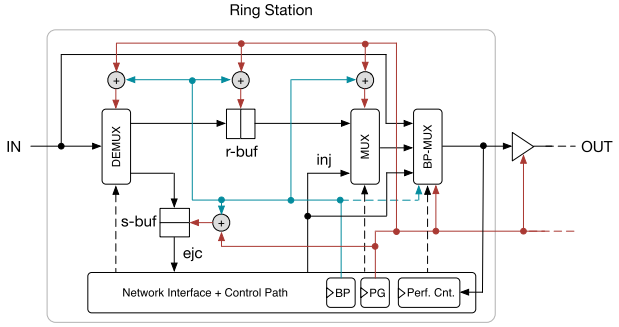


Fig. 7 The internal structure of an RS with BP/PG bits and the link counter in the network interface (dashed lines represent selection signals for MUXs/DMUXs, grey circles are OR gates)

inject on the output link (OUT) when the RS is set in the bypass mode. In this mode, both the internal buffers, the input DMUX, and the output MUX is disconnected from the power source (power-gated). At the same time, traffic on the injection port is directly injected into the output link. In this way, the RS can save energy and limit the packet traversal delay. Interestingly, in this case, the drivers on the output link are still fully active. On the contrary, whenever the RS is set to power-gated mode, all internal components and link drivers are switched off. Four additional OR gates ensure that BP or PG bits control power-gate and bypass functionality for the internal components. As already mentioned, the PG bit overrides the BP signal. Finally, a counter is available for each output link: depending on the granularity at which traffic is monitored and the frequency of captured events, the size of the counter can be varied from small (16-bits) to large (64-bits). Every time a packet (flit) traverses the link, the counter is automatically incremented. A single bit of the control router logic allows selecting the granularity at which traffic is monitored (i.e. if counters are updated when packets or single flits traverse the links). Figure 8 shows an example of mapping a hierarchical topology (i.e., local rings and a global mesh) upon a 4×4 physical 2D mesh.

4.3 NoC performance improvement via hybrid topology

Optimizing the local communication can improve packet latency, throughput, and energy efficiency [37]. Here, a hybrid interconnection is designed by combining the ring and 2D mesh topology capabilities, thus achieving better scalability while limiting the latency and energy consumption increment. Rings demonstrated to be very efficient [38] when the number of cores is (relatively) small. It has been used to create an effective 2D mesh topology (see Sect. 4.2.2). However, this design encounters a reduced advantage when the number of cores becomes very high. Indeed, while the area and power cost of RS and IRS (used to steer traffic between vertical and horizontal dimensions) may slightly reduce, the latency and throughput are strongly affected. In this case, communication between distant PEs is influenced by the need to traverse many physical hops.

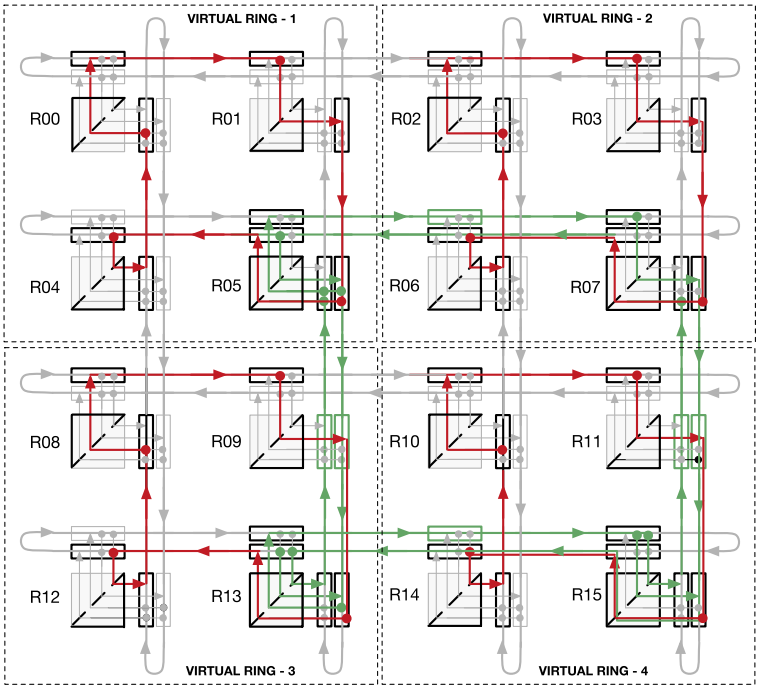


Fig. 8 An example of virtual topology mapping: grey structures represent components (i.e., interconnections, RSs or IRSs) of the router that are power-gated. Red lines correspond to active links used to build local rings, while green lines show links of the mesh. Furthermore, green boxes represented components set in bypass mode and used to construct the mesh among the virtual rings correctly

On the contrary, a hierarchical organization of the physical network can significantly alleviate the latency for long distant communication patterns while keeping the high throughput experienced for more localized communication patterns. Moreover, such a hybrid approach (combining a local high-speed topology and a global one) can still benefit from the integration of the reconfigurability mechanism exposed in the previous section, especially with a very high core count (e.g., CMPs with thousands of cores), leading to the possibility of creating even deeper (virtual) hierarchies.

Figure 9 depicts an illustrative presentation of the proposed architecture. Here, we propose a network architecture to connect many PEs (with more than 256 PEs) and use the deterministic and deadlock-free X–Y routing algorithm. Four PEs are locally connected through a small ring called *ringlet*. Within a ringlet, one of the PEs is designated as the master core. It is responsible for injecting/ejecting traffic towards the global traffic channels. A link between the ring switch and the mesh router is enabled, along with dedicated buffers. The advantage of this architecture is the absence of a dedicated bridge component to connect the mesh and the ringlets. The proposed NoC is divided into blocks, and a group of four ringlets forms a *block* unit. These four ringlets are directly linked to the mesh router, which

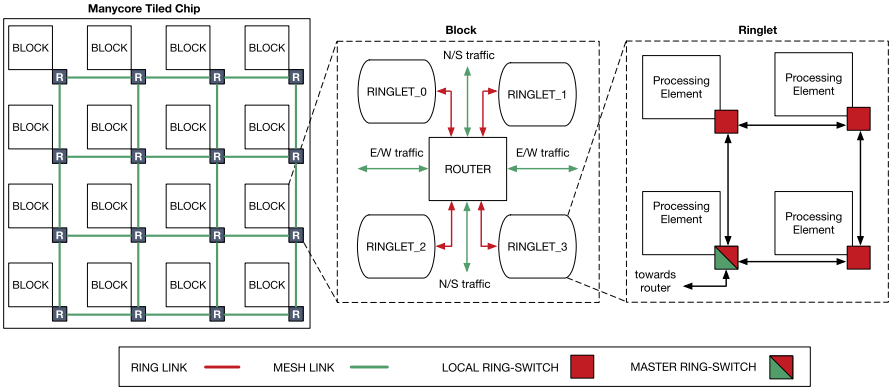


Fig. 9 An illustrative presentation of the proposed scalable NoC: 256 PEs organised into 4×4 block units, each connecting four ringlets. Each ringlet contains 4 PEs

is responsible for moving traffic outside the *block*. For instance, to support 256 cores, 16 modified mesh routers and 64 ringlets are needed. More complex traffic management policies can be applied at the block unit level, and multiple blocks are globally connected through a 2D mesh topology. To support traffic flow in the mesh network, each router has a high-performance 8×8 internal crossbar switch. Effective packet processing implemented in mesh routers allows decoupling of global and local traffic, as it is recommended to prioritize the local congestion management [37]. Every time a packet's destination is outside the local block, the packet is forwarded to another mesh router, thus bypassing PEs in the ringlets and minimizing the overall latency.

Next, we describe the primary components of the proposed NoC microarchitecture. Specifically, the internal organisation of the proposed mesh router and the ring switch.

4.3.1 Modified 2D mesh router

Figure 10 depicts the internal organisation of the mesh router, while Table 1 provides its main architectural characteristics. The router employs an 8×8 crossbar switch to support global traffic movement in both dimensions (i.e., N-S and E-W) and traffic exchange with local ringlets. Four channels are used to drive traffic within the 2D mesh network (i.e., global traffic). The other four input channels are used to steer traffic to/from the ringlets (i.e., local traffic). Each ringlet is associated with a dedicated channel, so the traffic exchange with the master ring switch happens through this dedicated link. In general, routers can have a significant number of VCs to hold a large amount of incoming traffic. For each input link, two VCs are introduced, which allow for better QoS support and also prevent deadlocks. In Figure 10, the path taken by the control information carried by the packet headers is highlighted in red, with blue lines showing the control signals activated by the internal router stages.

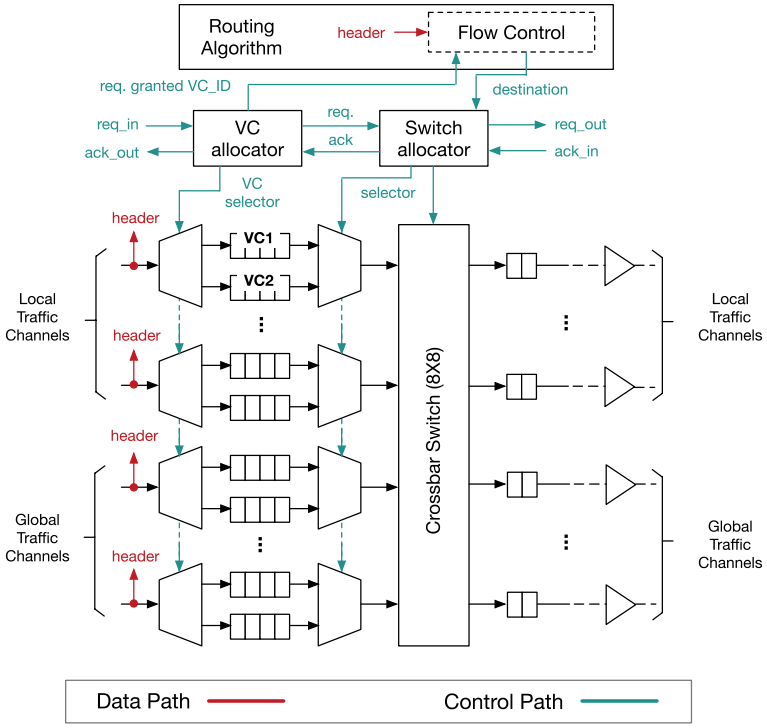


Fig. 10 Modified 2D mesh router microarchitecture: two groups of local/global channels are used to manage traffic within the 2D mesh and traffic exchange with local ringlets

Table 1 2D mesh router: primary microarchitectural parameters

Features	Parameters
No. of input and output ports	8 each (4 ringlets, 4 mesh)
Width of each port	42-bits (32-bits payload, 10-bits header)
No. of virtual channel	2 per input port
Packet switching	Store-and-forward (SAF)
Switch allocator arbitration	Round-robin
Packet routing	X–Y dimension order routing
Router pipeline stages	4 stages

Conversely, output channels do not use VCs, thus contributing to saving power. Large buffer requirements and QoS overheads reduce the ability to support many cores with an efficient area and energy usage [49]. Also, a large number of VCs consume a huge chunk of energy since more input buffers are needed to keep traffic separated. It is worth mentioning that buffers are one of the largest leakage

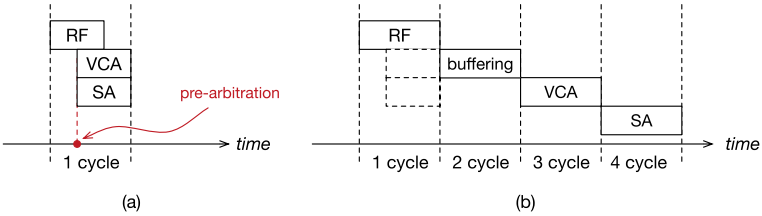


Fig. 11 Timing: (**a**) best-case (success) and (**b**) worst-case (failure) of pre-arbitration

power sources in the router. Their power consumption can represent up to 64% of the total router's leakage power [53], and also a significant amount of dynamic power [54].

Interestingly, single-flit packets represent a large segment of the network traffic for real applications [55]. Following this, the router is optimised for managing a single flit packet. The design selects a packet length of 42-bits, where 32-bits are used to transport data, and the remaining 10-bits are devoted to carrying header information. The packet size has been chosen considering the performance improvements [56] and also the overhead, i.e., increasing the packet size leads to a quadratic increment of the internal crossbar switch overhead.

The internal router is organised into a four-stage pipeline: routing stage, flow-control stage, VC allocation stage, and switch allocation stage. However, to reduce the latency of the packets to traverse the router, the proposed design has been optimised so that the operations performed by the four stages can happen in parallel, thus reducing the overall latency to one cycle. The entire packet transfer can be restricted to a single cycle thanks to the following design choices: (*i*) the adoption of the store-and-forward mode; (*ii*) the optimisation of the routing logic for processing a single-flit packet with a reduced overall size. These design choices lead to a router architecture with a latency of one cycle in most cases (see sect. 5.4). Wormhole and virtual-cut-through are not chosen because, in this case, the routing is parallel, and the entire packet will be routed out simultaneously. Hence, both of them do not offer any advantage compared to the store-and-forward approach. The employed routing mechanism is based on the X-Y dimension order routing (XY-DoR) algorithm since it provides a simple implementation with a deterministic routing latency. By decoupling the traffic between local ringlets and 2D mesh, the probability of congestion in the 2D mesh level becomes negligible, so the need for an adaptive algorithm disappears [37].

In particular, the routing logic with the flow-control module is fused. A speculative allocation technique for both the VC allocation stage (VCA) and the switch allocator stage (SA) has also been implemented. If the pre-arbitration fails, the packet is buffered while VCA and SA arbitration are performed sequentially. In that case, the latency increases up to four cycles. The timing of the proposed 2D mesh router in the event of pre-arbitration success is shown in Fig. 11a, while the event of failure is represented in Fig. 11b. Every time there is an incoming packet, the following operations are performed by the router's modules:

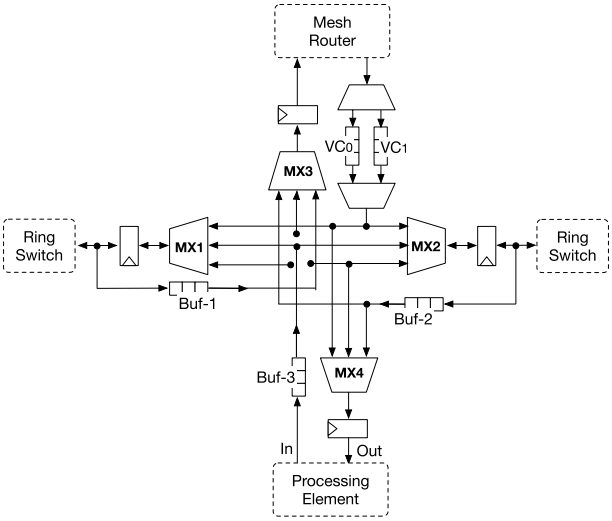


Fig. 12 The microarchitecture of the RSW of the ringlet's master: horizontal dimension is used to create the bidirectional ring connection, while the vertical dimension connects the mesh router and local PE of the ringlet

- *Routing/flow control module (RF)* Extracts the packet header and processes the information to determine the destination router. Suppose the packet destination is within one of the four ringlets belonging to the block. In that case, the RF module selects the corresponding output channel, reducing the latency of the VCA and SA modules. A control signal drives the input MUX at the input port.
- *VC allocator module (VCA)* Is responsible for allocating buffer resources for incoming packets by selecting one of the VCs. An allocation request signal (i.e., req_{in}) is set, and if the selected VC has space to buffer the incoming packet, an acknowledgement signal (i.e., *ack_ out*) is set too. In this case, the selected VC is also signalled to both the RF and SA modules.
- *Switch allocator module (SA)* Performs the two arbitration steps. First, multiple VCs in each input port are arbitrated to select one of the available VCs. Then, each of the selected VCs is routed to the selected output port.

4.3.2 Ring switch

A bidirectional ring is implemented upon the structure of a *ring switch* (RSW) to achieve high performance while keeping power consumption low. An RSW comprises two main MUXs that manage the traffic within the ring. An RSW drives traffic within the ring and steers it towards a 2D mesh router or local PE. Figure 12 depicts the microarchitecture of the RSW. The microarchitecture has incorporated

buffers (similar to VCs), which allow traffic management going to/coming from the 2D mesh router.

To avoid complex control logic, an RSW prioritises the traffic travelling in the same dimension (i.e., traffic that remains within the ring and moves in the same direction). Prioritisation also helps to reduce the size of internal buffers (buffers Buf-1 and Buf-2, see Fig. 12). In particular, one of the two directions is selected with high priority, thus moving first in the RSW. The prioritisation mechanism is implemented directly in the control logic of input-output MUXs.

The interface with the local PE is implemented using a dedicated buffer (i.e., Buf-3), which is written by the PE and read by the RSW (i.e., the PE injects traffic in the ring). The buffer is accessible by PE within its address space. Similarly, traffic that is ejected by the ring is collected temporarily in a local output buffer, from where the PE can extract the payload. The interface with the 2D mesh router is implemented similarly: traffic injected into the 2D mesh is stored temporarily in a small buffer, from where it is transferred to the router's input link. Traffic ejected from the 2D mesh router is moved within a VC buffer. When the 2D mesh router tries to access RSW, two VCs are implemented to support resource contention better. From this viewpoint, the RSW implements a round-robin selection strategy between the two VCs to keep control logic simple. When packets move within the ring or between the ring and the 2D mesh router, the following steps are performed by the RSW:

- The MUX of each input port determines the destination based on the packet's header information and the arbitration.
- Packets from the ring ports (see Fig. 12, horizontal dimension) have a higher priority than packets from the processing core or the 2D mesh router. Thus, such packets are moved first from the input port to the output port with minimal delay. This arbitration strategy also ensures that packets already in the main ring traffic flow are quickly routed to prevent the saturation of the network. Specifically, to enable the transfer, the RSW sets the request signal of the next switch in the ring (by following the travelling direction of the packets), waiting for the acknowledge signal to be set by the peer switch.
- When the master RSW receives a request from the 2D mesh router to inject packets into the ring, two available VC buffers are used to store the packets temporarily. If space is available in the selected VC buffer, RSW enables the corresponding acknowledgement signal of the 2D mesh router. Each buffer will take turns sending out packets via round-robin arbiters to exhibit fairness.

It is worth noting that, to minimise the number of resources used by routing structures, RSW modules that are not connected to the 2D mesh router have the same structure depicted in Fig. 12, except for the interface with the router. In that case, this interface has been removed to save area and power.

5 Simulation results

5.1 Simulation environment

Dataflow PXM We generated network traces using an in-house function-accurate simulator and monitored the requests sent to TD units. Activity synchronisation among simulated tiles is achieved using a small number of mutexes to protect shared resources, i.e., essentially the frame memory belonging to a specific PE. More than one thread was able to update it with a new value.

Software-level simulation We used an in-house function-accurate C/C++-based simulator for simulating thread distribution, where the simulation of each PE is bound to a specific core of the host simulation machine. The basic idea is to simulate the execution of concurrent dataflow threads while relying on a subset of the RISC-V ISA. A random traffic pattern and a matrix multiplication kernel have been developed to evaluate the proposed lightweight reconfigurable architecture with a 2D mesh NoC. We have added a small set of custom instructions to that ISA to implement reconfiguration. The simulated manycore design (for the software level) comprised 256 PEs implementing a five-stage RISC-V compliant in-order execution pipeline with 16 KiB I-cache and 16 KiB scratchpad memory. We also integrated a two-stage lightweight router. Such a customised simulation environment supports our proposed instruction set extension (mentioned in Sect. 4.1.1 and 4.2.1). Our tool simulated up to 1024 cores and executed special instructions designed to manage dataflow threads and the NoC configuration. Power consumption and area were modelled by integrating Orion 3.0 [57] and DSENT [58] tools into our simulation tool.

Hardware-level prototyping We used Xilinx Virtex-7 XC7VX690-3 FPGA device[3] (set clock speed to 400 MHz) to implement the proposed hybrid NoC design. The RTL for the entire VHDL design description of 2D mesh routers and ringlets is synthesised. The proposed routers and ringlets were tested under three statistical traffic patterns: uniform random, bit-reversal, and transpose [59]. Bit-reversal and transpose do not support smooth traffic operations. VHDL-based cycle-accurate models for traffic pattern generation have been developed. The network latency (i.e., the time for a packet to move from source to destination, including the time for a packet to cross the channel) and the throughput were evaluated. For the experiment, all packets that need to be independently routed to a dynamically determined destination were synthetically generated. Four packet injection rates (I_r, measured in packets/cycle): specifically $I_r = \{0.25, 0.50, 0.75, 1.00\}$ were used. For $I_r < 1.00$, nodes generating traffic were selected randomly, while with $I_r = 1.00$ (worst case), all the nodes injected packets simultaneously. The stress test ensured the functional capability of the proposed NoC design under both bandwidth and worst/average latency scenarios.

[3] Total no. of Lookup Tables (LUTs): 43300, Total no. of flip-flops (FFs): 86600 and Total no. of block random access memories (BRAMs): 1470.

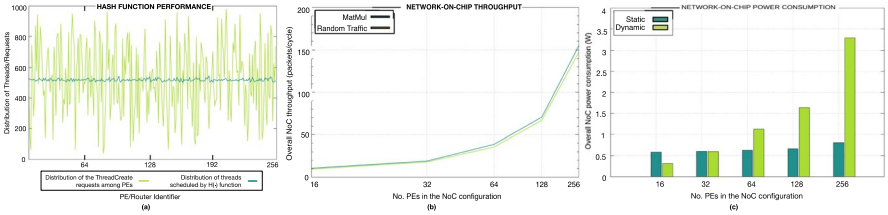


Fig. 13 Distribution of threads on the PEs (**a**), average throughput (**b**), and power consumption (**c**)

5.2 Evaluation of thread distribution

Figure 13a shows the performance provided by the proposed hashing function implementation. The purpose of this experiment is to show how a massive number of input keys for the $H(\cdot)$ module (e.g., thread scheduling requests, write operations) is distributed among the PEs. We simulated a random traffic pattern in the NoC by allowing each tile randomly inject a schedule request (injection rate set to $I_r = 1.0$) towards a randomly selected VN and PE. This pattern is more effective in showing the capability of the hashing mechanism since the traffic cannot be predicted. The green line represents the initial distribution among the PEs of the `CreateThread` requests, while the blue line shows the effective distribution of threads as they have been scheduled by the $H(\cdot)$ modules. The high fairness in the assignments of the threads to different PEs greatly contributes to the high overall performance of the network. Similar results have been obtained simulating the traffic pattern generated by a block matrix multiplication kernel. Random traffic pattern has also been used to assess NoC throughput and power consumption. Such traffic patterns effectively show the capability of our hashing scheme to balance threads' requests, thus avoiding overloading particular links.

Figure 13b shows the average throughput obtained for different configurations. Irrespective of the number of PEs in the system, the throughput grows steadily. Figure 13c demonstrates the power consumption of the combined 2D mesh router and hardware structures for managing the threads distribution, which is (relatively) low. Interestingly, the power consumption (static and dynamic) of such an augmented router microarchitecture remains low compared to that of a basic router based on a crossbar switch. In general, the area and power consumption for the scheduling logic is very low, while that of the TDT is in line with that of an L1-data cache (it is worth noting that the scratchpad memory substitutes the L1 data cache, and represents the main data exchange point between routers and PEs). One limitation of our employed dataflow thread model is that it does not support any jump or allow simultaneous read-write operations.

5.3 Simulating NoC reconfigurability feature

Random traffic represents the worst-case scenario regarding scalability since it uses software to take advantage of the network reconfiguration capability and maintains almost all active links. To simulate this scenario, we instantiated a pool of thread requests serviced by randomly selecting the cores. Also, each core randomly selects the number of requests to consume (e.g., a request corresponds to the scheduling of

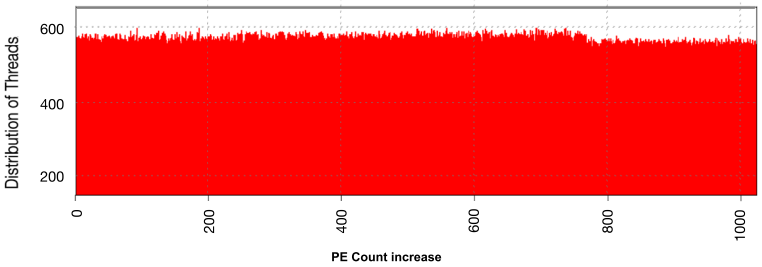


Fig. 14 Distribution of random traffic over 1024-based CMP

a new thread on a different core). The number of processed requests is directly proportional to the traffic generated by the routers on the network.

Figure 14 shows the traffic distribution for a 1024-core CMP with a pool of $580K$ requests. Without loss of generality, a single VC has been used. With this configuration, the overall power consumption of the interconnection is 47.13 W. Traffic generated by data-driven applications is more deterministic, thus offering a greater opportunity for power saving. A matrix multiplication algorithm following a data-driven paradigm using dedicated instructions is implemented (as presented in Sect. 4.1.1 and Sect. 4.2.1). The application has five dynamically scheduled threads, organised as follows: threads in charge of computing the same output element ($C_{i,j} = A_i \times C_j$) exchange data on a local virtual ring, arranged as an 8×2 matrix. A global 2D mesh enforces global communication. The overall power consumption is reduced to 16.20 W while simulating 1024-cores, and the execution time is in line with the execution on a conventional 2D mesh-based CMP. We can speed up the performance by increasing the number of available cores. The reconfiguration of the whole chip requires less than 3000 CPU cycles. When comparing the area occupation, the proposed solution is 39.4% less expensive than conventional routers, offering more opportunity for design scaling. These preliminary results show the benefits of using a dynamically configurable and lightweight interconnection for manycore CMPs. In this configuration, about 37.5% of the links can be switched off while preserving communications at the chip level. The possibility of using high-speed clocks ensures performance, thanks to the simplified router microarchitecture. Furthermore, each link has a counter for tracking traffic statistics whose value is exported to the software layer through a minimal instruction set extension.

5.4 Evaluation of hybrid NoC topology

5.4.1 Resource utilisation analysis

In the proposed design, for the implementation of a single block unit (i.e., four ringlets and associated modified 2D mesh router), the total number of used Lookup Tables (LUTs), flip-flops (FFs) and block random access memories (BRAMs) is

Table 2 Area and power comparison between a standard router architecture and the proposed mesh router

Router	Core support	Resources utilization			Power (W)	
		LUTs	FFs	BRAMs	Static	Dynamic
2D mesh	1	699	572	5	0.323	0.047
Proposed Mesh	16	1358	968	8	0.324	0.075

2434, 2768 and 48 respectively. Specifically, four ringlets consume 1076 LUTs, 1800 FFs and 40 BRAMs. In Table 2, the resource utilisation and power consumption between a standard 2D mesh router and the proposed design are compared. Unlike a typical router, the modified one can support sixteen cores via four ringlets with around 2× increment in the resource consumption compared to a traditional mesh router and with a less than 0.4 W increment of power consumption. Regarding power consumption, the values of static power (due to leakage currents and which depend on the manufacturing process) and dynamic power are presented separately. Results show that the proposed design consumes 1.0 mW and 28.0 mW more, respectively, regarding static and dynamic power. This result is strictly related to a large number of internal memory access by the proposed design.

Next, we compare the resource (area) utilisation of our single block (connecting 16 PEs in total) with the publicly available FPGA-friendly NoC generator CONNECT [60]. It is worth noting that CONNECT only works in 150 MHz, and CONNECT consumed 9600 LUTs, 4576 FFs and 1728 Distributed RAMs (DRAMs), each 64-bits in size to support 16 PEs. In comparison, the proposed model used 2434 LUTs, 2768 FFs and 48 BRAMs, each 36-Kbits in size. It is essential to highlight that such smaller DRAM elements have a higher cost regarding wires than BRAMs. We can see our design saves a substantial number of LUTs and FFs compared to CONNECT[4].

Furthermore, we used 64 modified mesh routers and 256 ringlets to support 1024 PEs. From Table 3, we can see that our model is very resource efficient compared to the standard flattened 2D mesh design. The NoC subsystem consumes up to 155776 LUTs, 177152 FFs and 3072 BRAM blocks (four FPGA devices were used, interconnected to each other through dedicated links). For connecting sixteen cores (one block), our design can save (on average) 2% LUTs, 0.7% FFs and 2.2% of BRAM. It might seem a small saving when the design is scaled to 1024 cores, then the total average resource-saving (over all the four FPGAs) increases to 129.3% for LUTs, 47.2% for FFs and 139.3% for BRAMs.

5.4.2 Energy cost analysis

Figure 15 presents the power consumption comparison between the proposed model and the standard 2D mesh. For one topology block (16 PEs), the power

[4] We used the code available on its official website without any further modification

Table 3 Relative resource utilisation: Values are expressed in percentage scale

	System configuration (No. PEs)						
	16	32	64	128	256	512	1024
Proposed router design							
LUTs	0.31	0.63	1.25	2.51	5.02	10.03	20.06
FFs	0.11	0.22	0.45	0.89	1.79	3.58	7.15
BRAMs	0.54	1.09	2.18	4.35	8.71	17.41	34.83
Ring switch design							
LUTs	0.25	0.50	0.99	1.99	3.97	7.95	15.90
FFs	0.21	0.42	0.83	1.66	3.32	6.65	13.30
BRAMs	2.72	5.44	10.88	21.77	43.54	87.07	174.15
Conventional 2D mesh router design							
LUTs	2.58	2.11	4.23	20.65	41.31	82.61	165.23
FFs	1.06	2.11	4.23	8.45	16.90	33.80	67.60
BRAMs	5.44	10.88	21.77	43.54	87.07	174.15	348.30

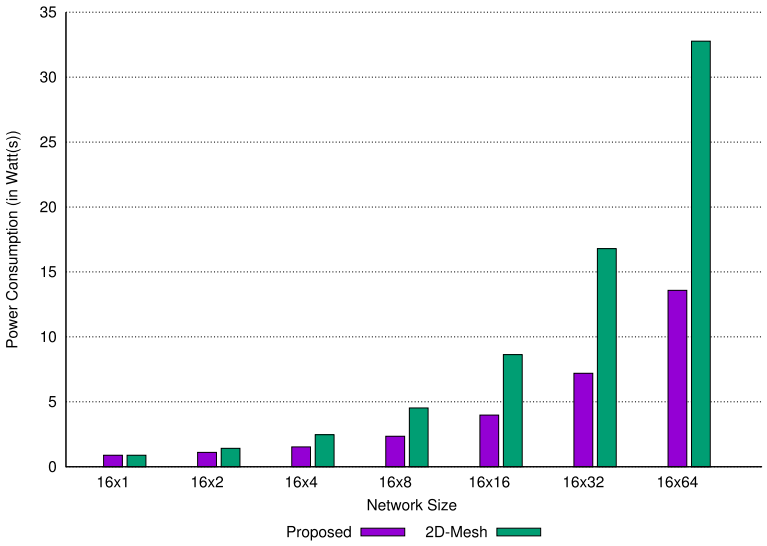
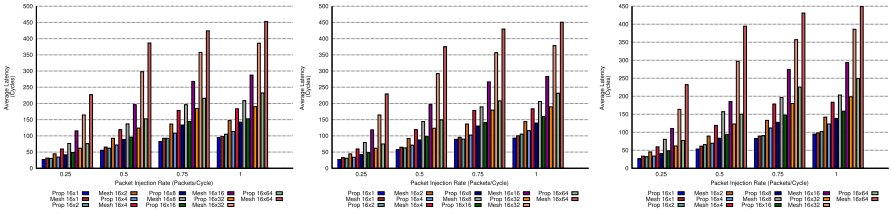


Fig. 15 Total power consumption with increasing network size

consumption is 0.399 W and 0.492 W, respectively, for the mesh router and the ringlet. However, as the network's size grows, the ringlets' total power consumption starts to dominate. For instance, for 16 topology blocks (i.e., 256 cores), the power consumption of routers is 1.276 W while the 64 ringlets consume 2.703 W, which is more than 2× the total router energy consumption. Following this trend, for a 1024-core configuration, all the ringlets consume around 2.5× of the entire router's power consumption. Apart from that, for a network size of 16



(a) Uniform random traffic. (b) Bit-reversal traffic. (c) Transpose traffic.

Fig. 16 Average packet latency

cores, the proposed design and the 2D mesh consume almost the same amount of power. However, as the network grows, the 2D mesh consumes more power. For instance, with a 16×8 cores configuration, the proposed model consumes 2.4 W while the conventional design consumes 4.5 W. The situation worsens when it touches 32.8 W for connecting 1024 cores, which represents 141.26% relatively more power compared to the proposed design.

5.4.3 Performance: latency analysis

Figure 16a,b,c show the average packet latency as a function of the four injection rates when the three different traffic patterns are used. Bars show that the network latency increases with the increased size of the injection rate, and the increase of the network size. Overall, the proposed system offers scalable performance with increasing network configuration. Because of the low injection rates (i.e., $I_r = \{0.25, 0.50\}$), the latency for each traffic pattern remains very consistent with the others. When increasing the injection rate up to $I_r = 0.75$ packets/cycle and using the bit-reversal traffic pattern, the latency is minimised. The worst case for the packet latency is represented by the transpose traffic pattern with an injection rate of 1.00 packets/cycle.

When comparing the proposed architecture with conventional 2D mesh, it was found that for all three traffic patterns, the proposed design outperforms traditional NoC design (by keeping the latency low) thanks to the traffic localisation inside the ringlets. Specifically, analysing the behaviour of the 2D mesh NoC, it is found that the 2D mesh design is very consistent for latency increments for all the cases. At the same time, it also has its largest latency for the transpose traffic pattern with the injection rate of $I_r = 1.00$ packets/cycle (similar to the proposed). For all three traffic patterns, the proposed architecture outperforms the standard 2D mesh by $\approx 10\%$ for the smallest network configurations (i.e., 16 cores) and $\approx 67\%$ for 1024 cores.

We also compared how the proposed hybrid NoC model performed while scaling up the network size. We have presented the average latency (averaging over all three traffic patterns considering four types of injection rates) versus the network size in Fig. 17. In all cases, standard 2D mesh NoC increases latency, while the latency increment in the proposed NoC comparatively is very low. The minimum latency

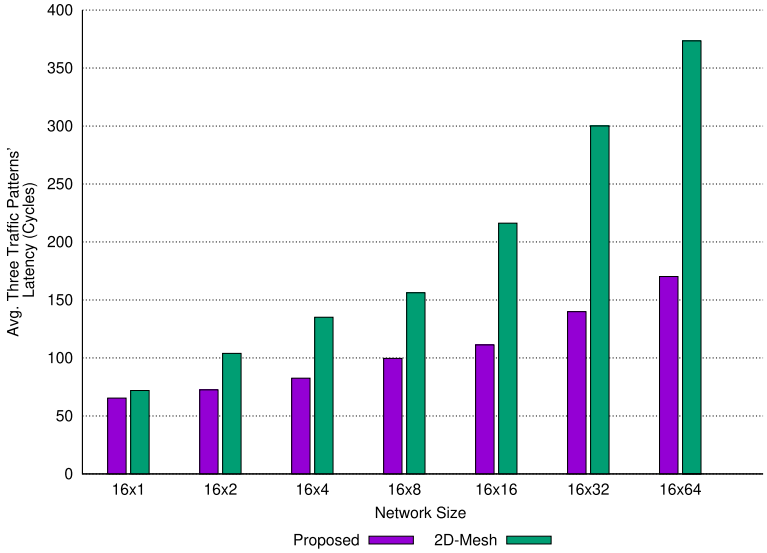
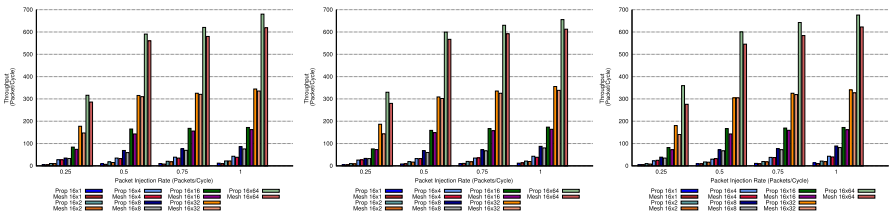


Fig. 17 Comparing the average packet latency of the proposed design with increasing network size



(a) Uniform-random traffic. (b) Bit-reversal traffic. (c) Transpose traffic.

Fig. 18 Average network throughput

saving is 10% for the smallest network size, and as the network size increases, the proposed model improves its performance. Proposed hybrid NoC model's latency improvements climb up more than 100% for 16x32 and 16x64 network sizes.

5.4.4 Performance: throughput analysis

Figure 18a,b,c show the NoC throughput for all three traffic patterns. Like latency, the network throughput is consistent with the offered packet injection rate. In the proposed design, the average throughput increases as the number of PEs increases. From this viewpoint, by analysing the number of packets delivered per cycle, an increase of the 2× factor with the increase in the number of PEs in the network has been observed. For instance, with an injection rate equal to $I_r = 1.00$ packets/cycle and a uniform random traffic pattern, the average throughput increases from

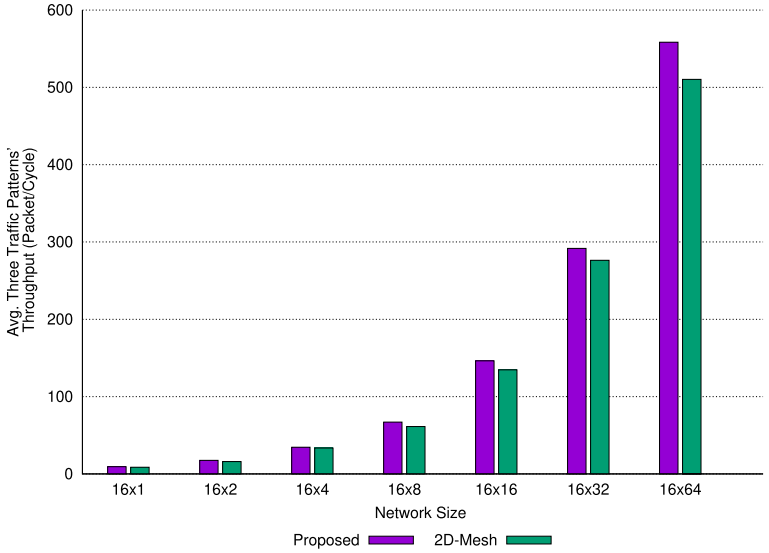


Fig. 19 Comparing the average network throughput of the proposed design with increasing network size

12 packets/cycle for a single block unit (i.e., 16 cores) to 22 packets/cycle for two block units (i.e., 32 cores). Similarly, for a configuration with 512 cores, the average throughput is 344.5 packets/cycle, while it increases up to 680 packets/cycle for a 1024-core configuration. Again, it represents approximately an improvement of a 2× factor with the network size doubling. This trend is also followed by the proposed design when other traffic patterns are considered. It clearly shows that the proposed NoC architecture is capable enough to offer higher performance and scalability compared to traditional 2D mesh. A similar trend in 2D mesh throughput has also been observed. The proposed design has performed better for the transpose traffic pattern[5], when the injection rate is low (i.e., $I_r = \{0.25, 0.50\}$ packets/cycle). For an injection rate equal to 0.75 packets/cycle, the proposed design performed well for all the traffic patterns. The design shows the best throughput for transpose traffic patterns for the largest network configuration (i.e., 1024 cores). However, considering the worst injection rate case, it is worth noting that the best throughput is achieved with uniform-random traffic, while the traditional 2D mesh topology did not demonstrate a similar consistency among the patterns.

These results clearly show that the proposed design can improve the performance of the NoC regarding higher throughput and lower average latency compared to the traditional 2D mesh topology. The capability of this design to sustain such performance also with high injection rates and random traffic patterns (which represent a critical pattern) can be mainly ascribed to the hierarchical organisation of the network.

[5] It represents skewed communication.

Finally, we compared how the proposed hybrid NoC model performs (throughput) while scaling up the network size. We have presented the average throughput (averaging over all three traffic patterns considering four types of injection rates) versus the network size in Fig. 19. It can be seen that in all cases, the proposed NoC outperform traditional 2D mesh. The advantage can be counted in the percentage scale from 3.24% for 16x4 network size to 9.95% for 16 x 2 network size. The proposed design has shown robust performance by improving latency and throughput.

6 Conclusion and future work

The framework mainly provides a hardware-software co-design mechanism for efficiently managing a massive number of concurrent dataflow threads at the NoC level. To improve the run-time adaptability of the dataflow thread management, the framework offers thread distribution and related hardware and software support. First, a hash-based thread distribution mechanism for a data-driven PXM with very low overheads has been proposed. A more complex thread distribution policy can also be employed at an added cost, but performance improvements are not guaranteed. Next, we move the thread control from the software to the NoC level. In this case, the underlying NoC infrastructure must be capable of supporting these threads. Hence, a software-controlled NoC was also proposed. It can reconfigure itself to provide better support to data-driven PXMs and implement a link between the dataflow thread and its physical substrate. To further improve the acceptance level of the framework, we prototype a hybrid NoC design to support both the control-driven and data-driven PXMs. The hybrid NoC has yet to be customised for dataflow threads and needs to be compared with other similar NoCs, which are currently existing limitations. Moreover, the primary aim of this work is to show the possibility of having such an NoC-based framework to improve the run-time adaptability of dataflow threads.

Apart from adding dedicated support to the hybrid NoC level for dataflow PXM, we also aim to integrate one non-intrusive real-time thread distribution monitor that gives thread mapping information. Such information helps map the threads better (in future runs). An analytical model that aims to verify the thread-to-core mapping issue can also be added as a fourth stage. If the thread mapping is not optimal, knowing how much the optimality gap is will be helpful. We also aim to port the hash-based thread distribution mechanism and software-defined NoC model into the FPGA to benchmark it homogeneously.

Acknowledgements It is worth noting that the current manuscript is based on extended versions of three previously-published articles, i.e., thread distribution [17], software-defined NoC [18] and hybrid NoC model [19]. Although the papers are independent, each presents a component of the unified hardware-software co-design-based framework. Overall, the primary research contribution of this work comes after combining the three stages, which allows us to answer the mentioned research question holistically and effectively.

Author Contributions SM: Conceptualization, Software (Validation and Verification), Writing (Original Draft, Review and Editing) and Visualisation. AS: Software (Validation and Verification), Writing

(Review and Editing) and Visualisation. SZ: Writing (Review and Editing). AP: Writing (Review and Editing)

Funding Not applicable. Open access funding provided by Royal Danish Library.

Availability of data and materials Not applicable.

Declarations

Conflict of interest The authors declare that they have no known conflict financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Ethical approval Not applicable.

References

1. Shin W, Oles V, Karimi AM, Ellis JA, Wang F (2021) Revealing power, energy and thermal dynamics of a 200pf pre-exascale supercomputer. In: Proceedings of the international conference for high performance computing, networking, storage and analysis. Association for computing machinery. New York
2. Schneider D (2022) The Exascale Era is upon us: the frontier supercomputer may be the first to reach 1,000,000,000,000,000,000 operations per second. IEEE Spectr 59(1):34–35. https://doi.org/10.1109/MSPEC.2022.9676353
3. Sato M, Ishikawa Y, Tomita H, Kodama Y, Odajima T, Tsuji M, Yashiro H, Aoki M, Shida N, Miyoshi I, Hirai K, Furuya A, Asato A, Morita K, Shimizu T (2020) Co-design for a64fx manycore processor and "fugaku". In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, pp 1–15. https://doi.org/10.1109/SC41405.2020.00051
4. Jia Z, Tillman B, Maggioni M, Scarpazza DP (2019) Dissecting the graphcore IPU architecture via microbenchmarking. arXiv preprint arXiv:1912.03413
5. Louw T, McIntosh-Smith S (2021) Using the graphcore IPU for traditional HPC applications. In: 3rd Workshop on Accelerated Machine Learning (AccML)
6. Vasiljevic J, Bajic L, Capalija D, Sokorac S, Ignjatovic D, Bajic L, Trajkovic M, Hamer I, Matosevic I, Cejkov A et al (2021) Compute substrate for software 2.0. IEEE Micro 41(2):50–55
7. Lee EA (2006) The problem with threads. Computer 39(5):33–42
8. Hoffmann M, Lattuada A, McSherry F, Kalavri V, Liagouris J, Roscoe T (2019) Megaphone: latency-conscious state migration for distributed streaming dataflows. Proc VLDB Endow 12(9):1002–1015
9. Nowatzki T, Gangadhar V, Sankaralingam K (2015) Exploring the potential of heterogeneous von neumann/dataflow execution models. In: Proceedings of the 42nd Annual International Symposium on Computer Architecture. ACM, pp 298–310
10. Gostelow KP, Plouffe W, *et al* (1977) Indeterminacy, monitors, and dataflow. In: ACM SIGOPS Operating Systems Review. vol 11. ACM, pp 159–169
11. Barrow-Williams N, Fensch C, Moore S (2009) A communication characterisation of splash-2 and parsec. In: Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on. IEEE, pp 86–97

12. Hoskote Y, Vangal S, Singh A, Borkar N, Borkar S (2007) A 5-GHz mesh interconnect for a teraflops processor. IEEE Micro 27(5):51–61
13. Dally WJ, Towles B (2001) Route packets, not wires: on-chip interconnection networks. In: Design Automation Conference, 2001. Proceedings. IEEE, pp 684–689
14. Vangal SR, Howard J, Ruhl G, Dighe S, Wilson H, Tschanz J, Finan D, Singh A, Jacob T, Jain S et al (2008) An 80-tile sub-100-w teraflops processor in 65-nm CMOS. IEEE J Solid State Circuits 43(1):29–41
15. Das R, Eachempati S, Mishra AK, Narayanan V, Das CR (2009) Design and evaluation of a hierarchical on-chip interconnect for next-generation CMPS. In: 2009 IEEE 15th International Symposium on High Performance Computer Architecture. IEEE, pp 175–186
16. Ausavarungnirun R, Fallin C, Yu X, Chang KK-W, Nazario G, Das R, Loh GH, Mutlu O (2016) A case for hierarchical rings with deflection routing: an energy-efficient on-chip communication substrate. Parallel Comput 54:29–45
17. Scionti A, Mazumdar S, Zuckerman S (2018) Enabling massive multi-threading with fast hashing. IEEE Comput Archit Lett 17(1):1–4. https://doi.org/10.1109/LCA.2017.2697863
18. Scionti A, Mazumdar S, Portero A (2016) Software defined network-on-chip for scalable cmps. In: 2016 International Conference on High Performance Computing Simulation (HPCS). IEEE, pp 112–115
19. Mazumdar S, Scionti A (2020) Ring-mesh: a scalable and high-performance approach for manycore accelerators. J Supercomput 76(9):6720–6752
20. Dennis JB, Misunas DP (1975) A preliminary architecture for a basic data-flow processor. In: ACM SIGARCH Computer Architecture News, vol 3. ACM, pp 126–132
21. Papadopoulos GM, Culler DE (1990) Monsoon: an explicit token-store architecture. In: Proceedings of the 17th Annual International Symposium on Computer Architecture. ISCA '90. Association for Computing Machinery, New York, pp 82–91. https://doi.org/10.1145/325164.325117
22. Dennis JB (1974) First version of a data flow procedure language. In: Robinet B (ed) Programming symposium. Springer, Berlin, Heidelberg, pp 362–376
23. Arvind Nikhil RS, Pingali KK (1989) I-structures: data structures for parallel computing. ACM Trans Program Lang Syst 11:598–632. https://doi.org/10.1145/69558.69562
24. Halbwachs N, Caspi P, Raymond P, Pilaud D (1991) The synchronous data flow programming language LUSTRE. Proc IEEE 79(9):1305–1320. https://doi.org/10.1109/5.97300
25. Bhattacharyya SS, Murthy PK, Lee EA (1999) Synthesis of embedded software from synchronous dataflow specifications. J VLSI Signal Process 21(2):151–166. https://doi.org/10.1023/A:1008052406396
26. Duran A, Ferrer R, Ayguadé E, Badia RM, Labarta J (2009) A proposal to extend the OpenMP tasking model with dependent tasks. Int J Parallel Program 37:292–305. https://doi.org/10.1007/s10766-009-0101-1
27. Nemawarkar SS, Gao GR (1996) Measurement and modeling of earth-manna multithreaded architecture. In: Proceedings of MASCOTS '96 - 4th International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, pp 109–114. https://doi.org/10.1109/MASCOT.1996.501002
28. Theobald KB (1999) Earth: an efficient architecture: for running threads. PhD thesis, McGill University, Montréal Québec
29. Vishkin U, Dascal S, Berkovich E, Nuzman J (1998) Explicit multi-threading (XMT) bridging models for instruction parallelism. In: Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures. ACM, pp 140–151
30. Pell O, Mencer O, Tsoi KH, Luk W (2013) In: Vanderbauwhede W, Benkrid K (eds) Maximum performance computing with dataflow engines. Springer, New York, pp 747–774. https://doi.org/10.1007/978-1-4614-1791-0_25
31. Yazdanpanah F, Alvarez-Martinez C, Jimenez-Gonzalez D, Etsion Y (2014) Hybrid dataflow/von-Neumann architectures. Parallel Distrib Syst IEEE Trans 25(6):1489–1509
32. Zuckerman S, Suetterlein J, Knauerhase R. Gao GR (2011) Using a codelet program execution model for exascale machines: position paper. In: Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era. ACM, pp 64–69
33. Suettlerlein J, Zuckerman S, Gao GR (2013) An implementation of the codelet model. In: Wolf F, Mohr B, an Mey D (eds) Euro-Par 2013 parallel Processing. Springer, Berlin, pp 633–644
34. Bolotin E, Cidon I, Ginosar R, Kolodny A (2004) Cost considerations in network on chip. Integr VLSI J 38(1):19–42

35. Parikh R, Das R, Bertacco V (2014) Power-aware NoCS through routing and topology reconfiguration. In: 2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC). IEEE, pp 1–6
36. Murali S, De Micheli G (2004) Sunmap: a tool for automatic topology selection and generation for NoCS. In: Proceedings of the 41st Annual Design Automation Conference. ACM, pp 914–919
37. Singh R, Bohra MK, Hemrajani P, Kalla A, Bhatt DP, Purohit N, Daneshtalab M (2022) Review, analysis, and implementation of path selection strategies for 2D NoCS. IEEE Access. https://doi.org/10.1109/ACCESS.2022.3227460
38. Ravindran G, Stumm M (1997) A performance comparison of hierarchical ring-and mesh-connected multiprocessor networks. In: High-Performance Computer Architecture, 1997, Third International Symposium on. IEEE, pp 58–69
39. Hamacher VC, Jiang H (2001) Hierarchical ring network configuration and performance modeling. IEEE Trans Comput 50(1):1–12
40. Kim J, Kim H (2009) Router microarchitecture and scalability of ring topology in on-chip networks. In: Proceedings of the 2nd International Workshop on Network on Chip Architectures. ACM, pp 5–10
41. Deb D, Jose J, Das S, Kapoor HK (2019) Cost effective routing techniques in 2D mesh NoC using on-chip transmission lines. J Parallel and Distrib Comput 123:118–129
42. Manzoor M, Mir RN et al (2022) PAAD (partially adaptive and deterministic routing): a deadlock free congestion aware hybrid routing for 2D mesh network-on-chips. Microprocess Microsyst 92:104551
43. Vazifedunn S, Reza A, Reshadi M (2023) Low-cost regional-based congestion-aware routing algorithm for 2D mesh NoC. Int J Commun Syst. https://doi.org/10.1002/dac.5360
44. Reddy BNK, Kar S (2022) Performance evaluation of modified mesh-based NoC architecture. Comput Electr Eng. https://doi.org/10.1016/j.compeleceng.2022.108404
45. Zhao J, Agrawal A, Nikolic B, Asanović K (2022) Constellation: an open-source SoC-capable NoC generator. In: 15th IEEE/ACM International Workshop on Network on Chip Architectures (NoCArc), pp 1–7. https://doi.org/10.1109/NoCArc57472.2022.9911299
46. Zheng N, Gu H, Huang X, Chen X (2015) Csquare: a new kilo-core-oriented topology. Microprocess Microsyst 39(4):313–320
47. Kim H, Kim G, Maeng S, Yeo H, Kim J (2014) Transportation-network-inspired network-on-chip. In: 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), pp. 332–343. IEEE
48. Koohi S, Abdollahi M, Hessabi S (2011) All-optical wavelength-routed noc based on a novel hierarchical topology. In: Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip, pp. 97–104. ACM
49. Grot B, Hestness J, Keckler SW, Mutlu O (2011) Kilo-noc: a heterogeneous network-on-chip architecture for scalability and service guarantees. In: ACM SIGARCH Computer Architecture News. ACM, vol 39, pp 401–412
50. Bourduas S, Zilic Z (2007) A hybrid ring/mesh interconnect for network-on-chip using hierarchical rings for global routing. In: First International Symposium on Networks-on-Chip (NOCS'07). IEEE, pp 195–204
51. Sandoval-Arechiga R, Parra-Michel R, Vazquez-Avila J, Flores-Troncoso J, Ibarra-Delgado S (2016) Software defined networks-on-chip for multi/many-core systems: A performance evaluation. In: Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems. ACM, pp 129–130
52. Lee J, Nicopoulos C, Lee HG, Kim J (2013) Tornadonoc: a lightweight and scalable on-chip network architecture for the many-core era. ACM Trans Architect Code Optim (TACO) 10(4):56
53. Chen X, Peh L-S (2003) Leakage power modeling and optimization in interconnection networks. In: Proceedings of the 2003 International Symposium on Low Power Electronics and Design. ACM, pp 90–95
54. Wang H, Peh L-S, Malik S (2003) Power-driven design of router microarchitectures in on-chip networks. In: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, p 105
55. Ma S, Jerger NE, Wang Z (2012) Whole packet forwarding: Efficient design of fully adaptive routing algorithms for networks-on-chip. In: IEEE International Symposium on High-Performance Comp Architecture. IEEE, pp 1–12
56. Lee J, Nicopoulos C, Park SJ, Swaminathan M, Kim J (2013) Do we need wide flits in networks-on-chip?. In: 2013 IEEE Computer Society Annual Symposium on VLSI (ISVLSI). IEEE, pp 2–7
57. Kahng AB, Lin B, Nath S (2015) Orion3.0: a comprehensive NoC router estimation tool. IEEE Embed Syst Lett 7(2):41–45

58. Sun C, Chen C-HO, Kurian G, Wei L, Miller J, Agarwal A, Peh L-S, Stojanovic V (2012) Dsent-a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling. In: Networks on Chip (NoCS), 2012 Sixth IEEE/ACM International Symposium on. IEEE, pp 201–210
59. Dally WJ, Towles BP (2004) Principles and practices of interconnection networks. Morgan Kaufmann, San Francisco, USA
60. Papamichael MK, Hoe JC (2012) CONNECT: re-examining conventional wisdom for designing NoCS in the context of FPGAs. In: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays. ACM, pp 37–46

Authors and Affiliations

Somnath Mazumdar[1] · Alberto Scionti[2] · Stéphane Zuckerman[3] · Antoni Portero[4]

Alberto Scionti
alberto.scionti@linksfoundation.com

Stéphane Zuckerman
stephane.zuckerman@cyu.fr

Antoni Portero
a.portero@fz-juelich.de

[1] Department of Digitalization, Copenhagen Business School, Solbjerg Plads 3, 2000 Frederiksberg, Denmark

[2] Department of Advanced Computing, Photonics and Electromagnetics, LINKS Foundation, Turin 10138, Italy

[3] Laboratoire ETIS, UMR 8051, CY Cergy Paris Université, ENSEA, CNRS, 95000 Cergy, France

[4] Institute for Advanced Simulation, Jülich Supercomputing Centre, Wilhelm-Johnen-Straße, 52425 Jülich, Germany