Approaching Ideal NoC Latency with Pre-Configured Routes

George Michelogiannakis[1], Dionisios Pnevmatikatos[2] and Manolis Katevenis[1]
Institute of Computer Science (ICS)
Foundation for Research & Technology - Hellas (FORTH) – member of HiPEAC
P.O.Box 1385, Heraklion, Crete, GR-711-10 GREECE
Email: {mihelog,pnevmati,kateveni}@ics.forth.gr

Abstract

In multi-core ASICs, processors and other compute engines need to communicate with memory blocks and other cores with latency as close as possible to the ideal of a direct buffered wire. However, current state of the art networks-on-chip (NoCs) suffer, at best, latency of one clock cycle per hop. We investigate the design of a NoC that offers close to the ideal latency in some preferred, run-time configurable paths. Processors and other compute engines may perform network reconfiguration to guarantee low latency over different sets of paths as needed. Flits in non-preferred paths are given lower priority than flits in preferred ones, and suffer a delay of one clock cycle per hop when there is no contention. To achieve our goal, we use the "mad-postman" [5] technique: every incoming flit is eagerly (i.e. speculatively) forwarded to the input's preferred output, if any. This is accomplished with the mere delay of a single pre-enabled tri-state driver. We later check if that decision was correct, and if not, we forward the flit to the proper output. Incorrectly forwarded flits are classified as dead and eliminated in later hops. We use a 2D mesh topology tailored for processor-memory communication, and a modified version of XY routing that remains deadlock-free. Our evaluation shows that, for the preferred paths, our approach offers typical latency around 500 ps versus 1500 ps for a full clock cycle or 135 ps for an ideal direct connect, in a 130 nm technology; non-preferred paths suffer a one clock cycle delay per hop, similar to that of other approaches. Performance gains are significant and can be proven greatly useful in other application domains as well.

[1]also with the University of Crete, Dept. of Computer Science, Heraklion, Crete, Greece
[2]also with the Technical University of Crete, Dept. of Electronic and Computer Engineering, Chania, Crete, Greece

1 Introduction

Networks-on-Chip (NoCs) are key components of the emerging Systems-on-Chip (SoCs). As SoCs grow in area, complexity and functionality, so do their communication requirements in terms of performance (latency and throughput) and number of interconnected components. Reducing NoC latency is crucial for SoC performance, since it is introduced to every communication pair within the SoC. Latency may become vital in the case of real-time SoCs. It may also play an especially important role in the case of processor units communicating with other processor units, local memory, shared memory or cache blocks. In this paper, we propose a NoC with latency close to the ideal, *i.e.* that of long buffered wires. We examine our proposal in a chip consisting of many processor units and RAM blocks. However, our ideas are general and can be easily adapted to other NoC styles.

We achieve low latency communication by defining and differentiating pre-configured preferred low-latency paths adapting the "mad-postman" [2, 5] technique proposed two decades ago for inter-chip communication networks. Preferred paths are formed by pre-driving tri-state select signals within a switch. Therefore, flits will be eagerly (*i.e.* speculatively) forwarded to their input's preferred outputs. Preferred path delay per hop is solely that of a pre-enabled tri-state driver. Pre-enabling is crucial because these control signals fan out to many bits, thus driving them incurs considerable delay.

Packets may consist of a single or multiple flits [11]. Flits that are eagerly forwarded to a wrong switch output are terminated later in the network as "dead" flits. They are forwarded to their correct output at a lower priority than flits which originate from the input having this output as preferred (if any), and suffer a latency of one clock cycle when there is no contention.

In order to provide preferred paths with flexibility and to be able to distinguish incorrect eager forwarding, we utilize a modified version of XY routing which remains deadlock-free. According to it, a flit is considered to have been correctly eagerly forwarded if it moves closer to its destination in any of the two axes. A flit is considered dead if the distance between it and the destination increased in any of the two axes with its last hop. This way, we can easily distinguish an incorrect eager flit forwarding, as well as a dead flit in the network.

Network reconfiguration is possible at any time by any processing element (PE) or other user block in the network. It is accomplished by sending specially formatted single-flit packets to the switching nodes that need to be reconfigured. Reconfiguration can be requested at any time, but is carefully applied to the switching node to prevent out-of-order delivery of flits belonging to the same packet. Dealing with out-of-order flit delivery complicates the NoC interfaces and is rarely allowed in NoCs.

To fully exploit the mad-postman technique and ensure its proper operation, we take a slightly different approach for switching node architecture than most past research. Our switch resembles a buffered crossbar [8], having one FIFO at each crosspoint and schedulers at each output. The scheduler monitors the FIFOs and the preferred path, and determines which FIFO it can serve next, if any. At each input a combinational routing logic determines if the incoming flit needs to be forwarded to a non-preferred output. If so, it enqueues the flit in the appropriate crosspoint FIFO.

We evaluate our proposed approach on a 2D mesh topology [3] tailored for our target application, *i.e.* processors communicating with RAM blocks. We attempt to minimize the number of switching nodes as well as the NoC overhead by placing one switching node per 4 RAM blocks. The RAM blocks are placed without any free space between them, essentially forming a bigger block. We also investigate floorplan options for our switching nodes by evaluating two different shapes (rectangular and cross-shaped), and outline some modifications to our switch to further reduce occupied area. Topology and floorplan choices, however, do not affect our low-latency contribution and are made according to application and optimization needs.

Simulation results show that, in a 130 nm technology, our design functions at 667 MHz under typical case conditions. It offers preferred path latency of approximately 360 ps per hop that increases to approximately 500 ps per hop when taking into account an 1 mm long wire at each output. This is compared to 135 ps latency for straight buffered wires of a similar length that offer no configuration or routing capability. Non preferred path latency is one clock cycle when there is no contention. Our base switching node design, for 39-bit wide datapaths, occupies an area of 637 μm $\times$ 310 μm when its floorplan is rectangular. We believe that our proposed NoC concept is the means to approach the ideal latency as closely as possible. It may also be combined with orthogonal past NoC research to further improve performance as well as other aspects.

The rest of the paper is organized as follows: Section 2 provides a summary of past NoC research. Section 3 explains the mechanism for pre-configured low latency paths. Sections 4 and 5 present our proposed switch architecture and describe our NoC's topology. Section 6 presents our placement and routing results, and section 7 identifies room for future work. Finally, section 8 provides our conclusions.

2 Related Work

Research has examined performance-enhancement techniques [6, 7, 9]. These approaches are based on pre-computing routing, virtual channel (VC) allocation, and arbitration decisions, as well as speculative pipelines to minimize deterministic routing latency. Implementations of these approaches with VCs and various datapath widths are able to function with a clock frequency of around 500 MHz in technologies ranging from 70 nm to 130 nm. While these approaches can yield per hop latency of one clock cycle, this latency is not guaranteed. These designs suffer higher penalties from contention and blocking delays, that significantly increase latency. Moreover, one clock cycle per hop is their minimum possible latency, while our proposed NoC provides constant minimum per-hop latency, independent of the clock period. Asynchronous approaches achieve 2 ns per hop [1] for highest-priority flits.

Finally, routing algorithms have also been proposed. Many recent NoCs utilize adaptive routing algorithms [4, 12], to route around congested or other problematic areas according to some criteria. As explained in subsection 3.4, our NoC implements a deterministic routing algorithm. Flexibility in preferred paths is already provided. The implementation of adaptive routing algorithms for non-preferred paths is left as future work.

3 Preferred Paths

3.1 Mad-Postman

Mad-postman [2, 5] was introduced in inter-chip packet-switched communication networks. It offered minimal per-hop latency by eagerly forwarding an incoming flit to the same direction in the same axis that it entered the switch from. There was no logic or delay during this forwarding more than that of simple multiplexor or tri-state cell. Incoming flits were also stored in the switch for checking that they were correctly eagerly forwarded. The network strictly followed XY routing algorithm. Thus, a flit was regarded as correctly eagerly forwarded if it followed XY routing. Flits which were incorrectly forwarded remained in storage in the switch and were later sent to the appropriate output. We find that this concept can be applied to NoCs.

3.2 Preferred Paths

The original mad-postman strictly followed XY routing. Therefore, a flit would suffer a routing logic and buffering penalty once at its final hop (in order to be ejected to the local PE output), and possibly once more when it changed axes when traversing the network. We would like our NoC to be able to provide complete paths with the minimum per-hop latency. Moreover, we would like to provide the flexibility to change those paths at run-time to meet various application demands, such as a processor allocating more RAM blocks for itself. To meet these goals, we introduce preferred paths.

Each input is directly connected to a tri-state buffer at each other port's output. Each output has at most one preferred input. That input's tri-state driver is pre-enabled. Therefore, an incoming flit to that input would be eagerly forwarded to each output having this input as preferred. This is achieved solely with the delay of a pre-enabled tri-state driver. Note that an input may have multiple preferred outputs. Thus, preferred paths can fork and simulate a broadcast network if so desired at run-time. However, preferred paths may not converge as only one tri-state may safely drive a wire at any time.

Each input also features a combinational routing logic that examines each incoming flit and determines whether it must be forwarded to an output other than the preferred. If so, it enqueues it in the appropriate crosspoint FIFO to be later forwarded by that output's arbitration logic. A flit needs to be forwarded to an output if it was mistakenly eagerly forwarded. Later hops regard that flit as dead.

Dead flits are not forwarded to any output by the routing logic. They propagate through the network in preferred paths until they reach an input with no preferred outputs. Then, they are either terminated or forwarded in XY manner and possibly enter a circle, as discussed in subsection 3.6. Dead flits occupy resources and therefore may be a nuisance. However, previous research indicates that this effect does not reduce the performance of the network beyond that of virtual cut-through or wormhole networks [5]. If fair arbitration is desired without demands for very low latency at some part of the network, that part can be reconfigured to remove any preferred outputs from switch inputs.

3.3 Packet Format

Packets may consist of a single or multiple flits, in the manner described in [11]. Single-flit packets are used for reconfiguration and read requests. Multi-flit packets are used for transferring multiple words of data to write to a RAM, or from a RAM as a reply to a read request. Flits feature 6 packet ID and 1 flit type control bits. The flit type bit marks the initial flit of a packet as request (single-flit packet) or address, and thereafter data flits with the same packet ID as body or tail. The 32 payload bits contain data in the case of data flits and destination address, byte enables and packet type in the case of address or request flits. Each switch is identified by unique X,Y coordinates. The flit's final destination is determined by two extra bits specifying the user block among the 4 the switch is connected to.

The initial flit of a packet is an address or request flit. Data flits in the same packet have the same ID and will be treated by each switch as the corresponding address flit was. Since flits are eagerly forwarded without being able to process their headers, all flits in a packet will be incorrectly eagerly forwarded in the same way throughout the network. The same applies to the duplicate flit complication, explained in subsection 3.5. Attempting to do otherwise would require combinational logic in preferred path hops, and thus would dramatically increase per-hop latency.

3.4 Routing

Based on our need to accurately classify flits as dead, we choose to implement a deterministic routing algorithm. Non-deterministic (adaptive) routing algorithms introduce the uncertainty in dead flit classification. This is due to the fact that conditions, and therefore adaptive routing decisions, are subject to change at any time. Therefore, the switch currently examining a flit is unsure if the flit's previous hop regarded this switch as the best next hop at the time, or if the flit was incorrectly eagerly forwarded. Since making switches aware of neighbouring network configuration is too costly, we adopt a deterministic routing algorithm.

As a result, we chose a slightly modified version of XY routing. XY routing instructs a flit to first complete its movement in the X axis, and then switch to the Y axis to reach its destination. Our NoC follows this routing algorithm, but is more flexible in allowing eager forwardings that do not adhere to strict XY routing. Specifically, a flit is considered to have been correctly eagerly forwarded, and therefore is not forwarded to another output by the switch, simply if it is approaching its destination in any of the two axes. This may result in a flit reaching its destination via a route that does not comply with strict XY routing.

In the example Figure 1 illustrates, the flit arrives from source S to destination D solely through preferred paths (solid lines). Switch A sees that the flit approached destination D in the Y axis, and therefore regards this eager forwarding as correct. XY routing would have the flit pass through non-preferred paths (dashed lines) and switch dimensions at node B, after having fully completed its traversal in the X axis. Because we would like to provide preferred paths with full flexibility, and also because disallowing these paths by forwarding flits again in non-preferred paths introduces an unnecessary overhead, we choose to modify our XY routing algorithm accordingly. Similarly, a flit is considered dead simply if it moves away from its destination in any of the two axes.

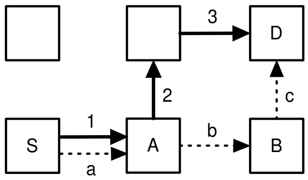


Figure 1. Correct eager forwarding scenario that does not comply with strict XY routing.

3.5 Duplicate Flits

Due to the above mechanisms, our NoC faces the complication of multiple copies of the same flit reaching their destination via different routes. An example of such an occurrence is illustrated in Figure 2. In that example, the flit leaving source S will be eagerly forwarded via the preferred path (solid lines) until it reaches destination D. However, switch A will regard this eager forwarding as mistaken, since it has no preferred path knowledge for its neighbours and the flit's distance from destination D increases in the Y axis. Therefore, it will forward another copy of the flit to destination D via non-preferred paths (dashed lines). Duplicate flits must be handled at the network interface logic of the blocks. Network interface issues are addressed in subsection 5.1.

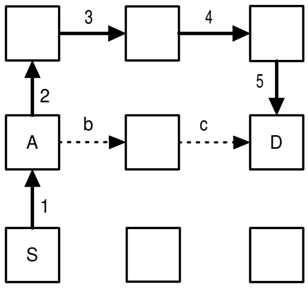


Figure 2. Duplicate flit scenario.

3.6 Deadlock-Freeness

XY routing is deadlock-free [10]. Therefore, deadlock hazards in our NoC are introduced by preferred paths since they do not necessarily follow XY routing and flits propagate in them without any control. Since preferred paths do not content for resources and our NoC follows XY routing otherwise, no flits will indefinitely wait to be served. To guarantee this, a switch needs to be able to serve FIFOs, and therefore resolve contention, if the preferred path has been continuously active for an unreasonably long period of time with a FIFO non-empty. FIFOs are then served until all are empty. During this time, flits arriving in the broken preferred path will be enqueued in the appropriate FIFO behind previous flits of the same packet. However, we need to investigate the possibility of a flit traversing the network indefinitely. We combat this issue in two ways.

First, we provide constraints which, if followed, guarantee that no flit will indefinitely travel through the network. If all preferred paths in the NoC are straight lines, flit propagation follows strict XY routing. Therefore, every turn is handled by routing logic and flits cannot enter a circle. This is the case with original mad-postman networks [2, 5].

Flits cannot enter a circle also in the case of preferred paths having exactly one turn. In this case, circles are formed by four different preferred paths. Therefore, a flit would be examined by routing logic four times before completing a single loop. At these times, the flit will either be considered dead, or it will be propagated according to XY routing. Therefore, in two out of the four checkpoints the flit will be forwarded according to the circle. However, in at least one of the other two it will be forwarded in the other axis and leave the circle.

Preferred paths with two turns may form a circle if the two routing logic checkpoints forward the flit according to the circle. Therefore, if preferred paths in our network contain up to one turn, no flits will indefinitely propagate in circle. This restriction does not take into account turns with switch data ports, as they cannot be part of a circle.

Second, we investigate the consequences of a formed circle. As already described in subsection 3.4, each switch in the circle will examine if the preferred path forwarding was correct and forward a copy as necessary. This guarantees that a copy of the flit in the circle will be delivered to its destination. The flit will continue to propagate inside the circle. Other flits contenting for occupied resources will be forced to wait. If the flit in the circle propagates such as the preferred path is not idle at any clock cycle, contenting flits will face an increased queueing delay. However, as already described in the beginning of this subsection, contenting flits will eventually be served. This poses a performance issue, but no deadlock will occur. When FIFOs are served, the flit in the circle will be examined by routing logic and therefore may be terminated as dead.

3.7 Reconfiguration

Reconfiguration of our network's preferred paths consists of changing outputs' preferred inputs in the appropriate switches. Any PE or other user logic block can request reconfiguration by sending properly formatted single-flit configuration packets. These packets contain the destination node, the output to be reconfigured, and the new preferred input. Configuration flits are enqueued in the appropriate crosspoint FIFO of their destination, even if that flit follows a preferred path. When the configuration flit is selected by the output's arbiter, it is stored in the output's configuration register which stores the active configuration, instead of being forwarded to the next hop.

If we were to immediately alter the tri-state enable signals, we would risk out-of-order delivery of flits belonging to the same packet. Consider the example of Figure 3. Switch S transmits a packet to destination D. The initial flit (flit I) is constantly in non-preferred paths (dashed lines), and therefore is forwarded at every hop by XY routing logic. If a user block in the network was to reconfigure switch A to select input 1 as preferred for output port 2, later flits (flit II) would now reach destination node D via a preferred path (solid lines). Therefore, if flit I is in transit and switch A is reconfigured before the last flits of that packet reach it, those last flits could reach destination D before the first flits.

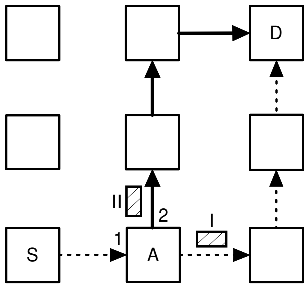


Figure 3. Out-of-order delivery scenario.

Because out-of-order delivery of flits belonging to the same packet can be a nuisance for destinations and also because address flits must always precede the corresponding data flits, it must be prevented by our NoC. This can be accomplished by delaying the application of the new configuration for each output until it is safe. Specifically, once a new configuration is received at an output, it is only applied when the old preferred path has been idle for 1 clock cycle, the new preferred path's FIFO is empty and no more flits from a packet are expected in those paths. As explained in subsection 3.3, flits forming a packet are labelled. Therefore, after receiving the packet's address flit and until receiving the tail flit, the arbitration logic knows that more flits are expected in this specific path and therefore delays the application of the new configuration. Blocks requesting reconfiguration are unsure exactly when it is applied, unless application demands dictate the implementation of a reconfiguration acknowledgment or polling mechanism. Due to this technique, no preferred path will change at each used switch from the time it receives the first flit of a packet until it forwards the last. This ensures that all flits of the same packet follow the same path, and therefore a switch will not wait indefinitely for tail flits. Thus, all flits belonging to the same packet will be delivered in-order.

Different packets from the same source to the same destination may be delivered out-of-order. Since switches have no information regarding more expected packets in the same path, they may apply their new configuration if the preferred path becomes idle for one clock cycle. Even if the source transmits back-to-back, the preferred path may become idle for one clock cycle due to broken preferred paths to avoid starvation effects caused by contention, as explained in subsection 3.6. Therefore, this issue must be handled in the network interface logic as explained in subsection 5.1.

Arbitration logic serves flits stored in FIFOs until all are empty, as described in section 4. If any flits arrive through the preferred path during this time, they are enqueued in their preferred output's FIFO. Arbitration logic can then serve them in priority according to the implemented algorithm. This imposes a single clock cycle delay regardless of contention from other inputs, in this infrequent scenario. However, attempting to do otherwise would introduce extra combinational logic and timing hazards. These flits will still be forwarded according to the preferred path. This serves the purpose of avoiding out-of-order delivery scenarios and taking advantage of preset multi-hop preferred paths.

3.8 Backpressure

Our NoC needs to provide a mechanism for not dropping flits due to full FIFOs. This mechanism must inform each output in a switch whether it can safely transmit a flit to the next hop. Likewise, the previous switch would also be informed if it can safely transmit to the current. Therefore, long packets reaching a congestion point will be stored in many, possibly consecutive, switches. Since flits of the same packet are guaranteed to follow the same path and arrive in-order, no recombination care must be taken.

Depending on area constraints and traffic patterns, we can adopt two different approaches. According to the first, if any of the next hop's FIFOs that have the output port in question as their input is almost full (to cover for backpressure signal propagation delay), the output's arbiter is alerted to not transmit any more flits until the signal is de-asserted and it is safe again. This approach requires only one wire from each output's next hop and the simplest logic.

According to the second approach, one wire from each of the next hop's FIFOs that have the output in question as their input alerts the switch exactly which crosspoint FIFO is almost full. This way, the arbiter needs to process packet IDs of flits in FIFO heads to determine if it can safely transmit any of them. With this approach, FIFOs that are not full are able to receive flits, and thus communication and FIFO utilization is more efficient. However, 6 wires are required at each output and also the arbiter must be able to process flit packet IDs in each FIFO head.

Extra care must be taken for flits forwarded in preferred paths. In our NoC we cannot know the final destination of flits travelling in preferred paths before they have been eagerly forwarded, nor can we control their transmission. Therefore, if a backpressure signal to an output port is asserted, the preferred path leading to this output is broken. Thus, all subsequent flits in that path are enqueued in the appropriate crosspoint FIFO, and later forwarded according to the preferred path. Alternatively, the preferred path could remain intact, but flits in that path are still enqueued as above. Therefore, as long as a flit travels in a preferred path, it is not affected by contention or congestion. However, since preferred path flits take precedence over flits in FIFOs, congestion is slower to resolve.

4 Switch Architecture

Switch input port components and connections are shown in Figure 4(b). Output port components are illustrated in Figure 4(a). Data wires are illustrated as solid lines and control wires as dashed lines. Our switch is a composition of the above figures, featuring 6 input/output ports. Each input is connected to each other port's output. Our switch resembles a buffered crossbar [8] in that it features one FIFO at each crosspoint and independent configuration and arbitration logic at each output. A combinational routing logic block at each input decides at which FIFO, if any, should the incoming flit be enqueued.

This choice of switch architecture takes into account mad-postman's operation, since incoming flits are examined by the routing combinational logic before being able to be enqueued into FIFOs. Therefore, dead flits do not occupy FIFO lines. Moreover, since our current NoC does not include virtual channels, as addressed in subsection 4.1, crosspoint queueing removes the nuisance of head-of-line-blocking. Finally, the use of one arbitration and configuration logic block per output results in simpler logic and therefore shorter critical paths.

Output configuration logic is responsible for storing and updating preferred path configuration. Arbitration logic is responsible for serving the FIFOs. Non-empty FIFOs as well as FIFOs to which a flit is being enqueued are selectable. Arbitration logic starts serving FIFOs once there is a selectable FIFO and the preferred path has been idle for one clock cycle. This serves the purpose of prioritizing preferred path flits without unreasonably preventing FIFOs from being served. It stops serving them when they are all empty. Arbitration takes place during the preferred path idle cycle, for the next cycle. Therefore, our NoC achieves one clock cycle per-hop latency for non-preferred paths when there is no contention. Arbitration algorithm details may depend on exact NoC demands.

Each output is driven by tri-states directly connected via dedicated wires to each other port's input. Each output is also driven by a tri-state which connects the output wire with a multiplexer which forwards the FIFO flit be-

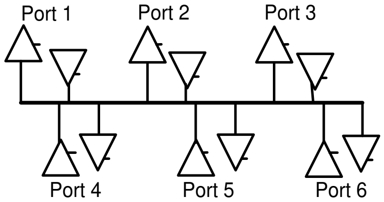


Figure 5. Preferred path bus.

ing served by arbitration logic, if any. Tri-state enable signals are driven by the output's configuration and arbitration logic. Preferred paths are thus formed by pre-enabling tri-states, therefore connecting an input with any number of preferred outputs.

Depending on preferred path flexibility and area needs, an extra optimization may be necessary to further reduce switch area. Instead of directly connecting each input to each other output, a preferred path bus could be deployed, as in Figure 5. This vastly limits the number of preferred input-output pairs that can be configured to only one input with any number of outputs. However, intermediate designs can also be implemented. For instance, one such preferred bus in the X axis and one in the Y could be deployed, perhaps even connected to each other with tri-states. Therefore, depending on exact preferred path communication needs, NoC area overhead can be reduced.

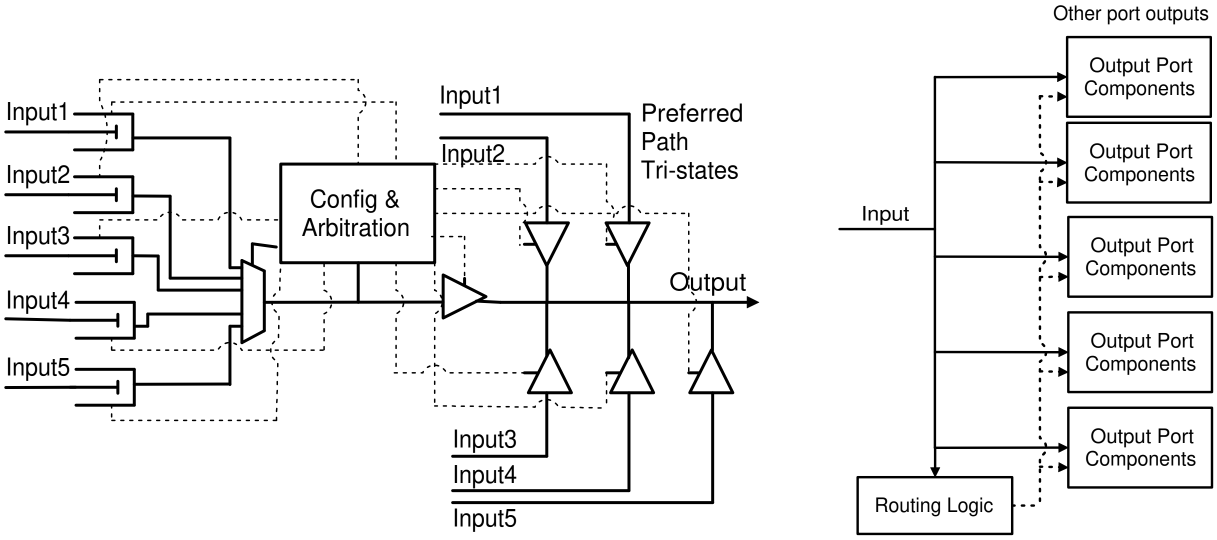
4.1 Virtual Channels

Virtual channels (VCs) are useful for defining multiple logical topologies within the network, adaptively routing around congested or faulty nodes and providing packet priority and thus guaranteed QoS classes [10]. However, in our NoC preferred paths already provide a means to prioritize packets compared to others as well as form different low-latency topologies. Moreover, our NoC's topology is already tailored to our specific application environment. Finally, our NoC faces challenges in implementing adaptive routing algorithms, as explained in subsection 3.4.

For these reasons, our current NoC does not include VCs. Introducing them would multiply FIFOs, which translates into a significant area overhead since our switch features one FIFO at every crosspoint. However, other NoC applications may have different design priorities and requirements which make VCs more attractive.

5 Network Topology

We tailor a 2D mesh topology to our target application, which is an array of processors and RAM blocks, aiming to minimize area overhead in addition to latency. This topology is illustrated in Figure 6. We assume a flexible system that assigns memory blocks to processors according to



(a) Output port components (b) Input port components

Figure 4. Switch architecture.

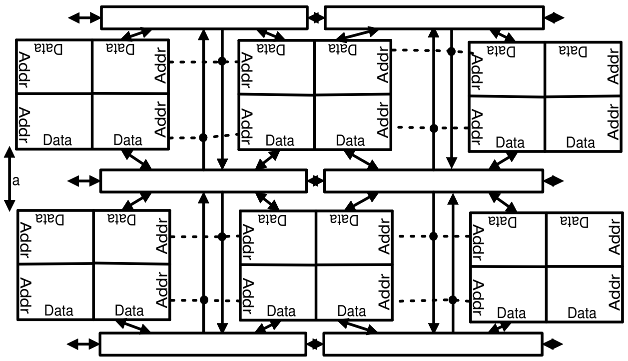


Figure 6. Rectangular-shaped floorplan.

application needs, and therefore profits from the reconfiguration capabilities of our NoC. We used single-port RAM blocks. In our 130 nm implementation library, RAM blocks feature data pins on one side of the X axis and address pins on one side of the Y axis. We therefore place four RAM blocks to form one larger network block. We rotate and mirror RAM blocks to place all data pins on the X axis and all address pins on the Y axis. CPUs and other user blocks may be placed as part of such a block or as a whole network block themselves, depending on their size.

In our current NoC each switch has 6 input/output ports. Each input is connected to each other port's output. Two of these input/output ports are used for inter-switch communication in the X axis, and other two in the Y axis. The rest two ports are used for communication with the data inputs of the 4 adjacent RAM or other user logic blocks. Given the data pin placement, one port is used for each two RAM blocks facing each other in the X axis. One data output is

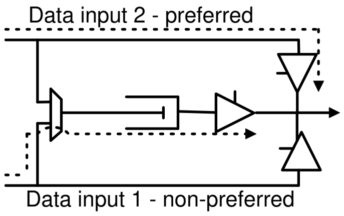


Figure 7. 2x1 switching logic.

wired to both RAM block data input interfaces. The two RAM blocks' data outputs are connected to a simple logic illustrated in Figure 7, resembling a 2x1 switch. It has no routing logic and only features one FIFO which is used by the non-preferred input. From the moment a RAM block receives a read request from the address interface until it is able to output data, it notifies this 2x1 switch to choose that RAM block's data output as preferred. This switch therefore imposes minimal latency impact. It may also be reconfigured as other switches. If area permits, a request FIFO may be implemented to store the order of requests received by the RAM blocks. This will enable it to always anticipate the next generated flit, and thus avoid non-preferred path delays in case of multiple requests to both RAM blocks.

RAM block address interfaces are wired to the nearest Y axis output. These connections are illustrated with dashed lines. Thus, the RAM blocks immediately above a switch have their address interfaces wired to the Y output leading upwards. This way, we avoid implementing extra output ports for address inputs. As data input interfaces, address interfaces monitor each incoming flit to determine if it is destined for that RAM block. The potential increased contention for outputs wired to address interfaces is outweighed

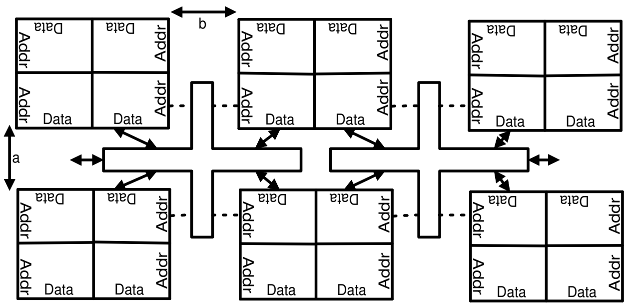


Figure 8. Cross-shaped floorplan.

by the significantly less area required by our switch. Finally, for switches that serve exclusively RAM blocks, we can further reduce the required switch area since RAM blocks will never need to communicate to each other directly. Therefore, each switch data input should not be connected to the other switch data output, therefore saving two internal switch connections and all the accompanying logic.

For switch placement, we examined two floorplan alternatives evaluated in section 6. In the first, switches are placed in the corners of larger network blocks as a cross, shown in Figure 8. This requires only a small distance between user blocks in each axis. Moreover, wire length, and therefore propagation delay, between each switch is minimal, even in the Y axis. The second placement, shown in Figure 6, has the switch solely in the X axis between two large blocks in a rectangular shape. User block distance in the Y axis is truly minimal, and is only used for memory address interface logic. Communication with switches in the Y axis is achieved by wires in higher metal layers routed above RAM blocks, or possibly in any metal layer routed above address interface logic. Y axis communication wire length is equal to twice the RAM block's height, approximating to 1 mm in our placement and routing.

5.1 Network Interfaces

PEs, RAM blocks or other user logic blocks need NoC interface logic. This logic is responsible for enabling communication between the NoC and the block. It is responsible for submitting properly formated packets divided in flits, as explained in subsection 3.2, as well as receiving flits destined to the user block. For data network interfaces, incoming flits must be briefly stored until the whole packet is complete and thus able to be submitted to the user block for processing. Address network interfaces only receive one flit per packet containing the address. In addition, network interface logic must be able to arbitrate between complete packets in a desired manner, *e.g.* submit read and write requests in the order they were transmitted by the source to satisfy sequential consistency.

Network interface logic must also identify which flits of a packet it has already received and discard duplicate copies. There are various implementations of this functionality according to design optimization priorities. To simplify this task, the flit control bits could be expanded, or the packet ID bits lessened, to include sequence number bits.

Out-of-order delivery of flits belonging to the same packet is impossible, as explained in subsection 3.7. However, flits belonging to different packets from the same source may be delivered in any order. Therefore, interface logic must submit packets for processing once complete, regardless of other incomplete packets. To implement this functionality, we deploy multiple small FIFOs in data interfaces and registers in address interfaces, and enqueue flits accordingly. In case we would like packets to be submitted in the order they were sent by the source, the relative packet order can be retrieved from the packet header.

Network interface logic must also handle multiple incoming packets from various sources. Flits from these packets may arrive in any order. If all buffer is used up, extra incoming flits are stored in previous hops through backpressure. Therefore, no excessive buffer space is required.

Since one data interface FIFO is reserved per packet until it is complete and submitted for processing, and packets may arrive out-of-order, deadlocks may occur. Lets assume that packet A has partially arrived at the target RAM. Packet B from the same source arrives at the final hop before A's tail flit. However, all of the RAM's data interface FIFOs are reserved. Therefore, packet B waits in the switches due to backpressure, not allowing packet A's tail flit to arrive.

This scenario requires several packets arriving out-of-order through the same path with partially complete packets, since the data interface logic deploys several FIFOs. Assuming sources do not submit packets interleaved, out-of-order delivery is only caused by reconfiguration. Therefore, limiting the number of active reconfigurations that can occur at any one time to less than the number of data interface FIFOs guarantees that at least one FIFO will be eventually freed and no deadlock will occur. Implementation of this restriction may require software synchronization primitives, defining areas each CPU can reconfigure paths in, or a reconfiguration acknowledgment mechanism.

6 Layout Results

We performed placement and routing using a 130 nm library available to European universities. Synthesis was conducted with Synopsys Design Compiler version 2004.06-SP2, placement and routing with Cadence SOC-Encounter version 3.3 and simulation with Verilog-XL version 05.10.002-p. We chose single-port RAM blocks of 4096 lines of 32 bits each (128kbits), with a column mux of 16. Without power rings, they are 715.07 μm long on the X axis, and 551.64 μm long on the Y axis. Larger RAM blocks had a disproportionally larger cycle time, while multi-port

Table 1. Switch p&r results (typical).

Impl. lib.	130nm		
P. supply	1.2V		
Clock freq.	667 typ. - 400 w.c. (MHz)		
I/O ports	6		
FIFOs	30		
FIFO lines	2		
Flit width	39 bits		
	Full switch	Pref. bus	Change
Gates	44874	38865	-13%
Cells	15001	13369	-11%
Cell area	195228μm^2	183056μm^2	-6%
Int. nets	13595	12703	-6.5%
Comb. area	84424μm^2	72420μm^2	-14%
Non-comb.	110798μm^2	110632μm^2	-0.1%
Leakage p.	91μW	85μW	-7%
Dynamic p.	80mW	77mW	-3%

RAMs were larger and more power consuming.

Switch p&r details are shown in Table 1. Results presented are under typical case conditions. Power consumption results are under heavy switching activity. Preferred path latency per switch ranged from 300 to 420 ps. If we also include a 1 mm long wire at the output, approximately twice a RAM block's height, latency increases to 450-550 ps, as compared to 135 ps for straight buffered wires of a similar length without any configuration or routing capability. When there is no contention, non-preferred path latency is one clock cycle. Contention without starvation effects increases non-preferred path latency depending on various factors, but does not affect preferred path latency. Our design functions at 667 MHz under our library's typical case conditions, and at 400 MHz under worst case conditions.

At 667 MHz, RAM blocks require a 25 μm wide power ring. Therefore, RAM block effective size is 740.07 μm $\times$ 576.64 μm. In an orthogonal shape, one switch occupies a minimum area of 637 μm $\times$ 310 μm. In the rectangular placement option as explained in section 5 and illustrated in Figure 6, switch height (a) is 170 μm at minimum. Since we require one switch every 4 RAM blocks (or user blocks of roughly the same size), NoC area overhead is 13%. In the cross placement option as depicted in Figure 8, switch height in the X axis (a) is 130 μm, while switch length in the Y axis (b) is 140 μm. In this case, NoC area overhead is 18%. This shows that area efficiency drops in the second case. However, cross-shaped switches have the least possible distance between each other even in the Y axis, therefore minimizing propagation delay between them.

P&r details of the area-efficient single-preferred path switch explained in section 4 are shown in Table 1. In the rectangular placement option, switch height (a) is 133 μm (22% decrease). In the cross placement option, switch height in the X axis (a) is 118 μm (9% decrease), while switch length in the Y axis (b) is 114 μm (18.5% decrease). This imposes a NoC area overhead of 10% in the first case and 16% in the second. These results show that the area gain is small, but in some applications it could outweigh the loss in preferred path flexibility.

7 Future Work

A number of issues should be addressed in the future. Firstly, while our current NoC utilizes a deterministic routing algorithm as explained in subsection 3.4, adaptive routing has significant benefits to offer. For instance, congestion can be avoided by later flits. Therefore, a customized version of an adaptive routing algorithm should be investigated to provide our NoC with more flexibility.

Secondly, our NoC needs to be made fault-tolerant since faults may appear in a chip's lifetime, especially in technologies narrower than 130 nm. This will impose an unavoidably increased area overhead. However, technology trends dictate that designs must be fault-tolerant in some way in order to be trusted for future designs.

Thirdly, our NoC needs to be evaluated in a complete system under various workloads and demands. This will enable us to accurately analyze our contribution's impact, as well as the effect of dead flits in our NoC's performance. Moreover, we can investigate the optimal method for choosing preferred paths in a given application environment, as well as the impact of this choice.

Finally, our NoC can face synchronization issues which may result in design limitations. Since preferred paths are purely combinational, flits traversing them can arrive at their destinations and other switches at any point during the clock cycle. Thus, flits may violate flip-flop setup or hold time upon arrival.

There are several approaches to combat this issue. Firstly, we could impose a constraint on our preferred paths so that this problem will never occur. For example, we could limit the number of continuous preferred path hops such that flits will enter a non-preferred path before the end of the clock cycle that they entered their current preferred path in. For example, assuming that flits are submitted from non-preferred paths in the very beginning of the clock cycle, that number is $\lceil \frac{PrefPathHop+WireLatency}{ClockPeriod} \rceil$.

Furthermore, we could deploy synchronizers at every switch and PE interface logic. While they will not affect preferred paths until flits exit them, their imposed latency in non-preferred paths is excessive. Finally, our switch components can easily be implemented asynchronously with known design methodologies. The problem in this approach lies in the necessary handshake between switches and PEs to guarantee that no flit fragments will be routed through the network. This handshake's imposed delay will prevent us from offering our current low per-hop latency, both in preferred and non-preferred paths.

8 Conclusion

We presented a NoC design that offers low latency in pre-configured paths. This latency is close to that of long buffered wires. To achieve our goal, we have resurrected and tailored mad-postman, a technique proposed two decades ago. According to our implementation, an incoming flit is eagerly forwarded to the input's preferred outputs, if any. This is achieved solely by pre-enabled tri-state drivers, and therefore with the least delay possible. Flits are then checked by routing logic to determine if they were correctly eagerly forwarded. If not, flits are forwarded to the correct output. Incorrectly forwarded flits are terminated in later hops as dead. When there is no contention, non-preferred paths impose a single clock cycle per-hop delay.

For routing, we implement XY routing. However, we make the modification that a flit is considered to have been correctly forwarded if it approaches the destination in any of the two axes, even if it does not follow strict XY routing. This way, flits may take different paths and gain from increased preferred path flexibility. A flit is considered dead if its distance from the destination increases in any of the two axes. Path reconfiguration occurs at run-time. Any user block can transmit configuration packets to any switch in the NoC. Switch architecture resembles that of a buffered crossbar [8]. FIFOs are placed at crosspoints, and each output port has independent arbitration and configuration logic.

P&r results show that preferred path latency varies from 300 ps to 550 ps, depending on placement and wire length. Our NoC imposes a 13% area overhead for the whole chip.

We believe that our work provides a different approach in some areas and can form the basis for future NoC implementations which focus on low latency. While there are open issues left for future work, a substantial number of past NoC research can be applied and therefore provide solutions. Depending on exact application needs, further latency, area or energy optimizations may be made.

Acknowledgments

We wish to thank our colleagues, both locally and throughout HiPEAC and SARC, for their assistance: Christos Sotiriou, Spyros Lyberis, Pavlos Mattheakis, Stamatis Kavvadias, Vasilis Papaefstathiou, Michalis Papamichail, Kees Goossens, Giuseppe Desoli, Krisztian Flautner, Chris Jesshope, Jose Duato, and Georgi Gaydadjiev. This work was supported by the European Commission in the context of the SARC (Scalable Computer Architecture) integrated project #27648 (FP6), and the HiPEAC network of excellence.

References

[1] E. Beigne, F. Clermidy, P. Vivet, A. Clouard, and M. Renaudin. An asynchronous noc architecture providing low latency service and its multi-level design framework. In *ASYNC '05: Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 54–63, Washington, DC, USA, 2005. IEEE Computer Society.

[2] R. B. C. Izu and C. Jesshope. Mad-postman: a look-ahead message propagation method for static bidimensional meshes. In *Proceedings of the 2nd Euromicro Workshop on Parallel and Distributed Processing*, pages 117–124. IEEE Computer Society Press, 1994.

[3] W. J. Dally and B. Towles. Route packets, not wires: On-chip interconnection networks. In *Proceedings of the 38th Design Automation Conference (DAC)*, pages 684–689, Las Vegas, NV, June 2001.

[4] J. Hu and R. Marculescu. Dyad: smart routing for networks-on-chip. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 260–263, New York, NY, USA, 2004. ACM Press.

[5] C. R. Jesshope, P. R. Miller, and J. T. Yantchev. High performance communications in processor networks. In *ISCA '89: Proceedings of the 16th annual international symposium on Computer architecture*, pages 150–157, New York, NY, USA, 1989. ACM Press.

[6] J. Kim, C. Nicopoulos, and D. Park. A gracefully degrading and energy-efficient modular router architecture for on-chip networks. *SIGARCH Comput. Archit. News*, 34(2):4–15, 2006.

[7] J. Kim, D. Park, T. Theocharides, N. Vijaykrishnan, and C. R. Das. A low latency router supporting adaptivity for on-chip interconnects. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 559–564, New York, NY, USA, 2005. ACM Press.

[8] D. S. M. Katevenis, G. Passas et al. Variable packet size buffered crossbar (cicq) switches. In *Proceedings of the IEEE International Conference on Communications (ICC)*, pages 1090–1096, Paris, France, June 2004.

[9] R. Mullins, A. West, and S. Moore. Low-latency virtual-channel routers for on-chip networks. *SIGARCH Comput. Archit. News*, 32(2):188, 2004.

[10] L. M. Ni and P. K. McKinley. A survey of wormhole routing techniques in direct networks. pages 492–506, 2000.

[11] L.-S. Peh and W. J. Dally. Flit-reservation flow control. In *In Proc. of the 6th Int. Symp. on High-Performance Computer Architecture (HPCA)*, pages 73–84, Jan. 2000.

[12] L. Shang, L.-S. Peh, A. Kumar, and N. K. Jha. Thermal modeling, characterization and management of on-chip networks. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 67–78, Washington, DC, USA, 2004. IEEE Computer Society.