Many-Core Real-Time Network-on-Chip I/O Systems for Reducing Contention and Enhancing Predictability

Zhe Jiang
University of Cambridge
Cambridge, United Kingdom

Xiaotian Dai
University of York
York, United Kingdom

Shuai Zhao
Sun Yat-sen University
Guangzhou, China

Ran Wei
Dalian University of Technology
Dalian, China

Ian Gray
University of York
York, United Kingdom

ABSTRACT

In safety-critical and high-integrity computing, it is important to guarantee both performance and time-predictability of I/O operations. However, with the continued growth of hardware and architectural complexity, satisfying such real-time requirements has become increasingly challenging because of complex I/O transaction paths and extensive hardware contention. This paper proposes a systematic framework with a novel I/O controller and a reconfigurable NoC, effectively optimising I/O transaction paths to encounter reduced contention. Moreover, we present a theoretical model and optimisation process to further improve real-time performance. The evaluations show that our approach outperforms state-of-the-art I/O processing techniques on a range of metrics.

ACM Reference Format:
Zhe Jiang, Xiaotian Dai, Shuai Zhao, Ran Wei, and Ian Gray. 2023. Many-Core Real-Time Network-on-Chip I/O Systems for Reducing Contention and Enhancing Predictability. In *Cyber-Physical Systems and Internet of Things Week 2023 (CPS-IoT Week Workshops '23), May 09–12, 2023, San Antonio, TX, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3576914.3587514

1 INTRODUCTION

In modern safety-critical systems, an increasing number of *Input/Output* (*I/O*) devices are being integrated into System-on-Chips (SoCs), driven by the diverse functionalities required by embedded computing [11]. For instance, in an autonomous control system, decision-making modules rely on a large amount of I/O inputs (*e.g.*, camera images) to understand the surrounding environment, and pilot/navigation modules generate a series of I/O outputs (*e.g.*, motor control) to perform a manoeuvre to avoid dangerous scenarios [13].

To ensure the correctness and timeliness of these safety-critical modules, it is vital to assure the time-predictability and performance of the associated I/O operations [13]. Considering the above example, significant timing uncertainty or unintended performance degradation during an I/O process may violate the function of these modules, leading to disastrous consequences [5, 11].

CPS-IoT Week Workshops '23, May 09–12, 2023, San Antonio, TX, USA

ACM ISBN 979-8-4007-0049-1/23/05.
https://doi.org/10.1145/3576914.3587514

Research challenges. I/O timing and performance requirements were usually *implicit* in conventional safety-critical systems, as the systems had relatively less complexity and fewer I/Os; hence, real-time I/O operations were triggered on interrupts of a high-resolution timer, *e.g.*, a microsecond timer in a RTOS [5].

As the number of hardware elements has grown, modern safety-critical systems usually involve *complex I/O transaction paths* and *extensive hardware contention*, leading to challenges in guaranteeing performance and response times of I/O operations. Specifically, to access an I/O device in a Network-on-Chip (NoC) based many-core system, I/O requests issued by a processor must pass through the OS kernel, I/O manager, and software drivers. At the hardware level, the I/O request is then required to be transmitted between multiple routers/arbiters and an I/O controller. After processing, corresponding results (*i.e.*, I/O responses) are sent back to the processor using a similar routine. Such complex paths bring significant communication latency and timing uncertainty to I/O operations. Moreover, along the transaction paths, hardware contentions occur extensively and intensively. For instance, contentions can happen on all routers deployed on an I/O transaction path. Such hardware contentions elevate the difficulty of satisfying I/O timing and performance requirements.

Related work. Existing efforts aimed at achieving real-time I/Os in multi/many-core systems often focus on a particular system level.

At the software level, Kim *et al.* [15] and Dong *et al.* [7] modified OS kernels to improve the predictability of I/O scheduling; Kim *et al.* [14] and Abdallah *et al.* [3] presented I/O contention-aware task mapping and scheduling to reduce I/O contentions between software tasks. However, it is tough to ensure real-time performance of I/O operations from a given software level, as I/O operations mainly rely on complex interactions with the underlying hardware. In addition, the software-based methods usually bring extra computational overhead and complexity, leading to a further reduction of I/O throughput [15].

At the hardware interconnect level, there are a range of approaches that impose predictability on NoC transactions, *e.g.*, Burns *et al.* [4] and Plumbridge *et al.* [16] proposed approaches for predictable on-chip communication flows. These papers concentrate entirely on NoC transactions; such work is important in improving I/O predictability, but does not solve predictability on its own.

At the hardware I/O level, industrial vendors and researchers have developed programmable I/O controllers for real-time application scenarios, *e.g.*, a Programmable Real-Time Unit (PRU) and a Time Processor Unit (TPU) developed by TI [1] and NXP [2], as well

as a GPIO command processor (GPIOCP) and a Real-Time I/O controller (RT-IOC) presented by Jiang *et al.* [12] and Zhao *et al.* [17]. This work presented dedicated co-processors for I/O processing, handling I/O requests directly at the hardware level, improving I/O throughput, and reducing communication latency. However, as with the other work, these controllers only focus on one phase of I/O operations (I/O requests) for a single I/O device. Without considering the full procedure of an I/O operation - the I/O response paths are ignored - it cannot ensure I/O real-time performance.

A complete solution must capture the entire request and return path of an I/O operation, from the application model, through the software level and the on-chip interconnect, to the I/O device. Therefore, we present a range of novel approaches to address this problem in totality. In the following sections, we present our methods.

2 NPRC-I/O: ARCHITECTURE

In this section, we give an overview of NPRC-I/O.

2.1 Hardware Level

At the hardware level, system elements are connected using R^{2}NoC (Fig. 1(b), and detailed in Sec. 4). To minimise hardware modifications, we follow a conventional architecture to mount processors and memory on the routers' local home ports. However, we replace traditional I/O controllers by NPRC-CCs to handle I/O operations (detailed in Sec. 3). In addition, we connect the NPRC-CCs and free routers (*i.e.*, a router without a local client) using a dedicated I/O crossbar (*i.e.*, I/O-Ring) to support run-time reconfiguration of connections between the routers and NPRC-CCs.

2.2 Software Level

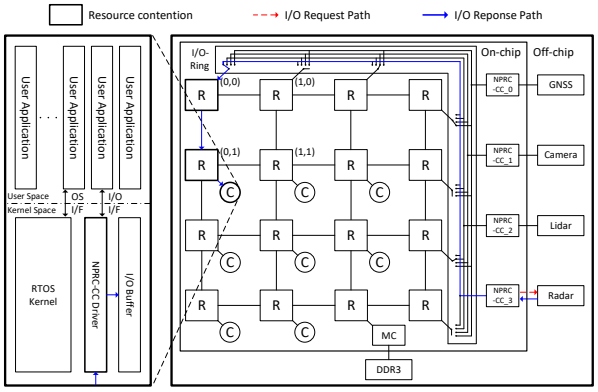
The software-level structure is made of kernel and user spaces.

Kernel space. In the kernel space, we deploy an RTOS with full privileges to provide a real-time environment for user applications that require timing guarantees. In this work, we use FreeRTOS, but the specific choice of RTOS is not important. With NPRC-CCs, I/Os are managed in the hardware; hence, we replace the software I/O manager and low-level I/O drivers (managing I/Os in the conventional systems) with new NPRC-CC drivers. Contrasting with the conventional architecture, user applications in NPRC-I/O communicate with I/O devices using our NPRC-CC drivers (Fig. 1(a)), without the involvement of the OS kernel. The implementation of the new drivers is straightforward as all complexity is handled in the hardware design of NPRC-CCs. The drivers simply forward I/O requests between user applications and NPRC-CCs. This new structure gets as much work out of the software as possible, simplifying I/O transaction paths and associated contentions in software, and decreasing overhead.

User space. Although NPRC-I/O introduces a new software structure, the system maintains the original I/O-application interfaces presented by traditional systems (Fig. 1(a)), ensuring *source compatibility* of the existing user applications – user tasks designed for a conventional system can be directly migrated to NPRC-I/O.

2.3 Working Procedures

In general, I/O requests in a system are issued either *periodically* or *sporadically*. Periodic requests are usually determined before system execution, *e.g.*, periodic sensor sampling, and sporadic requests are usually generated during system execution, *e.g.*, sporadic



(a) Software (b) Hardware Platform

Figure 1: System Architecture of NPRC-I/O. Radar and its I/O controller (NPRC-CC_3) are configured to connect with the router (0,0).

brake control. Moreover, since I/O requests inherit the properties of the software tasks issuing them, the timing demands of the I/O requests can be different, overall classified as *hard* real-time (HRT) and *soft* real-time (SRT). Precisely, an HRT I/O task needs a restricted response time bound for its execution, whereas an SRT I/O task's timing bound is relatively less restrictive - the I/O tasks are allowed to over-execute its soft deadline occasionally. Based on this understanding, the NPRC-I/O's working procedures are introduced in four phases:

Phase 1: NPRC-CC initialisation. Before run-time, NPRC-I/O initialises NPRC-CCs by sending configuration packets. The initialisation has two steps (i) pre-loading I/O requests, *i.e.*, a series of I/O operations, to NPRC-CC's local memory unit; (ii) reserving time budget for periodic and HRT sporadic I/O requests. The time budget is allocated by specifying the requests' starting time points and the worst-case ending time points.

Phase 2: R^{2}NoC Reconfiguration. An optimisation algorithm is executed with the timing information of the I/O responses, assisting NPRC-I/O to reconfigure the connections between NPRC-CCs and routers, by finding parameters that lead to minimal contentions.

Phase 3: Request process. At system run-time, NPRC-CCs execute all pre-loaded periodic I/O requests at the specified time points, guaranteeing their predictability and performance. NPRC-CCs also receive and buffer the sporadic I/O requests issued by the processors. NPRC-CCs execute HRT sporadic requests using the time budget reserved in Phase 1, ensuring HRT requests can always be served with an analytical timing bound. When an I/O device is not occupied, the connected NPRC-CC schedules and proceeds (both HRT and SRT) sporadic requests based on their priorities.

Phase 4: Response process. After an I/O request is processed, the I/O device transmits the corresponding result back to the software level through the NPRC-CC, I/O-Ring, and routers. R^{2}NoC provides a dedicated response path to ensure minimal contention and analysability.

As described in this section, ensuring system-wide I/O predictability and performance relies on the novel hardware. In the following sections, we detail the design of NPRC-CC and R^{2}NoC.

3 NPRC-CC DESIGN

To handle a broad range of I/O types, NPRC-CC is full-duplex, providing independent paths for I/O requests and responses.

The response path is pass-through, because the processing speed of the requesters (*e.g.*, processors) is often hundreds of times faster than I/O devices. The request path has four parts (Fig. 2): a periodic requests space (P-space), a sporadic requests space (S-space), scheduling circuits, and an I/O controller. The P-space stores the periodic I/O requests determined by hard real-time system specifications. The S-space buffers and prioritises sporadic I/O requests generated at run-time. The scheduling circuits connect the two spaces, selecting I/O requests to operate on the I/O device using the I/O controller.

3.1 Periodic Requests Space (P-space)

The design of the P-space consists of a memory module (including memory banks and a controller), a fetcher, and shadow buffers. The memory module stores the periodic I/O requests. If the fetcher receives a scheduling decision from the scheduling circuits, it then collects the specified requests from the memory module and decomposes them to corresponding I/O operations. After that, the fetcher maps these I/O operations into shadow buffers.

3.2 Sporadic Requests Space (S-space)

The S-space design contains two I/O pools, a loader, and shadow buffers. The I/O pools are deployed to maintain sporadic I/O requests with similar timing demands (*i.e.*, HRT or SRT). An I/O pool has a priority queue, an arbiter and a fetcher. Unlike conventional FIFO queues, the priority queue adopts a more complicated micro-architecture to enable *random access* of its stored contents. It uses a register chain and register banks to store I/O requests and their associated parameters (detailed in Sec. 5). The register chain is connected to a loader and a fetcher; the register banks are connected to the arbiter. At execution, the loader pushes I/O requests received from the processors; the arbiter continuously checks the requests' parameters, finding the request with the highest priority and controlling the fetcher to map the request to shadow buffers.

3.3 Scheduling Circuits

The scheduling circuits consist of a memory module, a scheduler, and a multiplexer. The memory module stores the timing information of the periodic and HRT sporadic I/O requests (Phase 1 in Sec. 2.3). Timing information is stored in a look-up table (called the *time slot table*) to record the run-time behaviours of these requests in each *hyper-period*. At run-time, the scheduler synchronises with a global timer and compares the synchronised results with the time slot table. Once the system runs at the starting time point of a periodic or HRT sporadic I/O request, the scheduler controls the multiplexer to load the I/O request from the corresponding shadow buffers (in S/P-space) to the I/O controller. At the same time, the scheduler removes the I/O request from the shadow buffers. If there is no I/O request specified at a given time point (*i.e.*, the time slot is free), the scheduler loads the sporadic I/O requests with the highest priority. Note, to ensure timing correctness, any loaded sporadic request must be able to complete execution before the starting time point of the next request.

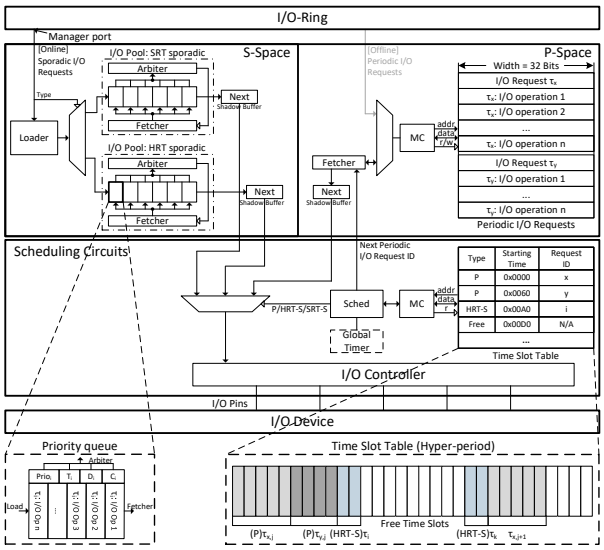


Figure 2: Micro-architecture of NPRC-CC.

3.4 I/O Controller

The NPRC-CC design is agnostic to the underlying I/O controllers; hence either standardised Intellectual Property (IP) cores or customised real-time controllers can be applied. The selection of I/O controllers only depends on the communication protocol required by the connected I/O device, *e.g.*, I^2C, SPI. In addition, we either removed the FIFO queues or minimised their depth in the selected I/O controller, because (i) I/O requests are maintained in the S-space and P-space; (ii) deploying FIFO queues forbids request prioritisation.

4 R^{2}NOC DESIGN

Although deploying NPRC-CCs effectively bounds the real-time performance of I/O requests, NPRC-CC cannot avoid the complex transactions paths and hardware contentions associated with sporadic I/O requests and I/O responses, as they are determined at run-time. In coping with these I/O transactions, we introduce R^{2}NoC to support run-time reconfiguration of connections between I/O controllers and routers. With R^{2}NoC, an optimisation method (see Sec. 5) can be applied, minimising hardware contentions associated with sporadic I/O requests and I/O responses. The R^{2}NoC design contains an open-source real-time NoC mesh [16] and an I/O crossbar (*i.e.*, I/O-Ring):

4.1 NoC Mesh

The design of the NoC mesh constructs routers in the style of a Manhattan grid, where the routers are addressed by their horizontal (X) and vertical (Y) coordinates (Fig. 1(b)). Each router has five 32-bit bi-directional ports, connected to the other routers located at its north, south, east and west, as well as a local client (using its home port). The NoC mesh encapsulates on-chip transactions as packets using a protocol developed in [16], and wormhole-routes the packets through each router towards their corresponding destinations. The routers' design is based on priority queues, transmitting the packets upon their priorities. As evidenced in [16], the NoC mesh can bound a packet's worst-case transmission time between any two routers.

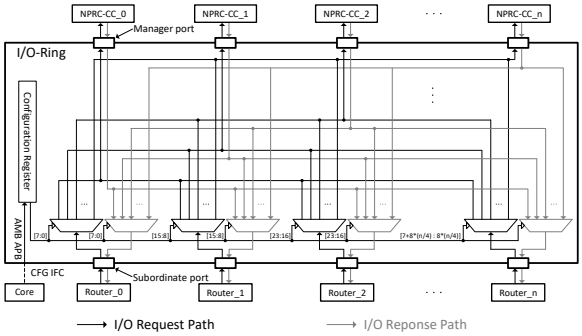


Figure 3: Micro-architecture of I/O-Ring in R^{2}NoC.

4.2 I/O-Ring

I/O-Ring is a dedicated crossbar which is placed between routers and NPRC-CCs. The idea behind the I/O-Ring is to establish a contention-free and one-to-one link between any router and NPRC-CC mounted on the I/O-Ring. However, doing this arbitrarily would be enormously expensive, so I/O-Ring is instead run-time reconfigurable to allow it to establish its connections based on the required timing properties and behaviour of the system.

The I/O-Ring design consists of interface ports, multiplexers and configuration registers (see Fig. 3). At the interfaces, the I/O-Ring presents groups of Subordinate ports and Manager ports, physically connected to the routers' home ports and NPRC-CCs. Inside the I/O-Ring, a Subordinate port is fully connected to all Manager ports with a multiplexer, used to select an active transaction path between the Subordinate and Manager ports. The path selections of the multiplexers are stored in the configuration registers (32-bit), where each multiplexer consumes 8 bitfields. We connect the configuration registers to an AMBA APB interface and map it to dedicated memory addresses. This allows the processor to access these configuration registers directly using memory read/write operations. Since the connections between the Subordinate and Manager ports are implemented using pure combinational logic, a transaction packet can always be transmitted within a fixed single clock cycle, giving the software the illusion that the NPRC-CC is directly mounted to the router's home port. For instance, in Fig. 1, the NPRC-CC_3 is configured to be "connected" to the router (0,0).

5 LATENCY ANALYSIS AND OPTIMISATION

This section provides an analytic bound of the worst-case end-to-end latency of given I/O flows and an optimisation process to further improve NPRC-I/O's real-time performance. We assume a closed system in which we know the maximum transmission time, the periods of all periodic I/Os, and the minimum inter-release time for sporadic I/Os. We also assume deadlines are given. For this work, the transmission rate is identical on all interconnected routers. This is a reasonable assumption for the currently-deployed NoC. With R^{2}NoC, traffic packets are scheduled based on their priorities in a local router, using a priority queue, *i.e.*, no extra blocking due to queuing of lower priority traffic. At most one I/O device can be connected (using the reconfiguration of the I/O ring) to each NoC router. The routing path is allocated *statically* by *shortest path first*.

5.1 System Model

We use a meshed NoC (R^{2}NoC) that has a dimension of $X \times Y$, with processor cores $P = \{p_0, p_1, ..., p_{m-1}\}$, $||P|| = m$ and I/O controllers $K = \{k_0, k_1, ..., k_{n-1}\}$, $||K|| = n$. Each I/O device has a dedicated I/O controller. The router on NoC location (x, y) is denoted $R_{(x,y)}$. An I/O traffic is defined as $(id_i, T_i, C_i, D_i, prio_i, route_i = \{src_i, ..., dst_i\})$ where id_i is used to identify the I/O flow; T_i is the period or minimal inter-release time; C_i is the maximum transmission time between two directly connected routers; D_i is deadline, with $D_i \leq T_i$; $prio_i$ is the traffic priority, and src_i and dst_i are the source router (where the I/O response comes out) and the destination router (where the I/O response comes in), respectively. As the request route is already reduced to its minimum due to the implementation of NPRC-CC, we focus on response routes in this analysis. Thus, a source refers to one of the I/Os and a destination refers to one of the processors.

5.2 Worst-case Latency Analysis

The worst-case end-to-end latency of I/O traffic (defined as the end-to-end delay from the time point at which the response is sent by the I/O to when it is received by the requesting processor), WCL_i, can be obtained using the following equation:

$$WCL_i = \sum_j WCL_{i,j} = \sum_j L_{i,j} + \sum_j B_{i,j} \tag{1}$$

where

$$\begin{cases} L_{i,j} = \sum\limits_{j;k \in hp(i,j)} l_{i,j,k} + C_i = \sum\limits_{j;k \in hp(i,j)} \left\lceil \frac{l_{i,j}}{T_k} \right\rceil \times C_k + C_i \\ B_{i,j} = \max\limits_{j;k \in lp(i,j)} (C_k) \end{cases} \tag{2}$$

$L_{i,j}$ is the latency term and $B_{i,j}$ is the blocking term at router j; $l_{i,j,k}$ is the latency at router j, caused by waiting for higher priority traffic k to be transmitted; The blocking time, $B_{i,j}$, as in a non-preemptive priority scheduled I/O system, is upper bounded by the maximal transmission time of any lower priority traffic k that goes through this router on its transmission path. As we are using priority queues, the blocking only occurs at most once.

5.3 Network Optimisation

To reduce the latency of the end-to-end I/O response times, routes can be optimised by: (i) placing I/O nodes, and (ii) assigning I/O traffic with appropriate priorities that will produce the lowest contention with other traffic on the NoC.

The optimisation problem is constructed as follows:

- **Objective**: Minimise the overall latency, *i.e.*, $\sum WCL_i$.
- **Subject to**: The deadline of all HRT I/O traffic must be met, *i.e.*, $WCL_i \leq D_i, i \in HRT$.
- **Targeted variables**: Allocation of I/O devices and the priorities of I/O traffic flows. An I/O node can be assigned to any permitted and unallocated NoC router.

The optimisation algorithm. In this paper, the optimisation is performed by a metaheuristic genetic algorithm (GA). GAs are used throughout the optimisation of real-time systems [6]. As only one objective exists (to minimise contention), we use a single-objective GA. We note that other methods, for example, Integer Linear Programming (ILP), can also be used following similar optimisation procedures to those given in this section.

Inputs. The inputs to the search algorithm include: (i) parameters of periodic and sporadic traffic; (ii) number of processor cores and I/O flows; and (iii) traffic routes.
Outputs. The outputs contain: (i) I/O node locations on the NoC - encoded as pairs (x, y); (ii) priorities of I/O flows; and (iii) the lowest worst-case latency of the accumulated I/O requests.
Gene encoding and fitness function. To enable the use of a genetic algorithm, the gene (genotype) is encoded as ($location$, $priority$) pairs, *i.e.*, [$location_1$, $priority_1$, $location_2$, $priority_2$, $\cdots$, $location_n$, $priority_n$], where the index is the traffic ID and n is the amount of total I/O traffic. The fitness function is the reverse of the accumulated sum of latency (fitness = $1/\sum WCL_i$), in which case a higher summed latency would suggest a lower fitness.
Algorithm complexity. The time complexity of the GA is in the order of $O(gnm)$, with g being the number of generations, n the population size and m the size of the individuals (*i.e.*, gene number). The complexity of exhaustive search (even with priorities given) is $O(P(k, h))$, where $P(\cdot)$ is permutation, k is the number of I/O devices to be allocated and h is the candidate locations. As can be seen from time complexity, the time it takes to find a solution using exhaustive search is dramatically large.

6 EVALUATION

Experimental platform. We built NPRC-I/O on an FPGA development board, the Xilinx VC709. NPRC-CCs and R^{2}NoC were implemented using BlueSpec System Verilog, and R^{2}NoC was configured to use 60 routers, formed as a 10×6 grid. As well as the NPRC-CCs and R^{2}NoC, the system also contained 32 MicroBlaze processors, memory and I/O devices. We deployed the processors to the home ports of the routers at the central grid and connected the routers and I/Os using I/O-Ring. We used FreeRTOS (v.10.4) as the OS kernel for all processors, with the modifications introduced in Sec.2.2. The software executing on the processors was compiled using the Xilinx MicroBlaze GNU toolchain. We also introduced four baseline systems (**BS**s) running on similar hardware architecture. BS|Legacy was a conventional NoC-based real-time system without any additional support, which left the I/O management entirely to the RTOS and routers (reviewed in Sec. 1). BS|SW was designed based on BS|Legacy with additional software patches [7], enabling I/O contention-aware scheduling at kernel level. BS|HW:GPIOCP and BS|HW:RTIOC were two real-time systems with dedicated hardware I/O co-processors (GPIOCP and RTIOC) presented in [12] and [17]. All systems ran at 100 MHz.

6.1 Theoretical Evaluation

Experimental setup. The experiment was based on simulations with theoretical evaluation. The implementation of the search algorithm was based on the *PyGAD* (Python Genetic Algorithm) package. I/O traffic was randomly generated with the size of the NoC and the number of processors and I/Os being fixed (32 and 28, respectively). The total I/O utilisation was set from 50%-100% to add workload to the NoC gradually. The population size was set to double the number of genes, and the maximum iteration number was set to 100 to upper-bound the search time. Our optimisation method is compared with (i) a heuristic that allocates the largest node (i.e. with the highest utilisation) first to the location closest to its destination; and (ii) a random search that executes a hundred times more than the GA and takes the best solution. Each utilisation was evaluated with 50 trials.

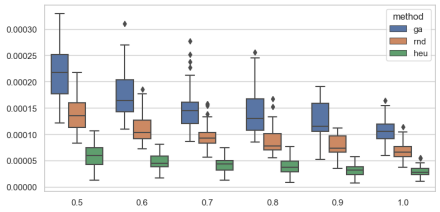


Figure 4: Search-based optimisation for minimised latency — worst-case latency w.r.t. total utilisation (x-axis: utilisation; y-axis: fitness).

Obs 1. The proposed search algorithm was able to find performant solutions with respect to overall latency with lower complexity.

This observation is based on Fig. 4. In the figure, the proposed genetic algorithm (*ga*) outperformed both random (*rnd*) and heuristic (*heu*). With the utilisation increases, *ga* can still find better solutions. Although the heuristic has slightly lower time complexity, its performance was much lower than both *ga* and *rnd*.

6.2 Hardware Overhead

Experimental setup. We configured an NPRC-CC to buffer 100 I/O operations and configured the I/O-Ring to support 16 NPRC-CCs. We first compared the NPRC-CC's overhead with standardised I/O controllers (*i.e.*, SPI, CAN, and I^{2}C) and other real-time I/O controllers. We then examined the I/O-Ring's hardware overhead along with the NoC mesh and a general-purpose AXI interconnect (AXI-IC). The standardised I/O controllers and AXI-IC were chosen from the Xilinx IP library, and the AXI-IC was configured to support 64 connections (like the NoC mesh). All components were synthesised and implemented by Vivado (v2021.1).
Obs 2. The design of NPRC-CC and R^{2}NoC was resource-efficient. NPRC-CC consumed a similar amount of hardware as other real-time I/O controllers; I/O-Ring increased the NoC mesh's overhead.

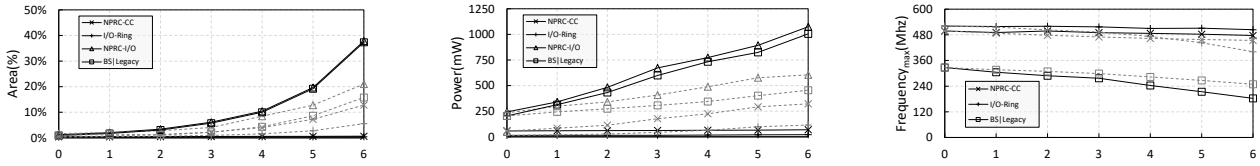
As shown in Table 1, NPRC-CC consumed more resources than the standardised I/O controllers: SPI (155.0% LUTs, 226.2% registers), CAN (141.0% LUTs, 172.4% registers), and I^{2}C (149.71% LUTs, 165.12%), but the costs are still very reasonable for real-world implementation. The additional overhead comes from the hardware-level implementation of I/O scheduling and management. When compared to other real-time I/O controllers, NPRC-CC required similar hardware resources: GPIOCP (110.8% LUTs, 144.6% registers, 100% RAMs), and RT-IOC (92.2% LUTs, 115.0 % registers, 50% RAMs).

For R^{2}NoC, deploying I/O-Ring brought negligible extra overhead: 14.9% LUTs and 2.8% registers. The introduced overhead was significantly less than a general-purpose interconnect, *i.e.*, AXI-IC.

6.3 Hardware Scalability

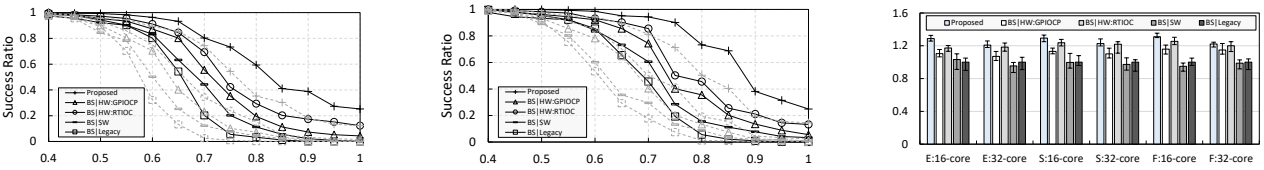
Experimental setup. We adopted the same method described in Sec. 6.2 to implement NPRC-I/O with scaling numbers of processors and I/Os. Additionally, we introduced two scaling factors: η^{core} and η^{io} to control the number of processors and I/Os (2^{η}).

We first compared the scalability of area consumption between NPRC-CC, I/O-Ring, NPRC-I/O, and BS|Legacy. The area consumption was normalised by the overall area of the experimental platform. We then examined the scalability of power consumption,



(a) Area consumption. (b) Power consumption. (c) Maximum frequency.

Figure 5: Area, power, and maximum frequency v.s. scaling factor η^{io} and η^{core} (x-axis: η^{core} for black solid lines, η^{io} for grey shading dash lines).



(a) 25% SRT tasks (x-axis: target utilisation). (b) 50% SRT tasks (x-axis: target utilisation). (c) Normalised average I/O throughput.

Figure 6: Case study: system-level real-time performance (in Fig. 6(a) and 6(b), black solid lines: 16-core systems; grey shading dash lines: 32-core systems. In Fig. 6(c), E: Ethernet; S: Q-SPI; F:Flexray; error bars: experimental variances).

Table 1: Hardware overhead (implemented on FPGA)

	LUTs	Registers	RAMs (KB)
SPI	674	519	0
CAN	741	681	0
ETH	1,595	943	0
GPIOCP	943	677	16
RT-IOC	1,134	1,021	32
NPRC-CC	1,045	1,174	16
NoC Mesh	18,452	15,238	0
AXI-IC	9,682	8,735	0
I/O-Ring	2,754	432	0

calculated as the sum of static and dynamic power simulated by the tool. Lastly, we evaluated the maximum frequency of NPRC-CC and I/O-Ring across the BS|Legacy using varying η^{io} and η^{core}.

Obs 3. The area and power consumption of NPRC-CC and R^{2}NoC were not affected by η^{core} and linearly scaled by η^{io}.

As seen in Fig. 5(a), when systems were scaled with η^{core}, the area consumption of both NPRC-CC and I/O-Ring were nearly constant. When systems were scaled with η^{io}, the area consumption of both NPRC-CC and I/O-Ring were linearly increased. In these cases, although NPRC-I/O required more hardware than BS|Legacy, the introduced area consumption was always bounded within 30%.

Power consumption is usually determined by four factors: voltage, clock frequency, toggle rate and design area [10]. Because the unified voltage, clock frequency and simulated toggle rate were assigned to the items being compared, the design area dominated overall power consumption. As expected, in Fig. 5(b), we observe nearly constant and linearly increased power consumption of NPRC-CC and I/O-Ring when the systems were scaled with η^{core} and η^{io}.

Obs 4. When the system was scaled with η^{core}/η^{io}, deploying NPRC-CC and I/O-Ring did not decrease maximum performance.

This observation is shown in Fig. 5(c). In all examined cases, the NPRC-CCs' maximum frequency was always around 490 Mhz, which is significantly higher than BS|Legacy. This is because the NPRC-CCs were instantiated individually. When the system was scaled with η^{io}, I/O-Ring's maximum frequency slightly decreased, but it was still greater than BS|Legacy. This means that I/O-Ring did not become a critical path.

6.4 Real-time Performance: Case Study

We now evaluate the systems using real-world use cases.

Task sets. We presented two sets of tasks: (i) 18 automotive safety tasks, selected from the Renesas automotive use case database [9], *e.g.*, CRC, RSA32, *etc.* (ii) 18 automotive function tasks, chosen from the EEMBC benchmark [8], *e.g.*, FFT, speed calculation, *etc.*

We employed a hybrid-measurement approach to obtain WCETs for all tasks. The raw data processed by the tasks was randomly generated off-chip and collected by the tasks via an Ethernet controller (1 Gpbs) at run-time. The calculated results of the safety tasks were sent back to a Quad-SPI flash (40 Mbps), and the calculated results of function tasks were sent back to a FlexRay receiver (10 Mbps). Each task had a randomly defined period (defined before each experiment), with overall processor utilisation approximately 40%. Additionally, we also collected tasks from the EEMBC benchmark as synthetic workloads, which could be added to the system to control overall system utilisation. Like the safety tasks, the calculated results of synthetic workloads were also sent back to the Quad-SPI flash. Note that, in practical systems the execution time of a task is affected by diverse factors (e.g., cache miss rate); hence, adding synthetic workloads to a system only gives it a *target utilisation.*

Experimental Setup. We introduced two groups of system configurations, which activated 16/32 processors to execute the task sets and synthetic workloads. We also presented two experimental setups for each system configuration, randomly assigned 25%/50% tasks as SRT tasks and the others as HRT tasks. The HRT task deadlines were equal to their periods (*i.e.*, implicit deadlines). In each experimental group, we executed the examined systems 1,000 times under varying target utilisations, from 40% to 100% (with an interval of 5%). Each execution lasted 250 seconds. For a fair comparison, we ensured the data input to the examined systems was identical in each execution.

We examined the systems using success ratio and I/O throughput. The success ratio recorded the percentage of trials that were executed successfully. For successful execution, a HRT task should not over-execute its deadline, whereas a SRT task should not exceed 120% of its period [5]. The throughput is normalised by BS|Legacy.

Obs 6. NPRC-I/O improved the system-level real-time performance. This observation is given by Fig. 6, In most of the examined cases, NPRC-I/O outperformed the BSs, in terms of success ratio, I/O throughput and experimental variances. However, with an increasing number of processors, the improvement of I/O throughput brought by NPRC-I/O decreased slightly. This is because introducing more processors also increased the number of I/O requests, pushing the I/O devices' throughput in all systems close to their physical limits.

Obs 7. Raising the distribution of SRT tasks increased the real-time performance of NPRC-I/O, when utilisation was relatively low.

This observation is given by comparing the experimental results in Figs. 6(a) and 6(b). When the system utilisation was lower than 85%, NPRC-I/O's success ratio significantly increased with the increased distribution of SRT tasks. But, when the utilisation exceeded 85%, NPRC-I/O achieved similar success ratios with different distributions of SRT tasks, indicating system utilisation dominated the real-time performance when the system was overloaded.

7 CONCLUSION

This paper presents a new systematic framework (*i.e.*, NPRC-I/O) for multi-/many-core real-time I/O processing. NPRC-I/O introduces a novel I/O controller (NPRC-CC) and a run-time reconfigurable NoC (R^2NoC), optimising I/O transaction paths with reduced contentions. Moreover, we present a theoretical model and optimisation to further improve NPRC-I/O's real-time performance. As shown in theoretical and practical evaluations, NPRC-I/O outperforms the state-of-the-art I/O processing techniques with varying hardware configurations. The reason that this work demonstrates both predictability and high performance is that it presents a configurable hardware-based solution to predictable I/O. The cost of this approach is that each new task set requires the system configuration to be updated by allocating time budget and locations in the I/O controllers. This requires offline analysis, and also benefits from a search-based optimisation stage. This paper demonstrates that this can be done tractably.

REFERENCES

[1] 2023. PRU. http://www.ti.com/tool/pru-swpkg.
[2] 2023. TPU. http://www.nxp.com/products/microcontrollers-and-processors.
[3] Laure Abdallah, Mathieu Jan, et al. 2016. I/O contention aware mapping of multi-criticalities real-time applications over many-core architectures. (2016).
[4] Alan Burns, LS Indrusiak, N Smirnov, and J Harrison. 2020. A Novel Flow Control Mechanism to Avoid Multi-Point Progressive Blocking in Hard Real-Time Priority-Preemptive NoCs. In *Proc. RTAS*.
[5] Alan Burns and Andrew J Wellings. 2001. *Real-time systems and programming languages: Ada 95, real-time Java, and real-time POSIX*.
[6] Xiaotian Dai, Wanli Chang, et al. 2019. A dual-mode strategy for performance-maximisation and resource-efficient CPS design. *ACM TECS* (2019).
[7] Pan Dong, , et al. 2021. Exploring Real-time Hybrid-Criticality System on ARM TrustZone Technology. *Journal of Systems Architecture* (2021).
[8] EEMBC. 2023. EEMBC benchmark. https://www.eembc.org/autobench/.
[9] Renesas Electronics. 2023. Renesas: Automotive Use Cases. https://www.renesas.com/solutions/automotive.html.
[10] John L Hennessy. 2011. *Computer architecture: a quantitative approach*.
[11] ISO. 2018. 26262: Road vehicles-Functional safety. *FDIS* (2018).
[12] Zhe Jiang and Neil C Audsley. 2017. GPIOCP: Timing-accurate general purpose I/O controller for many-core real-time systems. In *DATE*. IEEE.
[13] Zhe Jiang, Neil C Audsley, and Pan Dong. 2018. Bluevisor: A scalable real-time hardware hypervisor for many-core embedded systems. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 75–84.
[14] Jung-Eun Kim, Man-Ki Yoon, Richard Bradford, and Lui Sha. 2014. Integrated modular avionics (IMA) partition scheduling with conflict-free I/O for multicore avionics systems. In *Proc. COMPSAC*.
[15] Namhoon Kim, Stephen Tang, Nathan Otterness, James H Anderson, F Donelson Smith, and Donald E Porter. 2018. Supporting I/O and IPC via fine-grained OS isolation for mixed-criticality real-time tasks. In *RTNS*.
[16] Gary Plumbridge. 2014. Blueshell: a platform for rapid prototyping of multiprocessor NoCs and accelerators. *Computer Architecture News* (2014).
[17] Shuai Zhao et al. 2020. Timing-accurate general-purpose I/O for multi-/many-core systems: scheduling and hardware support. In *Proc. DAC*.