SDN-Based Circuit-Switching for Many-Cores

Marcelo Ruaro, Henrique Martins Medina, Fernando Gehm Moraes
PUCRS – FACIN – Av. Ipiranga 6681 – Porto Alegre – 90619-900 – Brazil
{marcelo.ruaro, henrique.medina}@acad.pucrs.br, fernando.moraes@pucrs.br

***Abstract*—Software-Defined Networking (SDN) is a paradigm adopted in computer networks that decouple the control logic from the physical to the software layer, reducing the routers' complexity and enabling a high-level network awareness. This work explores the SDN paradigm on NoCs to manage Circuit Switching (CS) connections in an original Multi-Physical Network (MPN) architecture. Related works focus its CS proposals only at the NoC context. This work proposes a CS design that covers all applications' lifetime stages, through a systemic many-core CS support that addresses both the hardware and software layers. The proposal explores the MPN's path diversity to provide a permanent CS during the real-time applications' lifetime. Results show that: (*i*) the MPN design provides a rich path diversity; (*ii*) the search path algorithm reaches success rates superior to 97% for 6 and 8 CS subnets; (*iii*) the SDN paradigm enables to adopt simple hardware infrastructures without software bottlenecks.**

Keywords-Circuit-Switching; NoC; Software-Defined Network; Many-Cores.

I. INTRODUCTION

Self-awareness becomes fundamental in large-scale many-core systems due to the high complexity of physical (e.g. temperature, power) and logic (e.g. high density of applications, resource sharing) events happening at runtime. Systems need to be aware of the resources' utilization to implement smart self-adaptive mechanisms for sustaining real-time (RT) constraints [1]. Circuit switching (CS) is a technique widely adopted to guarantee RT constraints at the communication level, being employed since 2004 with MANGO [2] and AEthereal [3] NoCs. CS guarantees throughput because it provides communication isolation by removing spatial or temporal resource sharing among flows. CS requires an initial phase called *connection* (or setup), which searches and allocates the path's resources.

The *connection* phase is the subject of several research works, with the goal to reduce the connection latency. As explored in Section II, proposals address CS at the NoC level, with the goal to design CS infrastructures with hardware-centered connection algorithms and constant connection time. However, these works lack in a systemic and self-aware CS support for applications, i.e. aiding the CS provisioning from the applications' beginning up to its ending, and addressing the technical challenges to accomplish this support. This work fulfills this gap by presenting a self-aware CS design guaranteeing RT support at the communication level. This work covers the details from the physical CS architecture up to the high-level CS management. The RT applications start their execution with all task exchanging data by dedicated CS paths, established during the application admission. This approach brings two main benefits: (*i*) removes the overhead to execute several connections and releases during the application lifetime, e.g. for each decoded frame [4]; (*ii*) provides communication predictability during the application lifetime.

This work proposes a CS controller implemented in software. This *CS-Controller* defines the paths by searching paths and configuring CS routers at runtime. The *CS-Controller* handles requests from system managers to establish connections during the application admission and to release connections when the application finishes its execution. This software-based control enables to reduce the hardware complexity, moving to the software the decision to establish the network paths. This approach is similar to a Software-Defined Networking (SDN) paradigm [5][6], an approach used in computer networks to reduce routers' complexity and to flexibilize the link allocation, providing a high-level management of the network. The goal of this work is to explore the SDN paradigm on NoCs, using it to define CS connections. Besides low hardware complexity, SDN inherently provides communication awareness, a fundamental attribute in self-adaptive many-core systems [1].

Figure 1(a) overviews a classical SDN organization [5]. In a typical NoC design, routers have the control logic implemented in hardware. Assuming an SDN approach, the control logic is decoupled from the physical layer and implemented by a software network controller. Decouple the control layer from the physical layer simplifies the router's design, making it configurable to be used according to different software policies. This infrastructure is transparent to the application layer, not requiring any modification by the application developer.

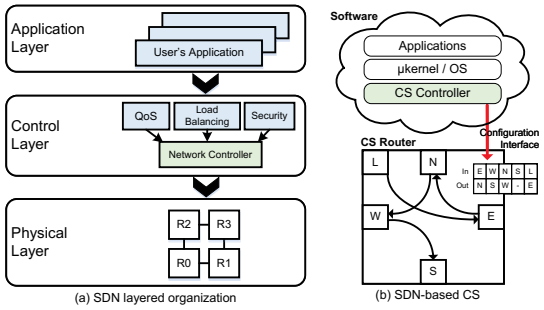

Figure 1. SDN organization, and its adoption for NoC design.

Figure 1(b) overviews the proposed SDN-based CS. The µkernel (OS) asks to the *CS-Controller* to define CS paths when an RT application is requested to execute into the system. CS routers, configured by the *CS-Controller*, act as a forwarding unit, linking an input port to an output port. The *CS-Controller* configure several CS routers to make a CS path. After the connection, the µkernel inject/receive the applications' data by CS.

The CS routers employ Elastic-Buffers (EB) [7][8] to reduce the silicon area and power consumption further. This work adopts a Multi-Physical Network (MPN) architecture to improve the path diversity, with one packet switching (PS) subnet, and a parameterizable number of CS subnets (defined at design-time). A many-core system, modeled at the RTL level, enables to evaluate the proposal with a clock cycle accuracy. We address system sizes with dozens to hundreds of Processing Elements (PEs) and RT applications benchmarks (described in C language). The *contribution*s of this work are a configurable EB-based CS router (Section IV), and an SDN-based infrastructure to manage the CS connections (main contribution, detailed in Section V).

This work is organized as follows. Section II presents related works. Section III details the assumptions related the many-core architecture. Section IV details the CS router architecture. Section V presents the SDN-based CS management. Section VI presents and discusses the results. Section VII concludes the paper and draws directions for future works.

2159-3477/17 $31.00 © 2017 IEEE
DOI 10.1109/ISVLSI.2017.74

II. RELATED WORK

Table I presents related works on CS NoCs with proposals addressing the search path method. The 2nd column details the hardware infrastructure to support CS. Time Division Multiplexing (TDM) and Spatial Division Multiplexing (SDM) flexibilize the CS resource utilization at the cost of complex designs [4][9]. Recent works expose the benefits on using Multi-Physical Network (MPNs). Works [10][11] compare MPN to SDM designs. With an equivalent bandwidth implementation, MPN increases the area following an *O(n)* complexity, while SDM increases $O(n^2)$. The path delay increases with the number of sub-channels (SDM) with a complexity *O(n)*, while remains unchangeable for MPN. Authors in [9] evaluate virtual channels (TDM) with MPN, concluding that MPNs have a simpler implementation, lower area, and simpler floorplan. The current work adopts MPNs due to its *design simplicity and rich path diversity*.

TABLE I. RELATED WORKS ON CS NOCS, WITH SEARCH PATH PROPOSALS.

Works	CS	Path Length	Implementation	Organization	Target
Stefan [12]	TDM	Minimal	SW	Centralized	NoC
Chen [13]	TDM	Shortest	HW	Centralized	NoC
Leroy [4]	SDM	Deterministic	SW/HW	Centralized	NoC
Liu [14][11]	TDM/MPN	Shortest	HW	Distributed	NoC
Lusala [15]	SDM	Deterministic	HW	Distributed	NoC
This work	*MPN*	*Shortest*	*SDN-based*	*Centralized*	*System*

The CS support for systems with dynamic workload requires a runtime path search mechanism. The 3rd column of Table I presents the resulting path length of the proposed approaches. Minimal path length corresponds to the Manhattan distance between the source and target PEs. Deterministic search uses a deterministic routing algorithm, as XY. The shortest path searches for all possible paths in the NoC, returning the minimal path between the source and target PEs. Works [4][12][15] addresses only minimal or deterministic search methods. Such approaches limit the path exploration, thus reducing the success rate to establish the CS connections. As works [11][13][14], the present work adopts the search for shortest paths. This approach increases the success rate to establish CS connections because the search space increases.

Works that search for the shortest path use dedicated hardware implementations (4th column), with centralized or distributed approaches (5th column). Chen et al. [13] propose a centralized design, implementing a dedicated hardware unit to establish the connection. Liu et al. [14] implement a distributed approach using a flood-based search, eliminating redundant paths, and achieving a constant setup time per hop. That work was extended to MPN and SDN in [11]. Our work proposes a centralized path search method, implemented in software, based on the grid routing algorithm proposed by Hadlock [16]. The option to choose the Hadlock's algorithm is due to its simplicity, low memory footprint, and better performance compared to other grid routing methods [17].

Hardware-based search path designs are faster than software approaches, at the cost of increased silicon complexity. Software-based solutions are scarce because the search path evaluation has as target the NoC context (6th column) without a systemic CS support (i.e., including the software stack in the evaluation). This work adopts a software-based search path algorithm, proposing a self-aware systemic CS management based on the SDN paradigm. To the best of the Authors knowledge, this is the first work to introduce the SDN-based paradigm for NoCs, and use it to manage CS.

III. SYSTEM MODEL

Figure 2(a) overviews the many-core and the MPN architecture. The system contains a set of PEs, organized in clusters. Each cluster has a Manager PE (M_{PE}), used for management purposes as the application admission control. Slave PEs (S_{PE}), execute user's tasks, with support for multitasking. An application repository (off-chip memory), stores the *applications' descriptors* and the *object code* of the tasks. The repository also triggers new application requests, handled by one M_{PE}. All PEs have an NI (Network Interface). The approach is not limited to heterogeneous or homogeneous systems. The current work adopts a homogenous architecture, with all PEs having a dual-port scratchpad memory and a CPU, as detailed in Figure 2(a). The NI is attached to the MPN. It serializes/deserializes data to/from the subnets. The MPN architecture contains one PS router (wormhole switching mode), and a set of CS routers. There is no data exchanged between routers.

Figure 2(b) presents an application example. An application A is modeled as a task graph: $A = \{T, D\}$, where $T = \{t_1, t_2, ..., t_n\}$ is the task set. The set D represents the *application descriptor*, which contains the communicating task pairs (*CTP*), and a tag (*is_real_time*) indicating when A is RT. The *CTPs* of the application presented in Figure 2(b) are: $t_A \rightarrow t_B$, $t_A \rightarrow t_C$, $t_B \rightarrow t_D$, $t_C \rightarrow t_D$. Tasks use MPI-like *Send* and *Receive* primitives to communicate. *Receive* blocks the consumer task execution, and fires a request to the producer task, which handles the request and delivery the data, unblocking the consumer task.

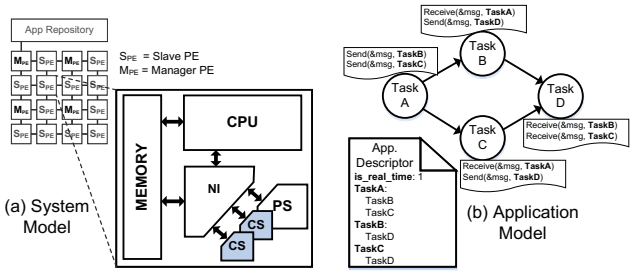

Figure 2 - Overview of the many core system, detailing the PE architecture and the application model.

IV. ELASTIC BUFFER-BASED CS ROUTER

This Section details the first *contribution* of this work, the CS router. The router has five input and five output ports (East, West, North South, Local). Each *input port* has one Elastic-Buffers (EB). Figure 3(a) details the EB architecture [7][8]. An EB contains a master and a slave latch, controlled by the EB control logic. The master latch is enabled when the clock signal is low, and the slave latch when the clock signal is high. This latch-based design allows the latches to be used as two independent storage locations. An EB uses a ready-valid handshake protocol. The *ready* signal notifies the upstream router that the current EB can receive a flit in its master latch. The *valid* signal notifies the downstream router that current EB has a flit to transmit. A flit is transferred to the downstream EB when both the *ready* and *valid* are asserted at the rising clock edge.

Figure 3(b) details the CS router architecture. Besides the EB, each port has an upstream *req* signal employed by the consumer PE to request data from the producer PE. The *req* is used for the inter-task communication, being asserted when the consumer task requests a message to the producer task.

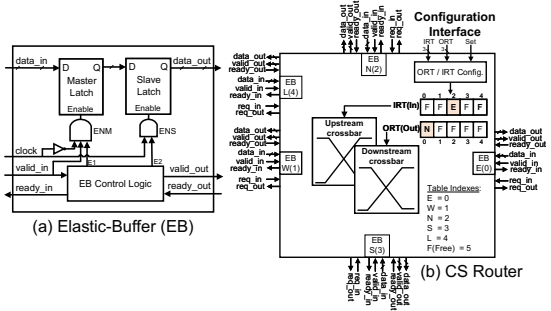

Figure 3 - (a) EB architecture [7], (b) CS Router architecture, with 5 EB.

The CS router has internally two crossbars, to connect the upstream (*ready*, *req*) and the downstream (*valid*, *data*) signals between the input and output ports. They are configured by the Input Reservation Table (IRT) and Output Reservation Table (ORT), respectively. Each table is a 5-entry array (number of input ports) with 3 bits at each slot (enabling to store six states: E, W, N, S, L, Free). In Figure 3(b), the North inport is forwarding data to the East outport. A *configuration interface* enables to program the IRT and ORT tables. This interface is the key feature to make the CS router simple, avoiding logic for routing and arbiter modules. The configuration information arrives at the CS routers by the PS subnet, through a configuration packet (Section V.A.1 details this process). With such mechanism, the CPU (or any other IP) is not involved in the setup of the CS router.

The adoption of EB fits with the CS behavior once data is transmitted in streaming. Thus, each CS router only buffers the data for 1 clock cycle, avoiding long wires, ensuring a reduced clock period. EBs also enable to reduce the silicon cost compared to a two slot FIFO [7][8], once EBs require only one flip-flop instead two.

V. SDN MANAGEMENT

This Section details the main contribution of this work, the SDN management for many-core systems. Despite the advantages of the SDN-based management, reduced hardware complexity and CS communication self-awareness, SDN imposes an important challenge: *the software overhead to manage NoC flows at runtime*. Figure 4 presents the proposed layered architecture to mitigate this issue. It assumes a *CS-controller* as a parallel unit at the middleware layer. This organization is generic and can be applied to different many-core platforms.

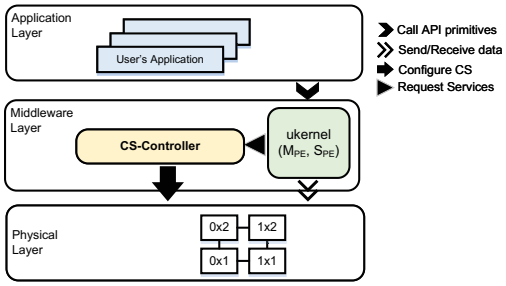

Figure 4. Proposed SDN layered organization for many-cores.

The µkernel (OS) makes the bridge between the user's applications and the NoC (physical layer). The user's application uses a communication API to send/receive data through the NoC. The main principles addressed with this organization are:

i. The *CS-controller* is decoupled from the baseline architecture. The system may work with the µkernel and the PS NoC, independently of the *CS-controller*.

ii. The *CS-controller* is a parallel system service of the µkernel.

Despite this work distribute the system management in the M_{PEs} µkernels, the *CS-Controller* is a single system component (centralized approach). A distributed implementation seems to be a natural choice because it distributes the search path computation load in several instances of the *CS-Controller*. However, we observed that the communication between the instances of the controller to ensure a global view of the CS connections becomes a bottleneck in the distributed approach. Moreover, due to low complexity of the SEARCH-PATH algorithm (Section V.A.1), its computation time, in the average case, grows linearly with the system size (Section VI).

A. CS-Controller

The *CS-Controller* executes two actions: (*i*) searches and configures CS paths; (*ii*) release CS paths. Equation 1 presents the memory storage requirements of the *CS-Controller* (in bytes).

$$S_{mem} = (R_n * CS_n * [p + 1]) + 3 * R_n \quad (1)$$

where: R_n is the number of routers, CS_n is the number of CS subnets, and p is the number of ports per router (5 - N, S, E, W, L_IN) plus one representing the input port of the NI (L_OUT). The first multiplication of the equation measures the amount of data required to store the status of the input port for each CS router, in a 3-dimensional array called *inport*. The *inport* stores *0* when a given port is free, or the $path_{ID}$, when the port is allocated. The $\{3*R_n\}$ multiplication corresponds to the size of inherited variables from the Hadlock's algorithm (propagation, neighbors, and detour).

1) SEARCH-PATH Algorithm

The SEARCH-PATH algorithm implements a heuristic that explores the path diversity of the MPN to select a CS *subnet* and a CS $path_{ID}$ from a source *S* to a target *T* PE.

The Hadlock's algorithm is a sub-function of the SEARCH-PATH algorithm. It searches for a path at each subnet. The Hadlock's algorithm is a grid routing search method, proposed in [16] and described in [17], originally used in VLSI synthesis. It is a bread-first algorithm that makes it expansion guided by a detour number *d*, assigned for each visited node. For minimal paths, *d(T)*=0. The detour number is increment according to the number of hops away from the minimal path. When a path *P* is found, the length of *P, len(P),* connecting *S* and *T* is defined according to equation 2.

$$len(P) = MS(S,T) + 2 * d(T) \quad (2)$$

where: $MS(S,T)$ is the Manhattan distance between *S* and *T*.

Figure 5 present the SEARCH-PATH algorithm. The algorithm initially searches minimal paths in all *subnets*. A *min_path* variable controls the Hadlock's algorithm search path mode. When *min_path* is TRUE, the Hadlock's search is limited to minimal paths (*d(T) = 0*), when *min_path* is FALSE, the Hadlock's search is released to find the shortest path (*d(T) > 0*), if it exists.

```
Algorithm: SEARCH-PATH (S, T)
1    input: S, T
2    output: pathID, subnet
3  begin
4     pathID = 0
5    min_path = TRUE
6    subnet = 0
7    while subnet < CSn do
8         if inport[S][subnet][L_IN]== 0 and inport[T][subnet][L_OUT]==0 then
9              pathID = HADLOCK(S, T, min_path)
10             if pathID > 0 then
11                break
12             end if
13        end if
14        subnet = subnet + 1
15        if subnet == CSn and min_path == TRUE then
16             subnet = 0
17             min_path = FALSE
18        end if
19   end while
20   return path_found, subnet
end
```

Figure 5. SEARCH-PATH algorithm.

Initially, the *min_path* is set to TRUE (line 5). The first round of the loop (lines 7-19) explores all CS subnets, stopping when a minimal path is found. Line 8 tests if the source (L_IN) and target (L_OUT) routers' interface with the NI are free or not. If TRUE, the Hadlock's algorithm is called (line 9), returning the $path_{ID}$ if a path was found, 0 otherwise. If all CS subnets were explored and no minimum path exists (line 15), the algorithm switches the search to non-minimal paths (*min_path=FALSE*), at lines 16-17. The search stops at the first path found. By employing this approach, the complexity of the SEARCH-PATH is $O(R_n^2)$ in the worst case and $O(R_n)$ in the best case.

When the Hadlock's algorithm reaches the router *T*, it executes the *retrace* phase to define the path's routers. This work use this phase to

store the $path_{ID}$ in the *inport* structure (software allocation), and to configure the CS routers (hardware allocation).

The CS router configuration corresponds to a small packet sent by the *CS-Controller* to a CS router to configure the IRT/ORT tables. Each configuration packet contains three flits (*header*, *payload_size*, *config*), sent through the PS subnet. The *header* contains the target address and a flag specifying that the packet must be consumed by a given CS router and not by the NI. The *config* flit has the following information: input port (3 bits), output port (3 bits), subnet number (CS_n bits). This packet is only sent during the allocation phase. It is not necessary to clear the IRT/ORT tables because the allocation is managed by software.

2) RELEASE-PATH Algorithm

The RELEASE-PATH algorithm uses the $path_{ID}$ stored in the *inport* array to release the CS connection. For each path, the algorithm removes the $path_{ID}$ value associated with the allocated input ports, setting it to 0 (free). The release phase is only executed in software. The RELEASE-PATH is a simple algorithm with complexity *O(len(P))*.

B. CS Management Protocol

Figure 6 presents the sequence diagram detailing the protocol for CS connection and CS release. New application requests to execute in the system at runtime. The *App. Repository* sends the *app descriptor* to an M_{PE}, which execute a task mapping heuristic to select the S_{PE} address for each application's task. Next, the M_{PE} requests to the *App. Repository* to load the code of the mapped tasks into its corresponding S_{PE}s (task allocation phase). In parallel to the load of the tasks' code, the M_{PE} requests CS connections to *CS-Controller*.

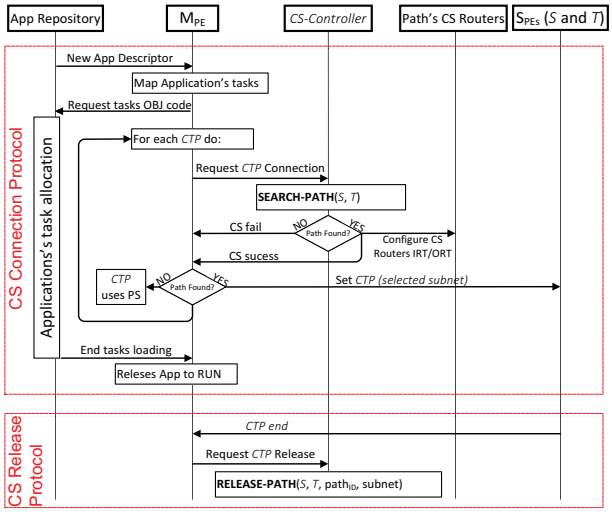

Figure 6. CS management protocol.

Best-effort applications not have timing bounds and use the PS subnet. RT applications (*is_real_time*=1) request CS for all *CTPs*. For each *CTP*, the M_{PE} requests to the *CS-Controller* to establish a CS connection between the source and target S_{PE}s. The *CS-Controller* executes the SEARCH-PATH algorithm and returns to the M_{PE} if the CS establishment failed (returning 0) or succeeded (returning the $path_{ID}$ and *subnet*). If succeeded, the M_{PE} sends messages to the *S* and *T* S_{PE}s, notifying the selected *subnet*. This message configures the *S* and *T* S_{PE}s to inject and receive data in the configured CS subnet. If failed, the *CTP's* communication uses the PS subnet. When both the *CTPs'* allocation finishes and the tasks are allocated into S_{PE}s (each task notifies the M_{PE} when it is ready), the M_{PE} releases the application sending a message to each task to start the execution.

When a task finishes, the S_{PE} notifies the M_{PE} the end of a given *CTP*. If the *CTP* communicates through CS, the M_{PE} uses the task identifier (unique for each running task), to recover the *subnet* and $path_{ID}$, and requests the *CS-Controller* to release the connection through the execution of the RELEASE-PATH algorithm.

The *CS-Controller* exposes to the system the CS services (connection and release) in a generic way because it only requires the source/target addresses to make connections. Thus, it also can handle requests from other IPs (rather than M_{PE}s), without a µkernel. Additionally, as the IPs are not involved in the process to configure the CS router, the CS connection can be established between generic IPs (memories, GPUs, I/O modules).

VI. Experimental Results

In this section, we evaluate the silicon area, power, performance, memory requirements, and latency of the proposed SDN-based CS. The many-core hardware is described in VHDL (for synthesis) and RTL SystemC (for performance evaluation of large-scale systems). Applications and µkernel are implemented in C language.

A. Area and Power of the CS Router

Both CS and PS routers were synthesized using the Cadence ASIC design flow. Table II compares the CS router with a low-area PS router, for different flit widths. The PS router is configured as follows: 8-flit buffer depth, round-robin arbitration, XY routing, no virtual channels. The CS router corresponds, in average, to 21% of the area and 16% of the power compared to the PS router. The area and power difference increase with the flit size. Such results show the low cost to adopt multiple CS subnets. For example, the adoption of 2 virtual channels (VCs) almost doubles the router silicon area of the PS router [18]. With the same silicon area of a 2 VC router, it is possible to use 1 PS and 4 CS routers.

TABLE II. GATE NUMBER (COMB. AND SEQ. GATES), AREA (µM^2) AND ESTIMATED POWER (µW) FOR THE CS AND PS ROUTERS (28 NM SOI TECHNOLOGY @ 1GHz).

Flit width	5-port **CS** router				5-port **PS** router			
	Comb. gates	Seq. gates	Area µm^2	Power µW	Comb. gates	Seq. gates	Area µm^2	Power µW
8	398	135	907	701	1079	472	3521	3683
16	517	215	1267	1154	1672	832	5948	6121
32	802	375	2011	1334	2927	1552	10921	10606

B. Performance Evaluation

The experimental setup addresses many-core sizes from 64 PEs (4 M_{PE}, 60 S_{PE}) to 400 PEs (16 M_{PE}, 384 S_{PE}), with three CS subnets configurations: 4, 6, 8. The PS routers use 32-bit flits, and the CS routers 16-bit flits. Results use the following definitions:

Path Diversity (P_D): number of available CS paths (equation 3).

$$P_D = \#PEs * \#CS\ subnets \quad (3)$$

Path Exploration (P_{EX}): percentage of CS requests w.r.t the *path diversity* (equation 4).

$$P_{EX} = \frac{\#\ CS\ requests}{P_D} \quad (4)$$

Success Rate: percentage of established CS connections per CS requests (equation 5).

$$success\ rate = \frac{\#\ CS\ connections}{\#\ CS\ requests} \quad (5)$$

The performance evaluation addresses several scenarios with different RT benchmarks: DTW (Dynamic Time Warping – a pattern recognition algorithm), JPEG decoder, MPEG-2 decoder, VOPD. Figure 7 presents the applications' communicating graph. We selected a heterogeneous set of applications aiming to create a fair evaluation. Each S_{PE} executes simultaneously two tasks. The evaluated scenarios execute several instances of the applications in such a way to have all S_{PE}s executing two tasks (*system occupation* equal to 100%). These

scenarios enable to stress the CS infrastructure. Real-life scenarios hardly run with a system occupation equal to 100%. Also, real-life scenarios mix best-effort and RT applications, using the PS network for best-effort applications.

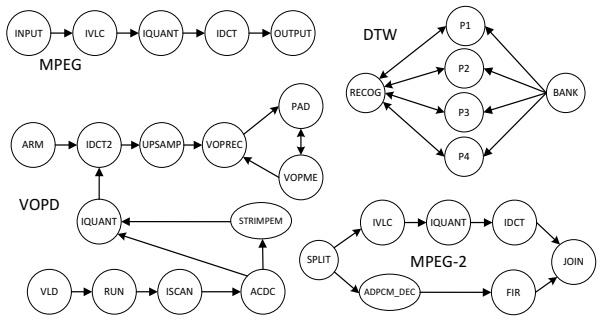

Figure 7. Communicating task graph of the applications benchmarks.

Table III details the results. The 1st and 2nd columns present the scenario's configuration. The 3rd column presents the number of tasks executing for each scenario. The 4th column presents the P_D for each scenario.

1) Success rate

The 5th column of Table III presents the number of CS requests and the achieved P_{EX}. The number of CS requests is a function of the applications' graph. Increasing the number of CS subnets decreases the P_{EX} due to a higher P_D. For example, the 20x20 scenarios generate 916 CS requests, resulting in a P_{EX}=57% for 4 subnets, and a P_{EX}=28% for 8 subnets. This result shows the rich path diversity provided by the MPN design. Even with a system occupation of 100%, the amount of CS requests is below the P_D. The 6th column presents the *success rate*. The *success rate* increases with the number of subnets, due to the higher P_D. The scenarios with 6 and 8 subnets reach a success rate superior to 97%. Figure 8(a) presents the *success rate* as a function of the P_{EX} (bottom x-axis) and CS request (top x-axis) for scenario 20x20:6. This scenario has a P_{EX}=38%, with a *success rate* of 98.25% (highlighted in the Figure 8(a)).

The mapping heuristic also contributes to the high success rate. This work adopts a task mapping heuristic which distributed the RT tasks in free PEs [19]. Other heuristics [20] may increase the proximity of the communicating tasks, contributing to increasing the success rate further.

The achieved success ratio can be compared to related works that address the shortest path search [11][13][14]. It is worth to mention that each proposal uses a different method to evaluate the success rate. Our success rate is computed according to the number of CS requests. In [14], the success rate is computed according to the *router rate*, a parameter that refers to the portion of clock cycles in which a node is used for transferring data. In [13] the success rate is computed according to the requested BW. Work [11] does not present the success rate. Additionally, works [13][14] addresses TDM while ours use MPN.

In [14] it is presented the success rate for a 16x16 mesh with a *master percentage* of 20% and 50%. The *master percentage* is similar to ours P_{EX}. Our success rate for P_{EX} =50% is 92% (20x20:4, worst scenario), while [14] achieves 30% of success rate with a master percentage=50%. That work also evaluates the HAGAR solution with the same configuration, which reaches a success rate near to 28%.

In [13] the authors vary the requested BW to evaluate the success rate. The Authors also considers a background traffic (*bk*) as the percentage of used TDM slots. The *bk* can be roughly compared to ours P_{EX}. Assuming 20% of *bk*, the success rate achieved for a requested BW of 16% (maximum presented) was 82% for a 4x4 mesh. Our success rate for an 8x8 mesh with the P_{EX} of 20% was 100% regardless the number of subnets (4, 6 or 8).

2) Path length

The 7th column presents the Manhattan distance between communicating tasks. It is worth to note that the average and standard deviation values for the scenarios are similar, regardless the system size. As the system occupation increases, the number of contiguous regions to map the tasks reduces, explaining the maximum hop number.

The 8th column presents the path length (CS hops). The algorithm succeeds to establishes short paths, reducing the interference for the next connections, which contributes to increasing the *success rate*.

3) SEARCH-PATH Execution Time

The 9th column presents the SEARCH-PATH execution time, in clock cycles (*cc*), including the execution time of the algorithm and the configuration of the CS routers through the PS subnet. For all 20x20 scenarios (worst cases), the higher average value is 8,521 *cc*, or 8.5 µs@100 MHz. Figure 8(b) plots the SEARCH-PATH execution time as a function of the P_{EX} (bottom x-axis) and CS requests (top x-axis), for 20x20:6 scenarios, respectively. Compare Figure 8(a) with Figure 8(b). The execution time in Figure 8(b) remains near 7,029 *cc* up to a P_{EX}=31.3% (third quartile). After this point, the minimal paths become scarce, requiring the search of non-minimal paths. With a P_{EX}=38%, the execution time reaches the worst case, failing to establish some CS paths

Figure 9 presents the average execution time for the SEARCH-PATH algorithm as a function of the number of PEs and CS subnets. The execution time grows linearly with the system size at an average rate of 33.27 *cc*/PE. The 10th column evaluates the impact of the proposed SDN-based CS on the applications' start time. In the worst case, applications were delayed by 4.5 *ms*. This delay only occurs when the applications begin the execution, being not noticed by the end user during the application execution. Some related works [4][14] may assume a new CS connection for each application's message, with an overhead proportional to the messages volume.

TABLE III. CIRCUIT-SWITCHING RESULTS FOR SIMULATIONS WITH 100% OF RT APPLICATIONS AND A 100% OF SYSTEM OCCUPATION.

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
Scenario – system size	**CS nets**	**Simultaneous Running Tasks**	**Path Diversity**	**CS requests (P_{EX})**	**Success Rate**	**Manhattan Distance** *Avg. | Std.Dev | Max.*	**CS Hops** *Avg. | Std.Dev | Max.*	**SEARCH-PATH time** *Avg. | Worst.*	**App. Start Overhead** *Average (ms.)*
8x8 (64 PEs)	4	120 (100% occupation)	256	127 (49%)	94.4%	2.6 | 1.6 | 8	2.8 | 1.9 | 14	4258 | 48614	1.4
	6		384	127 (33%)	99.2%		2.7 | 1.6 | 8	3821 | 13166	1.6
	8		512	127 (24%)	100%		2.6 | 1.6 | 8	3725 | 12158	1.1
16x16 (256 PEs)	4	480 (100% occupation)	1024	623 (60%)	88.6%	2.5 | 2.0 | 21	2.7 | 2.1 | 21	5336 | 52965	3.1
	6		1536	623 (39%)	97.7%		2.6 | 2.1 | 22	5326 | 102433	3.9
	8		2048	623 (30%)	99.8%		2.5 | 2.0 | 21	5220 | 38361	3.5
20x20 (400 PEs)	4	768 (100% occupation)	1600	916 (57%)	90.94%	2.7 | 2.1 | 26	3.0 | 2.7 | 30	8521 | 486180	4.0
	6		2400	916 (38%)	98.25%		2.7 | 2.3 | 26	7716 | 174033	4.5
	8		3200	916 (28%)	100%		2.7 | 2.1 | 26	7121 | 107214	2.9

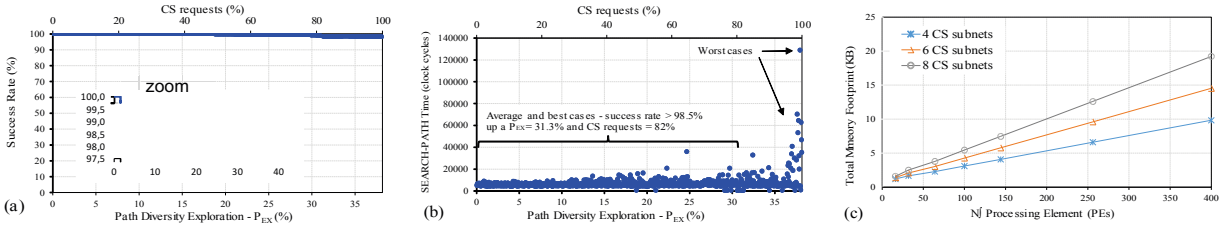

Figure 8. Performance results: (a) success rate for 20x20:6 CS subnets. (b) SEARCH-PATH exec. time for 20x20:6. (c) *CS-Controller* memory requirement.

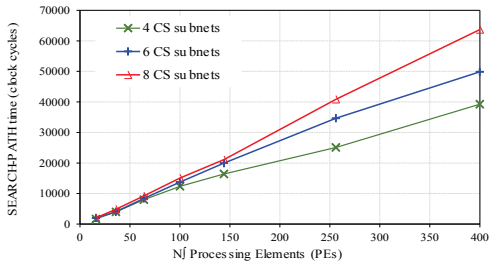

Figure 9. SEARCH-PATH average execution time.

4) *Memory Footprint*

Figure 8(c) presents the memory footprint as a function of the number of the PEs. This memory usage includes data (Equation 1) and code of the *CS-Controller*. The memory requirement increase at a linear rate of 0.03, 0.05, and 0.06 KB/PE, for 4, 6, 8 CS subnets, respectively. The total storage required for a 20x20:8 system is 19.22 KB, a small amount of memory for a many-core system.

C. *Application's Latency Evaluation*

Figure 10 presents the frame decoding latency of the MPEG-2 decoder using scenarios: PS without disturbing traffic (*PS Baseline*); PS with disturbing traffic (*PS+Disturbing*); *CS*.

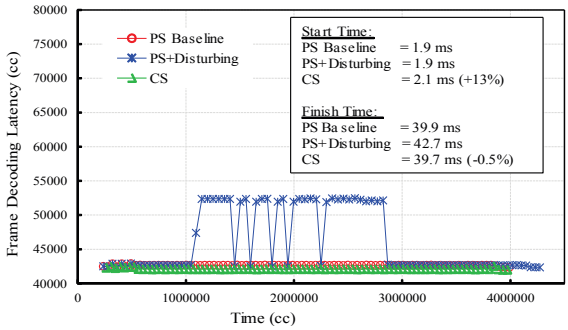

Figure 10. MPEG-2 start time and frame decoding latency.

The *PS+Disturbing* scenario shares the PS NoC with packets from other applications, resulting in increased frame decoding latency and jitter. The *CS* scenario presents a constant and predictable latency, with a latency smaller than the *PS Baseline* scenario due to the streaming behavior of the CS routers, transmitting one flit/cycle (without routing and arbitration logic). Note that the application executes faster using CS, despite the delay to start. The results of Figure 10 address an application running for only 39.9 *ms*. The impact of the delay to start the application is even more negligible for longer execution time, once no new CS connections are required during the application execution.

VII. CONCLUSION

This work proposed a CS infrastructure for many-cores based on the SDN paradigm. This approach enabled to design a simple MPN for CS, through configurable CS routers based on Elastic-Buffers. The adoption of a systemic CS management fills the gap in the literature related to the evaluation only at the NoC context. This work provides permanent CS for the RT applications, providing communication predictability. The search path based on the Hadlock's algorithm, plus the rich path diversity provided by MPN, enabled the proposal to achieve a connection success rate higher than 97% for MPNs with 6 and 8 CS subnets executing real applications.

Future works include: remap tasks when CS failed; extend the SDN-based approach to other services, as fault-tolerance and security; include the proposal with RT time schedulers ensuring RT support for both communication and computation levels.

ACKNOWLEDGEMENT

The Author Fernando Gehm Moraes is supported by CNPq funding agency.

REFERENCES

[1] Dut, N.; Jantsch, A.; Sarma, S. *Toward Smart Embedded Systems: A Self-aware System-on-Chip (SoC) Perspective*. ACM. Trans. Embed. Comput. Syst., v. 15(2), 27 pages, Feb. 2016.

[2] T. Bjerregaard and J. Sparso. *A router architecture for connection-oriented service guarantees in the mango clockless network-on-chip*. In: DATE, 2005.

[3] K. Goossens, J. Dielissen, and A. Radulescu. *Aethereal network on chip: concepts, architectures, and implementations*. IEEE Design & Test of Computers, 22(5):414–421, 2005.

[4] Leroy, A.; et al. Concepts and Implementation of Spatial Division Multiplexing for Guaranteed Throughput in Networks-on-Chip. IEEE Trans. on Computers, v.57(9), pp. 1182-1195, Sept. 2008.

[5] Jarraya, Y.; Madi, T.; Debbabi, M. *A Survey and a Layered Taxonomy of Software-Defined Networking*. IEEE Communications Surveys & Tutorials, v.16(4), pp. 1955-1980, Fourth quarter 2014.

[6] Kirkpatrick, K. *Software-defined networking*. Communications of the ACM, v.56(9), pp 16-19, Sept 2013.

[7] Michelogiannakis, G.; Dally, W. *Elastic Buffer Flow Control for On-Chip Networks*. IEEE Trans. on Computers, v.62(2), pp. 295-309, Feb. 2013.

[8] Michelogiannakis, G.; Becker, D.; Dally, W. *Evaluating Elastic Buffer and Wormhole Flow Control*. IEEE Trans. on Computers, v.60(6), pp. 896-903, June 2011.

[9] Yoon, J.; et al. *Virtual Channels and Multiple Physical Networks: Two Alternatives to Improve NoC Performance*. IEEE Trans. on CAD of ICs and Systems, v.32(12), pp. 1906-1919, Dec 2013.

[10] Ejaz, A.; Jantsch, A. Costs and benefits of flexibility in spatial division circuit switched networks-on-chip. In: NoCArc, 2013, pp. 41-46.

[11] Liu, S.; Jantsch, A.; Lu, Z. *MultiCS: Circuit switched NoC with multiple sub-networks and sub-channels*. Journal of Systems Architecture, v.61(9), pp. 423-434, Oct 2015.

[12] Stefan, R.; Nejad, A.; Goossens, K. *Online allocation for contention-free-routing NoCs*. In: INA-OCMC, 2012, pp 13-16.

[13] Chen, Y.; Matus, E.; Fettweis, G. P. Trellis-search based Dynamic Multi-Path Connection Allocation for TDM-NoCs. In: GLSVLSI, 2016, pp. 323-328.

[14] Liu, S.; Jantsch A.; Lu Z. Parallel probe based dynamic connection setup in TDM NoCs. In: DATE, 2014, 6p.

[15] Lusala, A.; Legat, J. Combining SDM-BASED Circuit Switching with Packet Switching in a NoC for Real-Time Application. In: ISCAS, 2011, pp. 2505-2508.

[16] Hadlock, F. *A shortest path algorithm for grid graphs*. Networks, v.7(4), pp. 323-334, Winter 1977.

[17] Sherwani, N. *Algorithms for VLSI Design Automation*, 3rd ed., Springer, 2005.

[18] Mello, A.; et al. Virtual Channels in Networks on Chip: Implementation and Evaluation on Hermes NoC. In: SBCCI, 2005, pp. 178-183.

[19] Ruaro, M and Moraes, F. G. *Dynamic real-time scheduler for large-scale MPSoCs*. In: GLSVLSI, 2016, pp. 341-346.

[20] Singh, A.; et al. Mapping on multi/many-core systems: Survey of current and emerging trends. In: DAC, 2013, 10p.