# Distributed SDN Architecture for NoC-based Many-core SoCs

Marcelo Ruaro
marcelo.ruaro@acad.pucrs.br
PUCRS, Brazil
TU Wien, Vienna, Austria

Nedison Velloso
nedisonnfv@gmail.com
PUCRS, Porto Alegre
Brazil

Axel Jantsch
axel.jantsch@tuwien.ac.at
TU Wien, Vienna
Austria

Fernando G. Moraes
fernando.moraes@pucrs.br
PUCRS, Porto Alegre
Brazil

## ABSTRACT

In the Software-Defined Networking (SDN) paradigm, routers are generic and programmable forwarding units that transmit packets according to a given policy defined by a software controller. Recent research has shown the potential of such a communication concept for NoC management, resulting in hardware complexity reduction, management flexibility, real-time guarantees, and self-adaptation. However, a centralized SDN controller is a bottleneck for large-scale systems.

Assuming an NoC with multiple physical subnets, this work proposes a distributed SDN architecture (D-SDN), with each controller managing one cluster of routers. Controllers work in parallel for local (intra-cluster) paths. For global (inter-cluster) paths, the controllers execute a synchronization protocol inspired by VLSI routing, with global and detailed routing phases. This work also proposes a short path establishment heuristic for global paths that explores the controllers' parallelism.

D-SDN outperforms a centralized approach (C-SDN) for larger networks without loss of success rate. Evaluations up to 2,304 cores and 6 subnets shows that: (*i*) D-SDN outperforms C-SDN in path establishment latency up to 69.7% for 1 subnet above 32 cores, and 51% for 6 subnets above 1,024 cores; (*ii*) D-SDN achieves a smaller latency then C-SDN (on average 54%) for scenarios with more than 70% of local paths; (*iii*) the path success rate, for all scenarios, is similar in both approaches, with an average difference of 1.7%; (*iv*) the data storage for the C-SDN controller increases with the system size, while it remains constant for D-SDN.

## CCS CONCEPTS

• **Computer systems organization** → **Multicore architectures**; System on a chip; • **Hardware** → *Networking hardware*;

## KEYWORDS

Network-on-Chip (NoC), Software-Defined Networking (SDN), Distributed Management, System-on-Chip (SoC), Many-core.

**ACM Reference format:**
Marcelo Ruaro, Nedison Velloso, Axel Jantsch, and Fernando G. Moraes. 2019. Distributed SDN Architecture for NoC-based Many-core SoCs . In *Proceedings of International Symposium on Networks-on-Chip, New York, NY, USA, October 17–18, 2019 (NOCS '19),* 8 pages.
https://doi.org/10.1145/3313231.3352361


*NOCS '19, October 17–18, 2019, New York, NY, USA*

ACM ISBN 978-1-4503-6700-4/19/10…$15.00
https://doi.org/10.1145/3313231.3352361

## 1 INTRODUCTION

A state-of-the-art Many-Core SoC (MCSoC) contains up to hundreds of homogeneous or heterogeneous processing elements (PEs), leading to significant design complexity. While researchers have proposed solutions to separately provide QoS, fault-tolerance, aging, security, energy, power and temperature management, the next SoC generation must integrate and combine available techniques to solve all these issues simultaneously in a scalable, multi-objective, and self-adaptive management. The need for such comprehensive design drives the research for self-adaptation properties [5].

Networks-on-Chip (NoCs) implement parallel and scalable communication in MCSoCs [6]. To obtain self-adaptation in the communication infrastructure, the MCSoC needs to be able to know and control the NoC resources at runtime. Past works for NoC management focus on hardware-centric approaches, with specific router designs [1, 6]. These approaches have the advantage of high-performance. However, they do not allow effectively to manage the network due to their lack of a holistic view of the system. Software-based management also has been explored to meet specific purposes, as to manage dynamic Time Division Multiplexing (TDM) [20] or Spatial Division Multiplexing (SDM) allocation for real-time guarantees [11].

The Software-Defined Networking (SDN) paradigm [14] is a concept in the computer network field and stands as a promising proposal to be applied for NoC management in MCSoCs. SDN means to remove the communication control logic from the hardware level bringing it to the software level. This approach simplifies the on-chip communication architecture since it promotes a generic and simple communication design paradigm, and also leveraging to a self-adaptive communication infrastructure due to its global view of the system and flexible management. In SDN, routers that conventionally have its design customized for a specific purpose, become programmable hardware units, with the ability to change its connections according to applications' needs. The routers are configured by a software SDN Controller, which establishes and releases connections between PEs at runtime following a single or multi-objective policy.

Recent research highlight the benefits of the SDN paradigm for NoC management in MCSoCs [7, 10, 17], which can be summarized as: (*i*) hardware complexity reduction; (*ii*) management flexibility; (*iii*) real-time guarantees; (*iv*) security; (*v*) self-adaptation.

SDN presents two challenges: (*i*) higher path setup latency compared to hardware-based approaches (typically two orders of magnitude higher); and (*ii*) lack of scalability due to the adoption of a centralized controller. Challenge (*i*) is inherent to software-based approaches, such overhead is not prohibitive when the connections do not need to be made for every packet (which is costly even for hardware approaches), and the priority is flexible and multi-objective management. Recent research [4, 16, 17] show that the

setup latency can be mitigated establishing connections at the application startup (in parallel o to the object code loading) and releasing at the end of application execution. Thus, the applications use the connections previously established, without the need to wait for new connections during their execution.

Regarding challenge (*ii*), related work on SDN for NoC assume only centralized controllers, without evaluations for large-scale systems (more than 256 PEs). To achieve scalability, computer networks already adopt distributed SDN management [14].

Thus, motivated by challenge (*ii*), this work has as *main goal* the proposition and evaluation of a distributed SDN architecture for large scale MCSoCs. Routers are grouped into clusters, with one controller per cluster. For *local paths* (i.e., intra-cluster paths) each controller defines the paths independently and in parallel to other controllers. For a *global path*, (i.e., extra-cluster paths), the controllers cooperate in a synchronized way to manage the path setup and release. The global path establishment is inspired by VLSI routing [3], with global and detailed routing phases and the adoption of the Hadlock shortest path search algorithm[3].

The *contributions* of this work can be summarized as follows: (1) a cluster-based SDN architecture; (2) a synchronized path establishment and release protocol among clusters; (3) a heuristic to define short paths among clusters.

The rest of this paper is organized as follows. Section 2 discusses related work, positioning our work w.r.t the state-of-the-art. Section 3 presents the cluster-based SDN architecture (contribution 1). Section 4 details the distributed SDN protocol (contribution 2). Section 5 presents the heuristic to define short paths (contribution 3). Section 6 evaluates the SDN proposal, and Section 7 concludes this paper.

## 2 RELATED WORK

Table 1 classifies the related work on SDN for NoCs, detailing the similarities and differences to the work herein proposed.

Table 1, 2$^{nd}$ column details the SDN proposal goal. Works [7, 10, 18] use SDN for specific purposes as QoS, security and power saving. Works [2, 4, 8] propose a generic SDN architecture as proposed in this work. The rules implemented inside the controller can be designed according to different policies, for fault-tolerance, security and QoS.

Table 1, 3$^{rd}$ column details the adopted NoC architecture that are majority based on a 2D-mesh topology [2, 4, 7, 8, 18], due to its easy implementation and reusability. Berestizshevsky et al. [2] use two physical networks, one for control and another one for the application's data. We adopt a Multiple Physical Network (MPN) architecture. MPNs enable communication isolation and a higher path diversity by implementing several simple and replicated networks. Previous works show the benefits to adopt MPNs. Authors in [6, 12] compare MPNs to SDM. With an equivalent bandwidth, MPN increases the area following an $O(n)$ complexity, while SDM increases $O(n^2)$. The path delay increases with the number of subchannels (SDM) with a complexity of $O(n)$, while it is constant for MPN. Yoon et al. [21] present a comparison of MPN and Virtual Channels (VC) and conclude that MPN presents better area scalability. Also, MPNs scale better regarding power dissipation and are better suited for new technology nodes.

Table 1, 4$^{th}$ column details the evaluated system sizes (i.e., maximum PE count). An important observation can be drawn from this column: as most works adopt a centralized controller, evaluations are limited to a small number of PE count ($\leq$ 256). The work of [18] addresses up to 1,024 PEs; however, the SDN concept adopted in this work follows a different paradigm, without an SDN controller able to achieve global and self-aware communication management. The system size of 16x16 found in the related work points to a possible limit of the centralized control in SDNs, as empirically observed in [2, 4]. This work aims to address such limitation by proposing a distributed SDN management.

Table 1, 5$^{th}$ column qualitatively compares the router complexity. Works [4, 7, 8] embed additional hardware features in the router design increasing its complexity. We argue that in SDN, routers must be as simple as possible, having only the role to forward packets and to accept configuration from a controller. Such design philosophy drives the SDN router design of computer networks [14], which this work also follows, as detailed in Section 3.1.

Table 1, 6$^{th}$ column presents works that validate the proposal using RTL simulations. Our work uses a SystemC-RTL model for simulations. We argue this level is necessary since it enables to measure the real overheads of the proposed methods. Virtual simulators as Mininet [7], and Omnet++ [2] can hide communication bottlenecks and deadlocks in an MCSoC context.

The last column of Table 1 presents the method to design the SDN controller. Most works adopt a centralized controller [2, 7, 8, 10]. Consequently, the study of distributed SDNs is the main motivation and goal of the work herein presented.

## 3 SDN ARCHITECTURE

This Section presents two SDN architectures, the baseline centralized SDN (C-SDN) architecture (based on [17]), and the proposed

**Table 1: Related work in SDN for NoCs.**

| Work | Goal | NoC Architecture | System Size | Router Complex. | RTL | SDN Controller |
|---|---|---|---|---|---|---|
| Cong et al.[4] 2014 | Generic SDN | Mesh, 2D, One physical network | 256 | High | No | Inside each router |
| Sand. et al.[8] 2016 | Generic SDN | Mesh, 2D, One physical network | 70 | High | Yes | Centralized, Intra-chip |
| Bere. et al.[2] 2017 | Generic SDN | Mesh, 2D, Two physical networks | 256 | Minimal | No | Centralized, Intra-chip |
| Scion. et al.[18] 2018 | SDN for power saving | Mesh, 2D, Two hierarch. networks | 1,024 | Medium | Yes | Does not exist, made by each OS |
| Kostrzewa et al. [10] 2018 | SDN for QoS | Torus, 2D, One physical network | 10 | N.A. | Yes | Centralized, Intra-chip |
| Ellinidou et al. [7] 2019 | SDN for security | Mesh, 2D, Multiple many-core chips | 30 | High | No | Centralized, Off-chip |
| **This work** | **Generic SDN** | **Mesh, 2D, Multiple physical networks** | **2,304** | **Minimal** | **Yes** | **Distributed, Intra-chip** |

distributed SDN architecture (D-SDN). Figure 1(a) overviews the assumed baseline MCSoC architecture. It contains a set of PEs interconnected by an NoC. Each PE has a CPU, a local scratchpad memory, a Network Interface (NI), and a Packet-Switching (PS) router. The presence of a shared-memory is also frequent, with the local memory assuming the role of an L1 cache. In Figure1(a), the PEs are homogeneous, but heterogeneous PEs can also be assumed. Shared-memory and heterogeneous systems are orthogonal features to our proposal.

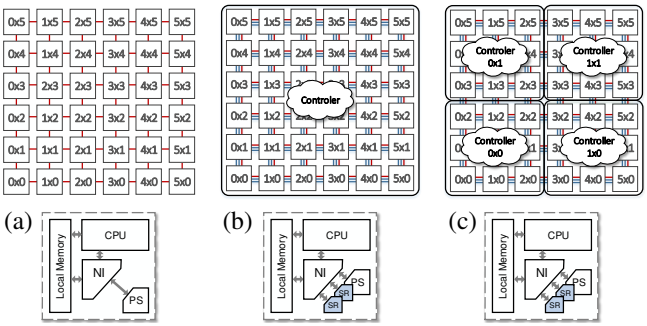


**Figure 1: (a) Baseline MCSoC, (b) C-SDN, (c) proposed D-SDN with a cluster-based SDN management.**

Figure 1(b) depicts the C-SDN. This architecture contains an MPN, with one PS subnet and a set of Circuit-Switching (CS) subnets with SDN routers (SR). The subnets are not connected to each other. The PS subnet is shared between best-effort applications' data and SDN control data. Each PS router has input-buffers (usually 8-flit depth), credit-based control flow, and wormhole packet switching (PS).

Figure 1(c) presents the proposed D-SDN. The hardware architecture is the same as the C-SDN. The proposed software architecture comprises of several *controllers*, and each one is in charge to manage a cluster of SRs. The definition of the cluster size occurs at design-time.

## 3.1 SDN Router (SR)

The SR, Figure 1(b) and (c), is a forwarding unit that connects a given input port to an output port. Instead of using input buffers, SRs use Elastic Buffers (EB) [13]. A crossbar connects the input port with its respective output port. The SDN controller configures the crossbar through a configuration packet. The SR has a configuration interface connected to the NI. When the NI detects the configuration packet in the PS subnet, it drops the packet, avoiding it to be handled by the CPU, extracts its contents, and configures one of the SRs. It has been shown that the 32-bit SR router area (1,177 gates) amounts to 25% of an 8-buffer depth 32-bit PS router, without virtual channels [17].

## 3.2 SDN Controllers

The *C-SDN controller* is a software task with exclusive access to OS services and assigned to some PE. The C-SDN controller implements the Hadlock shortest-path algorithm [9] that find paths in a given CS subnet. The CS subnet is selected following a utilization-based heuristic among all CS subnets [17].

The *D-SDN* management is implemented as a high priority distributed application. Each task of the application is a controller. The controllers work in parallel for local paths. When a global path is requested, the controllers execute the distributed SDN protocol.

# 4 DISTRIBUTED SDN PROTOCOL

This Section details the distributed protocol to establish and release a path. When a given path requester (e.g., application, OS, hardware module, among others) needs a path establishment or release for communicating PEs it sends a `PATH_REQ` message to the *controller* of the cluster where the source PE address is located. Such a controller becomes a temporary *coordinator*, which is responsible for coordinating a global path establishment or release.

## 4.1 Global Path Establishment

Global path establishment requires a consistent global state of the network, achieved through the communication between controllers. A *token manager* provides a token to one controller of the system at each time.

Figure 2(a) presents the global path establishment protocol. The requester asks a path to the controller. The receiving controller becomes the *coordinator* of the path, storing requests into a FIFO. The FIFO size is equal to the number of system PEs. The coordinator starts to handle a new global path request by removing it from the FIFO, sending a `TOKEN_REQUEST` message to the token manager, and continues in an *IDLE* state working on local paths only. When the coordinator receives a `TOKEN_GRANT` message it begins to execute the path establishment protocol, divided into three phases.

***Phase 1 - Consistency***. Consistency is the phase where the coordinator achieves the global view of network. It sends a `BORDER_STATUS_REQ` message to all controllers of the system. These controllers stop to handle new requests and start to work in cooperation with the coordinator to define a global path. All controllers execute in parallel the *update_border*() function, which creates the status of the cluster border ($B_{stat}$). The border status of each cluster is gathered at the coordinator after it receives all `BORDER_STATUS_ACK` messages.

***Phase 2 - Path search***. The coordinator starts to search for the path. Figure 2(b) overviews the process. The process is hierarchically divided in global and detailed routing. Global routing is a coarse-grain search heuristic that selects the clusters where the path will traverse. Detailed routing is a fine-grain heuristic that defines the intra-cluster routers of the path. After selecting the cluster in global routing, the coordinator sends a `DETAILED_ROUTING_REQ` to the involved controllers. Both the coordinator and controllers execute the detailed routing. When a given controller finishes the detailed routing, it sends a `DETAILED_ROUTING_RESPONSE` to the coordinator, informing about path success or failure. If one of the controllers fails in the detailed routing, the coordinator either can choose to re-execute the global routing with different parameters or it concludes that such path cannot be created. If successful, the protocol advances to the last phase.

***Phase 3 - Path Configuration***. With a `GLOBAL_PATH_END` message the coordinator notifies all controllers that the previous phase ended. Controllers not involved in the path just send a `GLOBAL_PATH_END_ACK` message to the coordinator, and are released to handle new path requests. The controllers belonging to the path define their intra-cluster paths by sending configuration packets to each SR router. Note, that the configuration process is also executed in parallel since each controller keeps the list of its path's

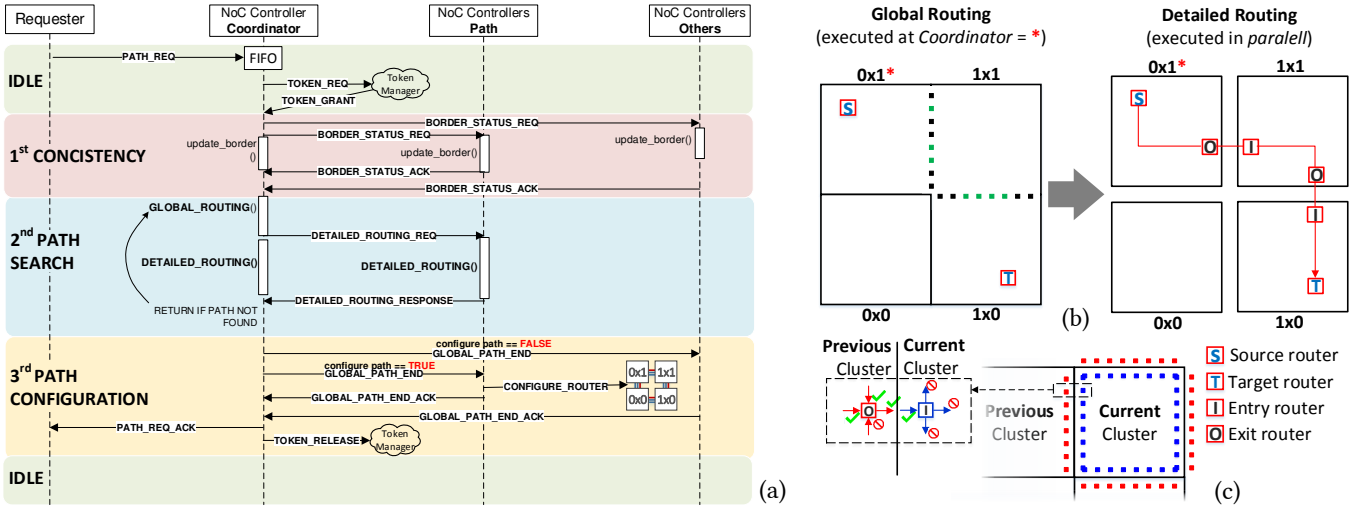


**Figure 2: (a) Global path protocol. (b) Global and detailed routing. (c) Representation of the cluster's border status ($B_{stat}$).**

routers. After the end of the configuration, the controller sends a GLOBAL_PATH_END_ACK message to the coordinator. When the coordinator receives all messages from the controllers, it concludes the path establishment protocol by sending a PATH_REQ_ACK message to the path requester and releases the token at the token manager.

The information related to the border status is mandatory since it represents the free crossing points between clusters. The first phase of the protocol obtains this information. Figure 2(c) illustrates how the *update_border*() function computes the border status. The goal is to return the status of the crossing points. The simplest solution would be to gather the status of the input port of the input router (blue router) at *current* cluster. However, we observed that this approach leads to a poor path success rate during detailed routing, since the router could be blocked due to the usage of all its neighbors' inputs (as is the case of the blue router of Figure 2(b)). The same applies to the output router (red router) of the *previous* cluster. In a worst-case scenario, a path can not reach it since all its input ports could be already blocked (this is not the case for the red router because only its south input is blocked).

To compute the border status accurately, we evaluate all inputs, as illustrated in Figure 2(c). A path can cross from a cluster to another if the output router can be reached for, at least, one input port, and if the input router has at least one free output port.

The communication complexity of this protocol in terms of number of messages is $O(4.N + 2.N.GR_a + 2.M)$, where $N$ is the number of controllers in the system, $M$ is the path's length (hops), and $GR_a$ is the number of global routing re-executions. The computational complexity is related to the *update_border*() and global and detailed routing heuristics. The *update_border*() complexity is $O(4.CS_n)$, where $CS_n$ is the number of CS subnets, and 4 is the border number. Section 5 presents the complexity of both heuristics.

## 4.2 Global Path Release

The *requester* starts the path release protocol, by notifying the *coordinator*. The path release is entirely done is software, using one structure (*sr_input*) that stores the status of each SR's input port. Thus, it is not necessary to release physically the SR. The controller releases a path by tracking *sr_input* from the source router until reaching a cluster border or the target router. When the path release reaches a cluster border, the controller sends a LOCAL_RELEASE_REQ message to the controller of the next cluster. If the release process reaches the target router, the controller checks if it is the coordinator of the path. If the controller is not the coordinator, it sends a LOCAL_RELEASE_ACK message to the coordinator. The coordinator finishes the path release protocol by sending the response to the path requester.

This protocol does not require a token to start its execution because the release process does not require a global view of the system, i.e., it is local to each cluster.

The communication complexity in terms of number of messages is $O(N)$, where $N$ is the number of clusters that the path crosses. The computation complexity is $O(NR)$ where $R$ is the path' length in hops.

# 5 *SPESH* HEURISTIC

This Section details the path establishment heuristic, *SPESH* – Short SDN Path Establishment Heuristic. Paths found by the heuristic may not be the shortest ones, because the global routing reduces the path search space due to its coarse grain search method [3].

## 5.1 Global Routing

Algorithm 1 presents the *GLOBAL_ROUTING()*. The global routing goal is to find clusters for a given path. The global routing only selects the clusters, and not the entry and exit point of each cluster. Hence, after global routing, the coordinator sends the border information (cluster side and status of each crossing point) to each controller, and each one computes the crossing point in parallel, as part of the detailed routing. Such approach reduces the global routing overhead.

The *GLOBAL_ROUTING()* algorithm assume as inputs:

$CS_n$: number of Circuits Switched (CS) subnets;
$S_c, T_c$: addresses of the source and target clusters, respectively;
$GS_u$: global subnet utilization, stores the utilization of each system's cluster subnet;

$AV_c$: available controllers, stores the controllers that failed during detailed routing. Initialized with all controllers as available;

$GR_a$: global routing attempts, stores the number of *GLOBAL_ROUTING()* calls made by coordinator. Initialized with 1. Incremented when global routing is re-executed;

$CB$: cluster border status, stores the status of all input/output ports that are in the 4 borders of each system's cluster.

The first phase of the protocol, *consistency*, fills the $GS_u$ and $CB$ variables.

**ALGORITHM 1:** GLOBAL_ROUTING

**Input:** $CS_n$, $S_c$, $T_c$, $GS_u$, $AV_c$, $GR_a$, $CB$
**Ouput:** set C, *subnet*

| | |
|---|---|
| $CS_n$ | Number of CS subnets |
| $S_c$ | Source cluster address (coordinat.) |
| $T_c$ | Target cluster address |
| $GS_u$ | Global subnet utilization |
| $AV_c$ | Available controllers |
| $GR_a$ | Global routing attempts |
| $CB$ | Clusters border input and output |
| $PL_{cur,\ min}$ | Current and minimal path length (hops) |
| $U_{temp,\ min}$ | Temp. and minimal subnet utilizat. |
| C | Set of cluster addresses |

```
1.  C ← Ø
2.  Umin ← MAX_UTIL + 1
3.  PLmin ← MAX_LENGTH + 1
4.  min_path ← TRUE
5.  If GRa > 1 then
6.      min_path ← FALSE
7.  EndIf
8.  Do
9.    For s in range 1 to CSn do
10.     PLcur ← HADLOCK(Sc, Tc, CB, AVc, min_path)
11.     If PLcur > 0 then
12.       Ctemp ← RETRACE()
13.       Utemp ← compute_path_subnet_util(Ctemp, s, GSu)
14.       If Utemp < Umin or (Utemp = Umin and PLcur < PLmin) then
15.         Umin ← Utemp
16.         PLmin ← PLcur
17.         C ← Ctemp
18.         subnet ← s
19.       EndIf
20.     EndIf
21.   EndFor
22.   prev_min_path ← min_path  // enables two iterations of the loop when GRa=1
23.   min_path ← FALSE
24. While C = Ø and prev_min_path = TRUE
25. Return C, subnet
```

The default operation mode of the algorithm is to find minimal global paths, $min\_path = TRUE$ (line 4). If the number of attempts to execute the global routing is higher than one ($GR_a > 1$), the algorithm switches to shortest paths, $min\_path = FALSE$ (line 6).

The internal loop, lines 9-21, seeks a path for all subnets. Line 10 executes the *Hadlock* algorithm at cluster level to find a cluster path, returning the path length ($PL_{curr}$). If the *Hadlock* algorithm succeeded (line 11), the *retrace* procedure is executed, returning the set $C_{temp}$, with the clusters' addresses in the path (line 12). The next step computes the impact of the path in the subnet utilization (line 13). If the cluster subnet utilization reduces, or if it is the same but with a shortest path length (line 14), the path is set as a candidate path (lines 15-18). The algorithm distributes the communication load because it selects the subnet that achieved lower utilization. This approach also helps to increase the success rate during the detailed routing since the path uses the clusters with a smaller load.

Next, the path search mode switches to shortest paths (line 23). If the previous search mode was set to find only minimal paths and the path was not found (line 24), the internal loop executes one more time. Thus, the algorithm explores all subnets, with minimal and non-minimal paths lengths, trying to reduce the subnets utilization.

The *GLOBAL_ROUTING()* algorithm finishes returning the selected subnet, and a set $C = \{c1_{x,y}, c2_{x,y}, ...\}$ with the addresses of the clusters that the path will cross. With $C = \emptyset$ the coordinator sends a GLOBAL_PATH_END message to all controllers finishing the global path search. With $C \neq \emptyset$, the coordinator sends the DETAILED_ROUTING_REQ for the clusters in $C$, starting the detailed routing at each controller.

The *coordinator* verifies if the detailed routing failed in some cluster at a given subnet after receiving all DETAILED_ROUTING_RESPONSE messages. The $AV_c$ array set the index related to all clusters/subnet where the detailed routing failed as not available for the next global routing execution. The $GR_a$ increments if the detailed routed failed in some cluster, which leverages the global routing re-execution.

The number of runs that the algorithms make is a trade-off between runtime and success rate to find the paths. Our experiments adopt empirically the number of re-execution to be less or equal to $CS_n$ ($GR_a \leq CS_n$), as we observed that the success rates slightly improve increasing the number of iteration.

The global routing complexity is $O(CS_n \cdot C_H)$, where $C_H$ is Hadlock's complexity $C_H = O(X \cdot Y)$, with $X$ and $Y$ the numbers of clusters along the X and Y axes.

## 5.2 Detailed Routing

The detailed routing has two steps: (*i*) from the entry and exit border information sent by the coordinator, select the entry and the exit routers of the cluster; (*ii*) find the shortest path between these two routers using the Hadlock's algorithm at router's level.

A controller receiving a DETAILED_ROUTING_REQ message should select two routers addresses to establish the path. If the path starts or ends in the cluster, one of the routers's address is the source or target routers' addresses of the path ($S_r$ and $T_r$). Otherwise, the path crosses the cluster's borders.

The algorithm tries to align the crossing point to $S_r$ or $T_r$. This approach maximizes the chances to find a minimal path since it tries to find a router inside a rectangle defined by $S_r$ and $T_r$ addresses. Note that the crossing point selection is synchronized with neighbor clusters without the need for communication between them since they have the same $B_{stat}$ (border status) sent by the coordinator.

Figure 3 shows an example of crossing point selection at the borders of cluster 1x1. The entry border is located at the west side, and exit border in the south side. The green dots represent free crossing points, and the black ones represent blocked crossing points. The example adopts a 16x16 system, with an 8x8 cluster.

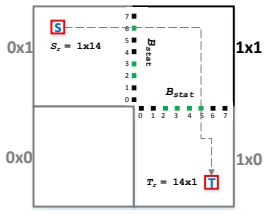


**Figure 3: Example of crossing point selection at cluster 1x1.**

For the entry border, west, the free crossing points indexes are 2, 3, 6. In this case, the crossing point selection algorithm chooses index 6 because it is at same axis than $S_r$. For the exit border, south, the free crossing points indexes are 2, 3, 4, 5. In this case, the selected index is 5 since it is the nearest index to $T_r$.

The detailed routing executes the Hadlock's algorithm locally (router's level) after selecting the routers' addresses. The Hadlock in this level uses the subnet selected in global routing, and the entry and exit routers addresses selected in the first step. After the Hadlock execution, the controller sends the DETAILED_ROUTING_

RESPONSE message to the coordinator informing if the path was found or not.

The detailed routing complexity is $O(2N + C_H)$, where $N$ is the cluster border size in number of PEs, 2 is due the double execution of step 1. $C_H$ is Hadlock's complexity $C_H = O(X \cdot Y)$, with $X$ and $Y$ as the cluster dimensions.

## 6 EXPERIMENTAL RESULTS

This Section presents the experimental results. Work [16] evaluates the centralized SDN (C-SDN) against a distributed hardware-based search path algorithm, with an average difference on the path search success equal to 0.23%. Such result allows defining the C-SDN approach as the baseline reference.

### 6.1 Experimental Setup

A SystemC-RTL description models the hardware. The software is modeled in C code (*mips-gcc* cross-compiler, version 4.1.1, optimization *O2*). The CPU running the controller is a Plasma processor [15] (MIPS processor) at 100 MHz.

Each scenario has a set of $total\_path$ requisitions with the source-target pairs randomically generated according to a locality rate $L$. The size of $total\_path$ is equal to $\#PE \cdot CS_n$, corresponding to the maximum number (worst-case) of simultaneous paths that can be established into the system. $L$ is the local path rate, corresponding to the number of local paths in $total\_path$.

A synthetic task (*requester*) serves as path requester. The number of requester tasks is the same as the number of controllers. The number of path requests per requester is $\frac{total\_path}{number\ of\ clusters}$.

Results evaluate the path establishment, using the following performance figures:

***Total Latency***: Time that each approach needs to establish *total_path*. It comprises the time from the first PATH_REQ to the last PATH_REQ_ACK. Includes FIFO time and token waiting time. The parallelism between controllers favors this metric when establishing local paths.

***Setup Latency***: Individual path setup latency. Time measured at the *coordinator*, from the moment that it starts to define a path up to its configuration. Excludes FIFO time and the token waiting time in case of global paths.

***Success rate***: $\frac{success\_paths}{total\_paths}$.

### 6.2 Scalability Evaluation

This section evaluates the scalability of the approach, by varying the system size, $CS_n$, and with $L = 0.8$. Five system sizes are evaluated: 36 PEs – 6x6(3x3) (6x6 system, 3x3 clusters), 64 PEs – 8x8(4x4), 256 PEs – 16x16(8x8), 1024 PEs –32x32(16x16), 2304 PEs – 48x48(16x16). The reason to adopt $L = 0.8$ comes from the fact that task mapping heuristics try to place communicating tasks near to each other [19].

The goal of the following experiment is to determine when D-SDN is faster than C-SDN. Figure 4(a) presents the *total latency* with D-SDN values normalized in relation to C-SDN. For $CS_n = 1$, the D-SDN is always faster than the C-SDN, reaching an improvement of 69.7% for 2,304 PEs. For $CS_n = 6$, D-SDN outperformed C-SDN for all system's sizes starting from 1,024 PEs and reaching an improvement up to 51% for 2,304 PEs. The *total latency* increases with the number of subnets ($CS_n$) because the space exploration of global routing increase, leveraging the algorithm to perform more re-executions. In summary, the C-SDN presents a better total latency for smaller systems, on average, lower than 256 PEs, while D-SDN outperform C-SDN for the larger system sizes.

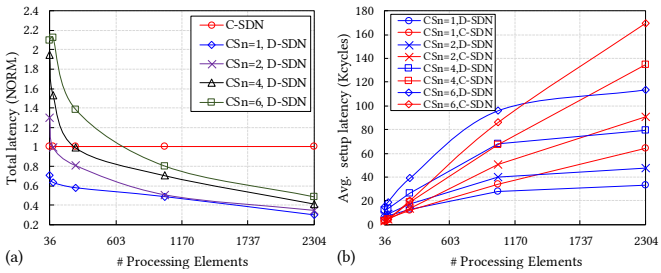


**Figure 4: Total and average setup latency for different system sizes and subnets ($CS_n$), for $L = 0.8$.**

While *total latency* reflects parallelism since the time is measured from the first path setup request until the last path setup ack, the graph of Figure 4(b) presents the *average setup latency* (*ASL*) per path. For system sizes below 1024 PEs on average, it is possible to observe that the D-SDN presents a higher *ASL*. The main reason explaining this result is that in D-SDN, even a short path can traverse two different clusters, requiring the execution of the distributed protocol. Note that the C-SDN *ASL* increases with the system size, due to the larger path search space. In the D-SDN the cluster size for 1,024 and 2,304 is the same, 16x16. Thus, the search space for local paths (80%) is the same, which explain the small difference between both results. This result unveils the improvements achieved due to the parallel execution of detailed routing and the routers' configuration phase.

Figure 5 evaluates the success rate, setup latency and path length, considering the 1024 PEs system, $CS_n = 4$, and $L = 0.8$ (other scenarios follows the same behavior).

Figure 5(a)(b) presents the *success rate* for D-SDN and C-SDN, respectively. The success rate is similar for both approaches. Both curves sustain a success rate equal to 100% up to 10% of *total_paths* for D-SDN and 11.3% for C-SDN. After such path saturation point, the number of established path reduces, reaching 23.7% for D-SDN and 22.76% for C-SDN for *total_paths* equal to 100%. As mentioned in the experimental setup, the *total_paths* is a worst-case scenario, with all possible paths in the NoC being requested.

Figure 5 (c)(d) depicts the *setup latency*, each dot corresponding to a path setup. Such graphs should be analyzed with the success rate graphs. In the beginning, with a success rate of 100% and few paths established, the setup latency is lower than 200K clock cycles. Near to the saturation point, the worst setup latency is achieved, with higher latencies observed in C-SDN (again, due to the larger path search space). After this point, both approaches fail to find some paths, which reduces the setup latency. The fact that it is not necessary to configure the routers when a path fails also contributes to reduce the setup latency. Figure 5(c) presents an interesting behavior, the division between global paths (top cluster of points) local path (bottom cluster of points). Such division occurs because the global paths have a setup latency overhead on average 7.3 times over local paths for this scenario (explained bellow).

Figure 5(e) presents the path length distribution. The x-axis corresponds to the path length, in hops, and the y-axis to the number of established paths (973 for D-SDN and 932 for C-SDN). The C-SDN applies a centralized shortest path algorithm (Hadlock), enabling

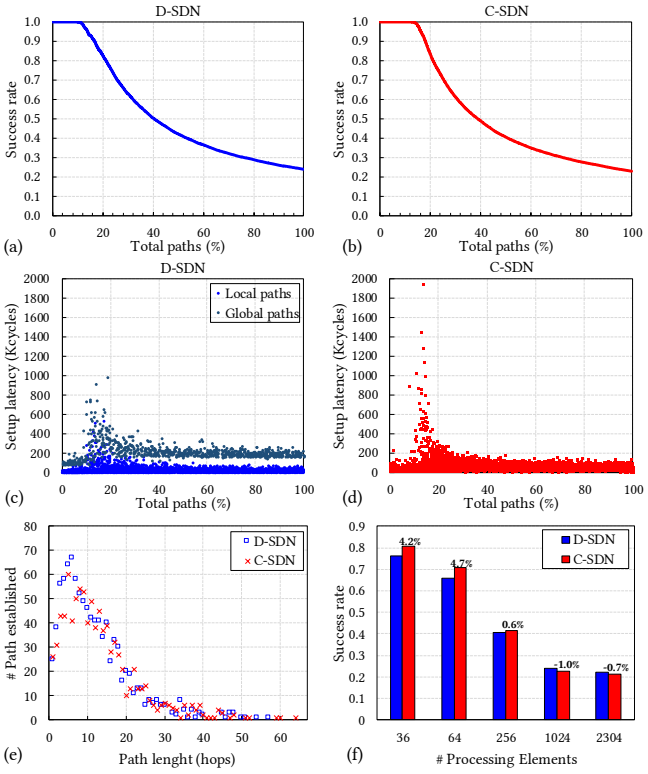


**Figure 5: (a,b) Success rate evaluation; (c,d) setup latency evaluation; (e) path length distribution; (f) success rate for different system's sizes, $CS_n = 4$, $L = 0.8$.**

this approach to find paths not reached by the distributed approach. According to the graph, C-SDN longest path has 64 hops, while D-SDN 57 hops. Note that, while in C-SDN the saturation point is reached later (13% better - Figure 5(a,b)), the C-SDN also has longer paths, creating more congestion and explaining its lowest final success rate. Thus, D-SDN has a higher number (10%) of established paths below 20 hops (843 against 767 for the C-SDN).

Figure 5(f) details the success rate for different systems sizes. The rate is slightly higher in C-SDN for system smaller than 256 PEs but lower in larger systems. The threshold of 256 PEs can also be observed for the other subnets. Below 256 PEs, the C-SDN outperforms D-SDN on average by 4.9%, and from 256 PEs, the D-SDN outperform C-SDN on average by 0.4%. The overall success rate difference between both approaches is 1.7%. Such difference allows us to conclude that the smaller path search space exploration does not negatively impact the path quality provided by D-SDN.

The experiment depicted in Figure 6 evaluates the impact of the cluster size. This experiment adopts the 48x48 system, $CS_n = 4$ and $L = 0.8$, with four different cluster sizes All four D-SDN scenarios achieve better performance than C-SDN, and the 16x16 case enjoys the lowest latency. The C-SDN latency is constant due to the absence of clusters. In general, we conclude that cluster size affects performance significantly. Hence, it is mandatory to identify the optimal configuration for a given system.

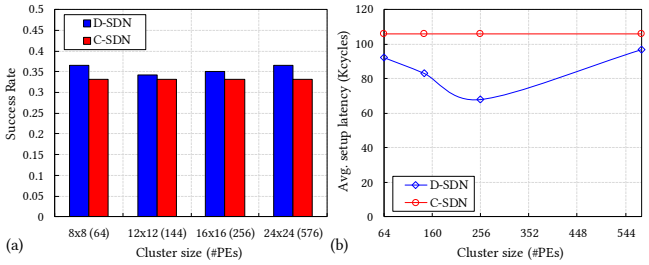


**Figure 6: Cluster size: (a) Success rate, (b) Avg. setup latency.**

## 6.3 Locality

This Section evaluates the locality effect in the C-SDN and D-SDN approaches. Figure 7 presents experimental results with $L$ varying from 0 (all paths being global) to 1 (all paths being local). This experiment adopts the 32x32(16x16) system, and $CS_n = 4$ (other $CS_n$ presents similar behaviour). Figure 7(a) addresses the total latency. As expected the D-SDN total latency reduces as the number of local paths increases (higher parallelism). D-SDN achieves the worst latency (on average 30%) for scenarios with L<0.7. This result is due to the higher number of global routing runs. For scenarios with local paths with L ≥ 0.7, the D-SDN outperforms C-SDN on average by 54%. We argue that scenarios presenting L ≥ 0.7 are those that correspond to real workloads due to the tasks mapping heuristics that favors local paths [19].

Figure 7(b) shows the success rate for this experiment. It is possible to observe that both approaches present a similar success rate, with a difference not higher than 1%. Also, the success rate is a function of the locality, because global paths lead to congestion, which reduces the number of available paths.

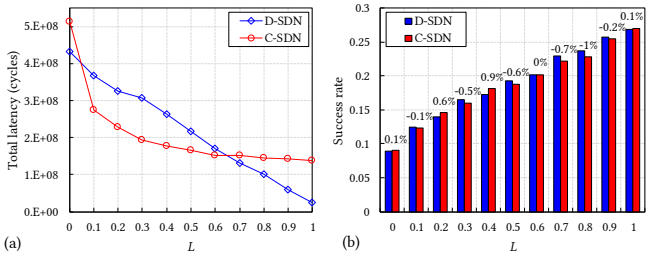


**Figure 7: Locality: (a) Total latency, (b) Success rate.**

## 6.4 D-SDN Protocol and Memory Overheads

Figure 8(a) presents the relationship between the setup latency to establish global and local paths (y-axis), with the same Manhattan distance, for different D-SDN system sizes (x-axis). According to the results, the distributed protocol imposes a significant overhead in small system sizes, e.g., the global path setup latency is 21.9 times higher for the smallest system and $CS_n = 4$. This graph represents the trade-off between the time to execute local and global paths. For small cluster sizes, the local path search is much faster than the search of global paths. The relationship reduces to 5–10 times by increasing the cluster size. This result follows the scalability evaluation related to the cluster size, where 16x16 clusters achieved the best results.

The second evaluated overhead is the memory footprint. The algorithm size (*text* segment) corresponds to ≈ 17 Kbytes for D-SDN and ≈ 7 Kbytes for C-SDN. Figure 8(b) presents the *data* memory requirement per controller. C-SDN presents a linear growth, while

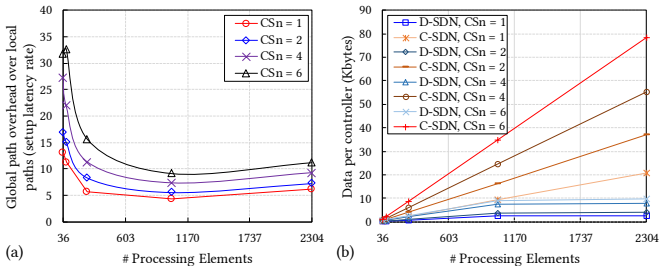


**Figure 8: (a) Overhead of global vs. local paths with same Manhattan distance. (b) Data requirements per controller.**

in D-SDN the memory footprint is a function of the cluster size, and not to the system size. Thus, the D-SDN memory footprint scales with the system size, while the C-SDN not.

## 6.5 Individual Setup Latency

This last experiment adopts a different data set from previous ones. The experiment consists in requesting only one path with its source and target addresses located at the opposite corners of the system, thus exploring the maximum Manhattan distance for a given path. The goal of this experiment is twofold: (*i*) evaluate the performance for a scenario without congestion; (*ii*) evaluate how the setup latency behaves increasing the number of clusters. The cluster size is equal to 8x8 for all scenarios, and only the system size changes.

Figure 9 compares the setup latency for D-SDN and C-SDN for this single path request. It is possible to observe that D-SDN always outperforms C-SDN. The improvement increases with larger systems regardless of the number of clusters, reaching up to 85.5% for 36 clusters (48x48(8x8)). The lower setup latency of D-SDN is due to the parallel phases of detailed routing and path configuration.

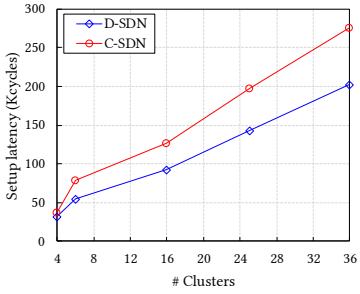


**Figure 9: Setup latency w.r.t cluster number.**

## ACKNOWLEDGMENTS

We acknowledge financial support by ÖAW Austria and CAPES/FAPERGS Brazil – Fernando Gehm Moraes supported by FAPERGS (17/2551-0001196-1 and 18/2551-0000501-0) and CNPq (302531/2016-5) and Marcelo Ruaro by CAPES/FAPERGS (88887.196173/2018-00).

## 7 CONCLUSION AND FUTURE WORKS

This work proposed and evaluated a distributed SDN architecture for the management of MCSoCs. The D-SDN has an inherent challenge that is the synchronization protocol. However, the cost of the protocol becomes relatively lower as the system grows. On average, D-SDN outperformed C-SDN in total latency in systems larger than 256 cores without loses in success rate. The goal of our proposal is to adopt SDN in systems with hundreds of PEs. Using larger systems, the results showed that the amount of memory for the controllers and the total latency for the establishment of paths scale with the size of the system. Also, the path quality, considering the path length and the success path establishment rate, does not suffer from the distributed approach, being slightly better than the C-SDN approach. Therefore, the proposed D-SDN approach advances the state-of-the-art related to a self-adaptive management of large NoC-based MCSoCs.

Future works are directed to develop a fault-tolerance protocol among controllers and providing QoS and security services for user's applications based on the distributed SDN architecture.

## REFERENCES

[1] A. Abousamra, A. Jones, and R. Melhem. 2013. Proactive circuit allocation in multiplane NoCs. In *DAC*. ACM, 35:1–35:10. https://doi.org/10.1145/2463209.2488778

[2] K. Berestizshevsky, G. Even, Y. Fais, and J. Ostrometzky. 2017. SDNoC: Software defined network on a chip. *Microprocessors and Microsystems* 50 (2017), 138–153. https://doi.org/10.1016/j.micpro.2017.03.005

[3] H. Chen and Y. Chang. 2009. Electronic Design Automation: Synthesis, Verification, and Test. Morgan Kaufmann Publishers/Elsevier, Burlington, MA, Chapter 12, 687–748.

[4] L. Cong, W. Wen, and W. Zhiying. 2014. A configurable, programmable and software-defined network on chip. In *WARTIA*. 813–816. https://doi.org/10.1109/WARTIA.2014.6976396

[5] N. Dutt, F. Kurdahi, R. Ernst, and A. Herkersdorf. 2016. Conquering MPSoC complexity with principles of a self-aware information processing factory. In *CODES+ISSS*. ACM, 37:1–37:4. https://doi.org/10.1145/2968456.2973275

[6] A. Ejaz and A. Jantsch. 2013. Costs and Benefits of Flexibility in Spatial Division Circuit Switched Networks-on-chip. In *NoCArc*. 41–46. https://doi.org/10.1145/2536522.2536526

[7] S. Ellinidou, G. Sharma, T. Rigas, T. Vanspouwen, O. Markowitch, and J. Dricot. 2019. SSPSoC: A Secure SDN-Based Protocol over MPSoC. *Security and Communication Networks* 2019, 4869167 (2019), 11 pages. https://doi.org/10.1155/2019/4869167

[8] R. Sandoval-Arechiga et al. 2016. Software Defined Networks-on-Chip for multi/many-core systems: A performance evaluation. In *ANCS*. 129–130. https://doi.org/10.1145/2881025.2889474

[9] F. Hadlock. 1977. A shortest path algorithm for grid graphs. *Networks* 7, 4 (1977), 323–334.

[10] A. Kostrzewa, S. Tobuschat, and R. Ernst. 2018. Self-Aware Network-on-Chip Control in Real-Time Systems. *IEEE Design Test* 35, 5 (2018), 19–2. https://doi.org/10.1109/MDAT.2017.2763598

[11] A. Leroy, D. Milojevic, D. Verkest, F. Robert, and F. Catthoor. 2008. Concepts and Implementation of Spatial Division Multiplexing for Guaranteed Throughput in Networks-on-Chip. *IEEE Trans. on Computers* 57, 9 (Sept. 2008), 1182–1195. https://doi.org/10.1109/TC.2008.82

[12] S. Liu, A. Jantsch, and Z. Lu. 2015. MultiCS: Circuit switched NoC with multiple sub-networks and sub-channels. *Journal of Systems Architecture* 61, 9 (Oct. 2015), 423–434. https://doi.org/10.1016/j.sysarc.2015.07.013

[13] G. Michelogiannakis and W. J. Dally. 2013. Elastic-Buffer Flow Control for On-Chip Networks. *IEEE Trans. on Computers* 62, 2 (Feb. 2013), 295–309. https://doi.org/10.1109/HPCA.2009.4798250

[14] Y. Oktian, S. Lee, H. Lee, and J. Lam. 2017. Distributed SDN controller system: A survey on design choice. *Computer Networks* 121 (2017), 100–111. https://doi.org/10.1016/j.comnet.2017.04.038

[15] S. Rhoads. 2016. Plasma - most MIPS I(TM). (2016). https://opencores.org/projects/plasma

[16] M. Ruaro, H. Medina, A. Amory, and F. Moraes. 2018. Software-Defined Networking Architecture for NoC-based Many-Cores. In *ISCAS*. IEEE, 385–390. https://doi.org/10.1145/2881025.2889474

[17] M. Ruaro, H. M. Medina, and F. G. Moraes. 2017. SDN-Based Circuit-Switching for Many-Cores. In *ISVLSI*. IEEE, 385–390. https://doi.org/10.1109/ISVLSI.2017.74

[18] A. Scionti, S. Mazumdar, and A. Portero. 2018. Towards a Scalable Software Defined Network-on-Chip for Next Generation Cloud. *Sensors* 18, 7 (2018), 1–24. https://doi.org/10.3390/s18072330

[19] A. Singh, P. Dziurzanskiand, H. Mendis, and L. Indrusiak. 2017. A Survey and Comparative Study of Hard and Soft Real-Time Dynamic Resource Allocation Strategies for Multi-/Many-Core Systems. *ACM Comput. Surv.* 50, 2 (April 2017), 24:1–24:40. https://doi.org/10.1145/3057267

[20] R. Stefan, A. Neja, and K. Goossens. 2012. Online Allocation for Contention-free-routing NoCs. In *INA-OCMC*. ACM, 13–16. https://doi.org/10.1145/2107763.2107767

[21] Y. J. Yoon, N. Concer, M. Petracca, and L. P. Carloni. 2013. Virtual Channels and Multiple Physical Networks: Two Alternatives to Improve NoC Performance. *IEEE Trans. on CAD of ICs and Systems* 32, 12 (Dec. 2013), 1906–1919. https://doi.org/10.1109/TCAD.2013.2276399