The Journal of Supercomputing (2022) 78:12310–12343
https://doi.org/10.1007/s11227-022-04365-y

Providing quality of service in omni-path networks

Javier Cano-Cano[1] · Francisco J. Andújar[2] · Francisco J. Alfaro-Cortés[1] · José L. Sánchez[1] · Gaspar Mora[3]

Accepted: 5 February 2022 / Published online: 2 March 2022

Abstract
New hierarchical crossbar switch architectures, such as Omni-Path (OPA) and Cray X2, have appeared to improve packet latency, reduce overall cost and increase fault tolerance of the high-performance interconnection networks in supercomputing and data center systems. These and other interconnect technologies (Infiniband or 40/100 Gigabit Ethernet) include support to provide quality of service (QoS) to the applications. In this paper, we show how this QoS support can be enabled to achieve bandwidth and/or latency differentiation in Omni-Path interconnection networks, as a representative case of hierarchical switches. To do that, three different table-based schedulers are used. We include the description of these schedulers and a comparative study by using the results obtained when we evaluate them with Hiperion, a simulation tool that implements an OPA model.

Keywords Quality of Service (QoS) · Scheduling algorithms · Interconnection networks · Omni-Path (OPA) · Simulation · Hierarchical-crossbar-switch architecture

1 Introduction

In the last decades, there has been a constant advancement in high-speed interconnection network technologies. This development has been fueled by the growth of the supercomputing and data center services, where the interconnection network is usually the limiting factor (bottleneck), i.e., the central element upon which the performance of the whole system relies. Therefore, it is critical to keep improving the overall interconnection network performance. This is achieved by the introduction of continuous improvements in the physical elements of the network (links,

Javier Cano-Cano and Francisco J. Andújar authors contributed equally to this work.

✉ Javier Cano-Cano
javier.cano@uclm.es

Extended author information available on the last page of the article

switches, NICs, etc.) and the techniques they implement (routing algorithms, congestion avoidance mechanisms, etc.).

Moreover, the total bandwidth per switch has increased due to a combination of higher pin density and faster signaling rates. As the total bandwidth increases, switch designers face two possibilities to exploit this bandwidth: to build switches with a high number of *thin* ports (high-radix switches) or to build switches with a low number of *fat* ports (low-radix switches). The current trend is to use high-radix switches [1–3] as they present advantages such as: the final packet latency is reduced, the interconnection network overall cost is reduced, the wiring is also reduced, the power dissipated by the network decreases, the network fault tolerance is increased and a distributed packet arbitration process may be applied. However, high-radix switches also face some problems: the cost and efficiency balance is not easy to be maintained, increased buffer requirements drive cost up, the virtual channel allocation process becomes more complex, among others. To address some of these issues, fully-buffered crossbar and hierarchical crossbar switch architectures have been introduced. Fully-buffered crossbar switches require a huge silicon area when the radix increases, making them unfeasible due to the associated costs. Hierarchical crossbar switch architectures overcome that drawback while achieving a very high port count. Some high performance devices such as YARC [1], Omni-Path [4] and Slingshot's Rosetta switches [5] use a hierarchical crossbar architecture to achieve high-radix interconnection devices.

Omni-Path (OPA) emerged with the aim of occupying a space in the select group of high-performance interconnection network technologies, such as InfiniBand (IB) [6] or 40/100 Gigabit Ethernet (GE) [7]. These interconnection network technologies have been competing to achieve better performance and market share than others. In terms of market share, since its introduction in the most powerful computers list TOP500 [8], OPA has ranged from 1.6 to 10%. Considering the 100 most powerful computers on the list, OPA have reached up to 13%.

Current interconnection networks carry not only traffic of applications such as backup or file transfer protocols, which does not require service differences, but also traffic from others like real-time protocols [9], MPI communications or traffic from users with different privilege levels in the system [10]. Therefore, QoS has become the focus of much discussion and research during the last decades [11, 12]. A sign of this interest is the inclusion of support aimed to provide QoS on interconnection networks such as GE, IB and also OPA.

One of the most important QoS mechanisms is the scheduling algorithm [13, 14]. High performance interconnection networks usually use packet-switching as switching technique. This kind of networks can carry packets from different applications, users and flows, interacting with each other in every interconnection network element. Without any scheduling policy, packets from different traffic flows[1] use as many resources as they need and, in the worst scenario, a single flow may consume all the system resources causing starvation on others. In such way, users may experience a poor system performance even if the system is not overloaded. Therefore, the scheduling algorithm is a crucial element to provide QoS.

[1] In this paper, we will use the term *traffic flow*, *packet flow* or just *flow* for referring to a sequence of packets with similar characteristics (delay and/or bandwidth requirements).

Scheduling algorithms orchestrate when packets from different flows will be delivered to satisfy the specified end-to-end delay and/or bandwidth requirements. However, in the context of high performance interconnection networks, the scheduling algorithms have to be as simple as possible in terms of computational and implementation complexity [15]. The scheduling algorithm latency must be smaller than the average packet transmission time for the obvious reason that the system will expend more time choosing packets to deliver than delivering packets, therefore degrading system performance. Also, low complexity is required because the scheduling algorithm is typically implemented in hardware, and thus a very complex scheduler will be more complex. Therefore, the scheduling algorithm design process involves some trade-offs.

A well-known scheduling algorithm family is "sorted-priority" schedulers, which use a global variable, called *virtual time*, that keeps track of the server's progress and it is updated when a packet is received or transmitted. Each packet has a timestamp tag, computed as a function of the *virtual time*. These schedulers offer very good fairness and low latency [16], but they are computationally complex due to the tag calculation and the sorting process.

Table-based schedulers are another well-known family of scheduling algorithms. This approach has been used in high performance interconnection network technologies such as OPA [17] and IB [6]. Table-based schedulers offer good latency and bandwidth performance with a low computational complexity.

Simulation is one of the most common approach to explore new techniques in high-performance interconnection networks. As stated before, the interest of hierarchical switch architectures is growing. However, as far as we know, no hierarchical switch simulation model is available. Therefore, we have decided to develop a hierarchical switch simulation model based on OPA. Moreover, the QoS support of this kind of networks is also important as explained before and it has not been addressed in any study yet. For these reasons, we have decided to perform a comparative study between some known output scheduling algorithms adapted to a popular hierarchical switch architecture such as OPA. This study led us to find out if known output scheduling algorithms are suitable and to know the adaptation process for hierarchical switch architectures.

In this paper, we will focus on OPA because it is a good example of hierarchical crossbar switch architectures and that has not been, as far as we know, largely studied in terms of QoS provision. We present a novel OPA simulation model, which is the first hierarchical switch simulation model available. Three scheduling algorithms adapted to the OPA technology are presented here. These algorithms have been implemented in our Hiperion simulator [18], allowing us to compare the performance and to find which scheduling algorithm is more adequate for full-scale OPA-based systems. The first and simplest scheduling algorithm is a round-robin scheduler, which is the baseline schema used for comparison purposes. The second is a table-based scheduling algorithm, which we have called Simple Bandwidth Table (SBT). This scheduler offers bandwidth differences but is not able to provide latency differences. Finally, for the third algorithm, we have adapted the Deficit Table Scheduler (DTable) [19] to the OPA technology. This scheduler is more complex

than the two previous ones although is able to provide bandwidth and latency differences with a reasonable computational and implementation complexity [20].

The structure of the paper is as follows: Sect. 2 reviews the OPA architecture and our OPA-based simulation model. Section 3 explains the main output scheduling algorithms proposed. Section 4 shows the results obtained evaluating bandwidth and latency differentiation, and, finally, Sect. 5 presents some conclusions.

2 The OPA architecture

As stated in Sect. 1, OPA rapidly grew in popularity after it was firstly introduced. The OPA architecture has some elements such as a hierarchical internal crossbar and multiple QoS tables that makes it different from the most popular high performance interconnection network architectures like IB and GE. This allows enabling QoS techniques that are simply not feasible in the rest of high performance interconnection network architectures. And in order to design, explore, and evaluate the performance of these possibilities, testing tools are required such as simulation programs, mathematical models, etc. OPA was initially developed by Intel until 2019, when all the OPA technology IP was transferred to Cornelis Networks, a new company that is continuing the support and development of OPA products [21, 22].

As described in Sect. 1, we have chosen OPA just as an example, but the findings and conclusions could be adopted to other similar architectures such as YARC [1], or Slingshot's Rosetta [5] switches.

We have collected all relevant information about the OPA architecture and QoS support, and we have developed the simulation tool Hiperion (HIgh PERformance InterconnectiOn Network), which includes an OPA simulation model [18]. Hiperion is an open-source simulation tool available for researchers and companies and includes multiple useful mechanisms to perform many comparative studies. The simulation model includes all the main features for simulating the movement of packets between source and destination using several configurable QoS strategies. These QoS strategies will be analyzed and compared.

2.1 OPA support for QoS

The OPA architecture offers support to provide QoS to applications, flows, packets, etc. According to [17], support is given through the following elements:

- Virtual Lanes (VLs) provide dedicated receive buffer space for incoming packets at switch ports. VLs are also used for avoiding routing deadlocks. The Intel Omni-Path architecture supports up to 32 VLs.
- Service Channels (SCs) differentiate packets from different Service Levels. The SC is the only QoS identifier stored in the packet header. Each SC is mapped to a single VL, but a VL can be shared by multiple SCs. SCs are used for avoiding topology deadlocks and avoiding head of line blocking between different traffic

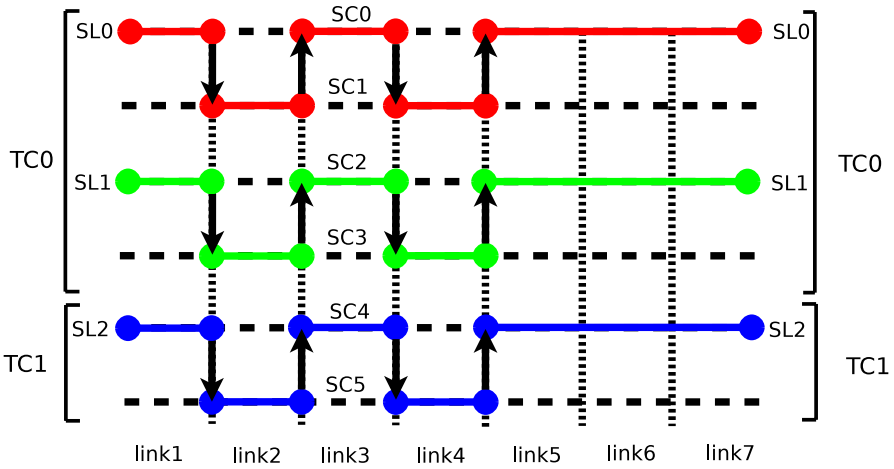

Fig. 1 An example of usage of TCs, SLs and SCs [17]

classes. The Intel Omni-Path architecture supports up to 32 Service Channels, however SC15 is dedicated to in-band fabric management.

- Service Levels (SLs) are a group of SCs. An SL may span multiple SCs, but an SC is only assigned to one SL. SLs are used for separating high priority packets from lower priority packets belonging to the same application or Transport Layer, avoiding protocol deadlocks, etc. The Intel Omni-Path architecture supports up to 32 SLs.
- Traffic Classes (TCs) represent a group of SLs aimed to distinguish applications' traffic. A TC may span multiples SLs, but each SL is only assigned to one TC. The Intel Omni-Path architecture supports up to 32 TCs.
- A vFabric is a set of ports and one or more application protocols. For each vFabric, a set of QoS policies are applied. A given vFabric is associated with a TC for QoS and associated with a partition for security.

SLs are mapped to SCs via the *SL2SC* tables and SCs are mapped to SLs via *SC2SL* tables, depending on whatever the packets are sent or received, respectively. Each SC carries traffic of a single SL in a single TC, and the Fabric Manager (FM) fulfills *SC2VL* and *VL2SC* tables, determining how SCs are mapped onto VLs at each port and vice-versa. The FM is also responsible of: discovering the fabric topology, provisioning the fabric components with identifiers, formulating and provisioning routing tables, monitoring utilization, performance and error rates and fulfilling arbitration tables.

OPA includes also QoS mechanisms such as *VLArbitration Algorithm* and *preemption Tables*. However, there is not much information about how these mechanisms work.

Figure 1 shows an example of the use of TCs, SLs, and SCs across the paths followed by three traffic flows (red, green and blue) in an OPA network. The different links crossed by these packets are ordered from 1 to 7. In this example, we assume

the use of two TCs (TC0 and TC1), three SLs (SL0, SL1 and SL2) and six SCs (SC0, SC1, SC2, SC3, SC4 and SC5). Moreover, each SL is assigned with two SCs, which, in turn, are mapped to two VLs. TC0 (i.e., traffic flows red and green) is used for example for a request/response high level communication library such as Partitioned Global Address Space protocol (PGAS)[2]. Let's suppose TC0 is assigned with SL0 (red traffic flow) and SL1 (green traffic flow), SL0 is mapped to SC0 and SC1, and SL1 is mapped to SC2 and SC3. On the other hand, TC1 is used, for instance, for storage communications. It is assigned with SL2, and SL2 is mapped to SC4 and SC5. The main goal of assigning a pair of SCs for each SL is topology deadlock avoidance, as it happens normally in torus topologies, while the SLs of TC0 are used for avoiding protocol deadlocks. As we can see in the figure, packets can change of SC link by link; however, the SL and TC are always consistent end-to-end [17].

2.2 OPA simulation model

We have carried out the study presented in this work using simulation for being one of the most popular technique to evaluate, verify and validate the behavior and performance of high performance interconnection networks. There are multiple simulation tools such as Garnet [23], xSim [24], etc. focused on on-chip networks. These simulators allow full-systems simulations, feasible for on-chip networks, due to the small network sizes. However, when the network grows to hundreds of elements, the computational resources needed make full-systems simulation unapproachable. Moreover, the characteristics of the off-chip and on-chip traffic are disparate. There are also multiple off-chip simulation tools such as CODES [25], SST [26], etc. However, these simulation tools do not have support for any hierarchical crossbar switch architecture with QoS. Therefore, we have proposed an OPA-based simulation model and a simulation tool called Hiperion based on the available public information [4, 17]. It is based on previous tools that have been used for years in our research group, and with multiple publications behind them [27, 28]. Our simulator Hiperion gives us a deep knowledge of its operation and a wide flexibility regarding the techniques that can be implemented and its interoperability.

Hiperion is a discrete-event based network simulator, which includes an OPA simulation model that mimics the behavior of main OPA elements, such as switches, links and network interfaces. The simulator main goal is to perform comparative studies tuning a large range of parameters such as queue sizes, topology, routing, packet sizes, scheduling algorithms, etc. The simulator is capable of running simulations using a wide variety of synthetic traffic types such as random, uniform, bit-reversal, bit-complement, etc., and MPI applications using the VEF trace framework [29]. Performance and scalability of the interconnection network are evaluated using several metrics: throughput, end-to-end latency, network latency, etc.

Figure 2 shows a detailed scheme of a 48-port OPA-based switch, which has been implemented into Hiperion. The OPA switch model assumes that each port delivers one flit per cycle. Hence, the bandwidth is defined based on the clock rate and the

[2] Partitioned Global Address Space languages combine the programming convenience of shared memory with the locality and performance control of message passing.

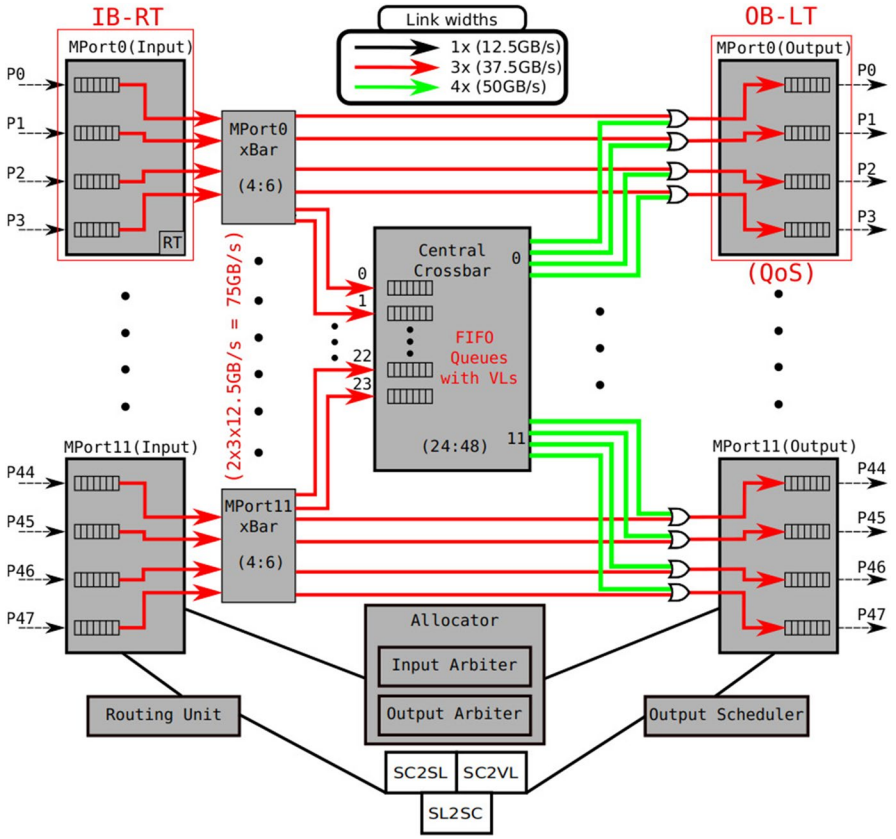

Fig. 2 Diagram of the modeled OPA switch of 48 ports. For clarity, MPorts are unfolded in Input and Output buffers

flit size. However, the OPA hierarchical architecture has a large range of internal links with different bandwidths [17]. The OPA model defines the input/output port bandwidth (12.5 GB/s) as a reference, thereby an *x3* internal link has a speed-up of 3 and so it may deliver 3 flits/cycle. The number of input and output links is represented as INPORTS:OUTPORTS in the crossbar elements, i.e. MPort xBars and Central Crossbar. For instance, in Fig. 2, the MPort0 xBar has 4 input links and 6 output links (4:6), and the Central Crossbar has 24 input links and 48 output links (24:48). The OPA model shown in this figure includes the following elements:

- Input buffers: They store the flits from the input ports. There is one input buffer per input port.
- Routing unit: There is one routing unit per input buffer.
- MPort Xbar: This crossbar has 4 input links, one per input buffer; and 6 output links: 4 links for the output buffers and 2 links for the Central Crossbar. Note that the 75 GB/s link to the Central Crossbar is represented in

this model as two *x3* links, i.e, they may deliver 3 flits/cycle, resulting on $2\ Links \times 3 \times 12.5\ GB/s = 75\ GB/s$.

- Output buffers: They store the flits of the output ports. There is one output buffer per output port.
- Input arbiter: Given an input buffer, it selects the virtual line (VL) that participates in the second allocator phase. The more VLs, the bigger the arbiter is.
- Output arbiter: Given an output buffer, it chooses which input port will transmit flits. A flit can arrive at this output buffer coming from an input buffer or from the Central Crossbar.
- Output scheduler: Given an output port, it chooses which VL will transmit flits to the neighbor switch. It provides QoS to the OPA switch.

As stated before, Hiperion is a discrete-event based simulation tool for modeling high-performance interconnection networks. Hiperion defines and implements the following discrete events:

- IB (Input Buffering): A flit arrives at an input port and is stored in the corresponding queue, depending on the VL. Each input buffer can receive 1 flit/cycle. If that flit is a packet header flit, it is set as *RT-ready*, and the routing event is called to determine the flit output port. In other case, the flit is set as *X-ready*, it is stored on the input buffer and it waits to be moved to the appropriate buffer in a Xbar event. When the output port is connected to the same Mport than the input port, the flit is moved to an output buffer. In other case, the flit is moved to a Central Crossbar buffer. For example, let's suppose an OPA switch with 48 ports and 4 ports per MPort (Fig. 2). If a flit needs to travel from the input port 0 to the output port 5, the input port belongs to the MPort 0, which contains input ports from 0 to 3, while the output port belongs to the MPort 1, which contains output ports from 4 to 7. Therefore, the flit must cross the Central Crossbar to arrive at MPort 1.
- RT (RouTing): Routes a packet and determines its output port when the packet header flit is tagged as *RT-ready*. After that, the header flit is tagged as *VA-SA-ready* and the input buffer storing this flit can be chosen in the first phase of the allocation event. The RT event is only applied to header flits. Non-header flits always follow the header flit, since OPA architecture implements virtual-cut though as switching technique [17]. The routing function is configurable and must be according to the simulated topology.
- VA-SA (Virtual Allocator and Switch Allocator): Performs the allocation using a two-stage allocator:
  - Virtual Allocator: Each input arbiter chooses a VL, only if its input buffer contains at least one *VA-SA-ready* header flit. The winning VL will be allowed to deliver a packet. Since the Central Crossbar links have VLs as well, the virtual allocator is also performed on the input buffers of the Central Crossbar.
  - Switch Allocator: Each output arbiter chooses an input buffer with a winning VL. The winning input buffers will be allowed to move a packet to an output buffer or to a Central Crossbar buffer, depending on the destination MPort.

Buffers allowed to transmit tag the top header flit as *X-ready*. A central buffer has to arbitrate between the 4 input buffers which are connected to its MPort. An output buffer has to arbitrate between the 24 Central Crossbar buffers and its 4 MPort buffers.

Currently, both virtual and switch allocators implement round-robin arbiters. However, we are developing more sophisticated arbiters able to provide applications with QoS.

- X (Xbar): Once the allocation is performed, the winning input and Central Crossbar buffers transmit the first packet of their winning VLs to the appropriate output buffer or Central Crossbar buffer. If a packet is moved from an input buffer to a Central Crossbar buffer, the header flit is tagged again as *VA-SA-ready* in order to perform a VA-SA event from Central Crossbar buffers to output buffers. If the packet reaches an output buffer, their flits are tagged as *OB-ready*. The bandwidth depends on the input/output pair. MPorts xbar can deliver 3 flits/cycle regardless of the destination buffer, while the Central Crossbar xbar can deliver 4 flits/cycle.
- OB (Output Buffering): Each output scheduler chooses which VL will send flits to the neighbor switch. The scheduler selects a VL with *OB-ready* packets and enough credits to transmit at least one packet. When the last flit of the packet is transmitted, (i.e., the tail flit), the output scheduler releases the winning VL and selects a new VL. Each output port can send 1 flit/cycle. At this point, QoS and packet preemption can be applied. Currently, three scheduling algorithms have been implemented (Sect. 3), but packet preemption is not implemented yet.

VL buffer storage space is dynamically managed, i.e. the buffer space is shared by all the VLs. The buffer storage space is divided according to the traffic requirements, ensuring a minimum and a maximum amount of flits per VL. This prevents a single VL from taking up all the flits in the buffer, causing starvation in the remaining VLs. This dynamical buffer storage management strategy provides more flexibility than static buffers [30].

The main QoS OPA support such as SCs, SLs, VLs, *SL2SC* and *SC2VL* tables, etc., have also been implemented in Hiperion. There are some additional mechanisms that have been implemented not directly related with QoS. However, they elements are crucial in some cases. Some of them are: variable Maximum Transfer Units (MTUs) per SL, message generation based on variable MTU sizes, variable injection rate definition per SL, among others. The goal of these QoS mechanisms and the simulation model implemented is to develop, test and compare different QoS scheduling algorithms.

3 Scheduling algorithms

The main goal of scheduling algorithms is to determine when packets from different SLs are delivered in order to satisfy the specified end-to-end latency and bandwidth requirements. Not all scheduling algorithms are capable of satisfy both

Table 1 Round-robin table QoS algorithm sample

SL	Weight
0	50
1	50

requirements, some are only able to fulfill bandwidth requirements. Moreover, in the context of high-performance interconnection networks, scheduling algorithms must meet two main characteristics: low computational complexity (the scheduler latency must be smaller than the average packet latency) and low implementation complexity (the scheduling algorithm is typically implemented in hardware and a high implementation complexity implies a large silicon area).

In this section we detail three scheduling algorithm proposals adapted to the context of hierarchical-crossbar-switch architectures, specifically, to the OPA architecture.

3.1 The round-robin output scheduler

The round-robin output scheduler is the simplest output scheduler. The main goal of a round-robin output scheduler is to distribute the total bandwidth among all SLs. The bandwidth that each SL will obtain is $\frac{1}{NumSLs}$, where *numSLs* is the total number of SLs. This scheduler could be based on an arbitration table or on an hardware implemented algorithm. Although both approaches are feasible as long as the bandwidth is properly distribute, we have chosen the arbitration table because the other algorithms presented in this work are also based on arbitration tables, as we will explain in Sects. 3.2 and 3.3. In fact, this scheduler can be implemented using the SBT scheduler, equally distributing all the bandwidth among all the SLs. Table 1 shows an example of SBT scheduling table configured to work in a round-robin way. The initial entry weights are not relevant as long as for a round-robin algorithm they are equal on each table entry. For this reason, the details about how the round-robin scheduler works can be found in Sect. 3.2. Note that the round-robin output scheduler does not provide any QoS differences. We have considered this scheduler in order to establish a comparison baseline.

3.2 The simple bandwidth table mechanism

Simple Bandwidth Table (SBT) is a table-based scheduler. It is one of the simplest techniques to provide bandwidth differences in a high performance interconnection network.

SBT scheduler is based on an arbitration table per output port with as many entries as SLs are considered. Each table entry is assigned to one SL and the entries store an entry weight. This weight represents how many packets an SL may deliver. Every time that an SL delivers a packet, the entry weight is decremented until it is equal to zero. Table 2 shows an example considering two SL, where SL0 has a weight of 55 and SL1 has a weight of 45. If in a given output port, SL0 delivers 3

Table 2 Simple Bandwidth Table QoS algorithm sample

SL	Weight
0	55
1	45

packets, the remaining weight will be 52. Therefore, the fraction of the total bandwidth ϕ_i assigned to the SLi is

$$\phi_i = \frac{\text{weight}_i}{\sum_{j=0}^{N-1} \text{weight}_j},$$

where N is the total number of SLs and *weight* is the entry weight assigned to each SL. In the SBT arbitration table (Table 2), SL0 will get 55% of the total bandwidth and SL1 will get 45% of the bandwidth. In our proposal, for the sake of simplicity, $\sum_{j=0}^{N-1} weight_j$ must be equal to 100. In this way, the bandwidth percentage of each SL can be easily obtained.

The arbitration table is cycled through in a round-robin way when the entry weight is equal to zero. The table is also cycled when the SL in transmission becomes "inactive", i.e. the SL has no packets to transmit[3]. When the sum of all entry weights is zero, the initial entry weights are restored. Note that an SL can only transmit when its weight is greater than zero. However, there is an exception: when an active SL does not have enough weight left but it is the only active SL, the transmission of packets is allowed. This exception avoids packet starvation and wasting the link bandwidth.

Finally, realize that the bandwidth is distributed by SL, not VL. Otherwise, if some SLs have a different number of VLs assigned than others, the total bandwidth cannot be distributed correctly between the SLs. Let's suppose two SLs and three VLs in the network. SL0 can use two VLs and SL1 can use the remaining VL. We want to distribute 50% of bandwidth to each SL and we assign the same weight to each VL. Then, SL0 will get $\frac{2}{3}$ of the total bandwidth, while SL1 will only get $\frac{1}{3}$. It would also be possible to distribute traffic between SLs by VLs instead of SLs, but this complicates the table configuration and offers no added benefit.

[3] From now on, we will refer as an inactive SL to that SL that has available weight in the arbitration table but does not have packets to transmit.

Algorithm 1 Generic implementation of SBT on OPA.

```
1:  switch: Device ID
2:  pOut: Switch output port ID
3:  Output: VL ID
4:  procedure SBT(switch, pOut)
5:      vl_w_flits_no_weight ← -1
6:      empty_entries ← 0
7:      for i ← 0, NumVLs-1 do
8:          vl ← (last_vl_selec + i) % NumVLs
9:          flit ← first_flit(vl)
10:         SL ← SC2SL[flit.SC]
11:         if is_active(SL) then
12:             if arbiter_table[pOut][SL] > 0 then
13:                 last_vl_selec ← vl
14:                 arbiter_table[pOut][SL]–
15:                 return vl
16:             else
17:                 vl_w_flits_no_weight ← vl
18:                 empty_entries++
19:             end if
20:         end if
21:     end for
22:     if empty_entries = total_entries then
23:         resetQoSTable(switch, pOut)
24:     end if
25:     if vl_w_flits_no_weight ≠ -1 then
26:         last_vl_selec ← vl_w_flits_no_weight
27:         return vl_w_flits_no_weight
28:     end if
29:     return ALLOCATION_FAILED                    ▷ No SL is active
30: end procedure
```

Algorithm 1 shows the generic mechanism of the SBT scheduler on every port. Note that the *first_flit*() function allows to extract the first flit from a given VL queue and the *is_active*() function determines if an SL is active (i.e. it has packets to transmit) or not. Since OPA uses virtual cut-through as switching technique, these algorithms are only applied to header flits, so that body and tail flits will always follow the header flit at one flit distance. Furthermore, the SC identifier is the only QoS identifier stored in packets [17]. For this reason, *SC2SL* tables are used to get the SL identifier from the SC packet identifier.

The main advantages of SBT are its capacity to provide bandwidth differences and to have a very low computational and implementation complexity (Sect. 3.2.1). However, SBT is not able of providing latency differences, which could be crucial in many scenarios.

3.2.1 Complexity considerations

In terms of computational complexity, SBT is quite simple. In this case, arbitration tables have as many table entries as SLs. OPA supports up to 32 SLs according to

Table 3 Arbitration table implementation with one table per switch

SL	Weight	Port 0	Port 1	.	Port ***p - 1***
0	x_0	$x_0 - \alpha_{0,0}$	$x_0 - \alpha_{0,1}$	.	$x_0 - \alpha_{0,p-1}$
1	x_1	$x_1 - \alpha_{1,0}$	$x_1 - \alpha_{1,1}$	.	$x_1 - \alpha_{1,p-1}$
2	x_2	$x_2 - \alpha_{2,0}$	$x_2 - \alpha_{2,1}$	.	$x_2 - \alpha_{2,p-1}$
.	.	.	.	.	.
$N-1$	x_{N-1}	$x_{N-1} - \alpha_{N-1,0}$	$x_{N-1} - \alpha_{N-1,1}$	.	$x_{N-1,p-1} - \alpha_{N-1,p-1}$

[17]. Hence, in the worst case, if all table entries have to be looked over in order to find the next active SL, just 32 table entries will be skipped.

One of the most computationally complex tasks in Algorithm 1 is the *is_active()* function. However, the optimization strategy suggested in [20] may be used in order to keep the complexity low. Regarding the implementation complexity, considering an arbitration table per output port would require a large silicon area on hardware implementations. Therefore, instead of keeping a table per output port, a single table per switch with the structure shown in Table 3 may be used. The arbitration table has as many columns as output ports (p) plus 2 extra columns ($p + 2$) and as many rows as SLs N. The first two columns show SL_i identifiers and the associated weight x_i to the SL_i. The other columns represent the remaining SL weights $x_i - \alpha_{i,j}$ for each output port j. Every output port row is populated with the associated weight to each SL. When $\sum_{i=0}^{N-1} x_i = 0$ in a given column, the values from the Weight column are copied to the column of that port and thus the port will be allowed again to deliver packets.

3.3 The DTable scheduling mechanism

As explained in Sect. 3.2, SBT is not able to provide latency differences. Furthermore, SBT has other problems that we will discuss in Sect. 4. Therefore, we implemented, adapted and tested the DTable scheduler [31] on our OPA-based simulation model.

The DTable scheduler is based on an arbitration table with an structure similar to SBT arbitration tables: a column for an SL identifier and another column for an associated weight for each table entry and SL. However, there is an important difference between SBT and DTable arbitration tables: SBT arbitration tables have as many table entries as SLs whilst DTable arbitration tables have a greater arbitrary number of table entries, e.g. 32, 64, 128, etc. This difference is used to provide latency differences on SLs. The number of table entries and the maximum distance between any pair of consecutive table entries assigned to the same SL allow to control the SL latency [32]. Note that now each SL can have multiple table entries, and therefore, the bandwidth ϕ_i assigned to SLi is

$$\phi_i = \frac{\sum_{j=0}^{J-1} \text{weight}_j}{\sum_{k=0}^{N-1} \text{weight}_k},$$

where J is the set of table entries assigned to SLi and *weight* is the entry weight assigned to the table entry. Moreover, each SL has assigned a *deficit counter* initially set to 0. The deficit counters represent the weight that the scheduler owes to the SLs. The purpose of this counter is explained further on.

When scheduling is needed, arbitration tables are cycled through sequentially in a round-robin way until an active SL is found. The DTable scheduler has also an *accumulated weight* counter which is equal to the sum of the selected table entry weight and the SL deficit counter. The scheduler will deliver as many packets from the selected SL as the accumulated weight allows. The accumulated weight is decremented when packets are transmitted.

There are two possibilities that make the scheduler to select the next active table entry:

1. The SL becomes inactive. In this case the remaining accumulated weight is discarded and the deficit counter is set to zero.
2. The accumulated weight becomes smaller than the size of the packet at the head of the queue. In this case the accumulated weight is saved in the deficit counter.

Algorithm 2 Generic implementation of DTable on OPA.

```
1:  Output: VL ID
2:  procedure DTABLE()
3:      entry ← -1
4:      SL ← -1
5:      VL ← -1
6:      if pending_pkts ≠ 0 then ▷ The arbiter is called only when the entire
    MTU is transmitted
7:          exit(-1)
8:      end if
9:      SL ← Last SL selected by this output port
10:     if !is_active(SL) then                    ▷ The selected flow is not active
11:         accumulated_weight ← 0
12:         deficit_counter[SL] ← 0
13:         SL ← -1                  ▷ Indicates the arbiter has not SL selected
14:     else if SL_MTU[SL] > accumulated_weight then
15:         deficit_counter[SL] ← accumulated_weight
16:         SL ← -1
17:     end if
18:     if SL = -1 then                ▷ Not SL selected, select one active SL
19:         entry ← Next table entry assigned to an active SL
20:         if entry = -1 then       ▷ The arbiter has not been able to find an
    active SL
21:             return ALLOCATION_FAILED  ▷ This cycle the output port
    will not deliver any packet
22:         end if
23:         SL ← entry.SL
24:         accumulated_weight ← deficit_counter[SL] + entry.weight
25:     end if
26:     accumulated_weight ← accumulated_weight − SL_MTU[SL]
27:     pending_pkts ← SL_MTU[SL]
28:     return entry.VL
29: end procedure
```

Algorithm 2 shows a generic DTable scheduler. When the scheduler gets the *"Next table entry assigned to an active SL"* (line 19) the arbitration table is cycled through in a round-robin way until an active SL is found. The function returns the entry identifier and a VL associated to the selected SL. As stated in Sect. 2.1, an SL may span multiple SCs. In that case, the function arbiters between the SCs belonging to the same SL in a round-robin way, and it selects the VL through the *SC2SL* tables. For instance, in a given configuration SC0, SC1 and SC2 have been associated to SL0 as well as VL0, VL1 and VL2 to SC0, SC1 and SC2, respectively. The first time that SL0 is allowed to deliver packets, it will deliver packets from SC0 and VL0, the second time SL0 will deliver packets from SC1 and VL1, etc. Obviously, other SC selection strategy can be applied, such as dividing the accumulated weight among SCs of the same SL.

3.3.1 The DTable scheduler and variable OPA MTUs

In our original OPA-based simulation model exposed in Sect. 2.2, the global MTU is one packet of 128 bytes (i.e. 16 flits of 64 bits).

However, if the MTU is one packet sized in all SLs, and minimum entry weight is also one packet sized, the deficit counter will never be used. Moreover, the main advantage of the DTable scheduler is the use of different MTUs for different SLs [33] which allows to decouple the bandwidth assignments from the latency requirements (see Sect. 3.3.2 for further details).

To achieve this, we have modified the delivery message system. Before sending the message to the next network element, the DTable scheduler has to ensure that: i) the entire message fits onto the neighbour receiving buffer and ii) there is enough remaining weight for the selected VL. Therefore, *SL_MTU* tables are used, which have as many entries as SLs and each entry stores the associated MTU of each SL. The message generation is also based on those tables. For instance, if a given SL has an MTU of three packets, the SL will always generate messages of three packets. Moreover, when a transmission is performed, the SL will deliver three consecutive packets. Note that because of the switching technique used (i.e. virtual-cut through) and the atomic delivering message system, all flits of the same message are stored, sent and received consecutively. Then, VL buffers must have enough space (i.e. flow control credits) for storing at least the biggest MTU in the system.

3.3.2 DTable configuration methodology

In order to provide applications, flows or SLs specific QoS differences, DTable arbitration tables must be configured in a proper way. DTable scheduling mechanisms themselves do not provide QoS without applying a proper configuration methodology [19].

As stated in Sect. 3.3, the maximum distance between any pair of consecutive table entries assigned to the same SL allows to control the latency distribution among SLs [32]. In a given arbitration table configured to meet their latency requirements, we would like to be able to assign the SLi a certain bandwidth ϕ_i in a flexible way. In other words, this means to keep the minimum bandwidth $min\phi_i$ that can be assigned to the SLi as small as possible, and the maximum bandwidth $max\phi_i$ assignable to the SLi as large as possible. Table 4 shows the definition of all parameters involved in the configuration methodology.

The maximum total weight that can be divided among the table entries is $M \times N$. However, we have fixed it to a lower value called *pool*, which is determined by the k configuration parameter. Sect. 3.3.1 explains that a specific *MTU* value can be assigned for each SL. Then, the bandwidth ϕ_i assigned to the SLi is:

$$\phi_i = \frac{\sum_{j=0}^{J-1} \text{weight}_j}{pool},$$

Table 4 Arbitration table parameters

$\max\phi_i, \min\phi_i$	Maximum/Minimum bandwidth assignable to the i^{th} SL
ϕ_i	Bandwidth assigned to the i^{th} SL
N	Number of entries of the arbitration table
n_i	Number of entries assigned to the i^{th} SL
GMTU	General maximum transfer unit
MTU_i	Specific maximum transfer unit of the i^{th} SL
M	Maximum weight per table entry
pool	Bandwidth pool
k	Bandwidth pool decoupling parameter
w	Maximum weight decoupling parameter

where J is the number of table entries assigned to the SLi and $weight_j$ is the weight assigned to the table entry j. Therefore, $min\phi_i$ and $max\phi_i$ assignable bandwidth values to the SLi are:

$$min\phi_i = \frac{n_i \times \text{MTU}_i}{pool}, \; max\phi_i = \frac{n_i \times M}{pool}.$$

Let's define M and $pool$ using the $GMTU$ parameter and the decoupling parameters w and k:

$$M = \text{GMTU} \times w, pool = N \times \text{GMTU} \times k,$$

where $k \leq w$ because the bandwidth pool has to be smaller than $N \times M$. Hence, the maximum and minimum bandwidth depend not only on the proportion of table entries n_i, but also on the w and k parameters and the proportion between their specific MTU_i and $GMTU$:

$$\min\phi_i = \frac{n_i \times \text{MTU}_i}{N \times \text{GMTU} \times k},$$
$$\max\phi_i = \frac{n_i \times \text{GMTU} \times w}{N \times \text{GMTU} \times k} = \frac{n_i \times w}{N \times k}.$$

Therefore, parameters w, k and the specific MTU_i assigned to each SL allow to vary the maximum and minimum bandwidth assignable to SLs without affecting the final latency [19].

3.3.3 DTable bandwidth correction algorithm

Once the configuration methodology has been applied, we can choose a bandwidth ϕ_i for each SL between the given $min\phi_i$ and $max\phi_i$ range. Then, the total entry weight $Tweight_i$ has to be computed as $pool \times \phi_i$. After that, we have to obtain the entry weight as $\frac{Tweight_i}{n_i}$ for each SL and fill in arbitration tables with these values. As

stated in Sect. 3.3, the entry weight represents how many packets can be delivered from an active SL, so it must have at least enough weight to deliver one packet/MTU. Moreover, it must be an integer value because float numbers will produce some issues:

- The fractional part will only be useful once it is accumulated in the deficit counter and the sum is equal to one packet/MTU.
- The final hardware implementation will require more silicon area due to IEEE 754 floating point representation [34].
- The final entry weight may not be enough for delivering a packet/MTU from an active SL without cycling through arbitration tables several times.

To put this right, the entry weight obtained as $\frac{T\text{weight}_i}{n_i}$ will always be rounded up. However, this could produce some bandwidth imprecisions. Table 5 shows an example about this issue. In this example, each SL will get $\phi_i = \frac{1}{\text{pool}}$. However, as seen in the $R\phi_i$ column, the real SLi bandwidth is $\phi_i \neq \frac{1}{\text{pool}}$. Specifically $R\phi_0 = \frac{448}{1216}$, $R\phi_1 = \frac{384}{1216}$ and $R\phi_2 = \frac{384}{1216}$.

To solve this issue, the DTable bandwidth correction algorithm is applied. First, the bandwidth difference between ϕ_i and $R\phi_i$ is obtained. The column $\phi_i - R\phi_i$ on Table 5 shows the bandwidth differences. Secondly, the amount of extra weight that SLi table entries require, called $Dweight_i$, is calculated:

$$D\text{weight}_i = -1 \times \text{Round}((R\phi_i - \phi_i) \times \sum_{j=0}^{N-1} \text{weight}_j)$$

For instance, in Table 5 we have $Dweight_0 = -(0.03502 \times 1216) = -43$, $Dweight_1 = --(0.01751 \times 1216) = 21$, etc. Finally, the $Dweight_i$ value is added to $\sum_{j=0}^{n_i-1} weight_j$ getting $Fweight_i$. As can be seen in Table 5, in the column $F\phi_i$, final bandwidths are very close to the desired ones.

Another important aspect is how and when $Dweight_i$ is added to arbitration tables. Assuming that the DTable configuration and adjustment are done by the FM during the starting up process, the simplest strategy is: (i) to populate a pre-arbitration table with the bandwidth imperfections discussed here; (ii) to perform the DTable correction algorithm and (iii) to send the final arbitration table to network elements. However, there is a large range of possible ways to add $Dweight_i$. In our OPA-based simulation model, the algorithm always starts from the end of the arbitration table incrementing weight to each entry weight in a round-robin way. Table 6 shows an arbitration table where SLs have a $Dweight_i$ of -3, 1 and 2 for SL0, SL1 and SL2 respectively.

The first three rows show the arbitration table before running the DTable bandwidth correction algorithm and the last three rows after running it. The first and fourth rows show the table entry identifiers and the third and sixth rows the SL identifier and the associated weight respectively. The algorithm starts with SL0 and the entry 6 performing $4 + (-1) = 3$, moves to the entry 4 performing $4 + (-1) = 3$ and then finishes with the entry 2 performing $4 + (-1) = 3$. Then, the algorithm

Table 5 DTable configuration example with adjustment errors

SL	n_i	MTU_i	$\min\phi_i$	$\max\phi_i$	ϕ_i	pool	E.W.	$T\text{weight}_i$	$\sum_{j=0}^{n_i-1} \text{weight}_j$	$R\phi_i$	$R\phi_i - \phi_i$	$D\text{weight}_i$	$F\text{weight}_i$	$F\phi_i$
0	64	1	0.05556	0.66667	0.33334	1152	7	384.0768	448	0.36842	0.03502	−43	405	0.33333
1	32	2	0.05556	0.33333	0.33333	1152	12	383.9616	384	0.31579	−0.01751	21	405	0.33333
2	32	3	0.08333	0.33333	0.33333	1152	12	383.9616	384	0.31578	−0.01751	21	405	0.33333
Total	128				1			1152	1216				1215	

N = 128, GMTU= 3, w = 4, k = 3

Table 6 Arbitration table example

0	1	2	3	4	5	6	7
SL, W	SL, W	SL, W	SL, W	SL, W	SL, W	SL, W	SL, W
0, 4	1, 2	0, 4	2, 3	0, 4	1, 2	0, 4	2, 3
0	1	2	3	4	5	6	7
SL, W	SL, W	SL, W	SL, W	SL, W	SL, W	SL, W	SL, W
0, 4	1, 2	0, 3	2, 4	0, 3	1, 3	0, 3	2, 4

continues with SL1 and the entry 5 performing $3 + 1 = 4$. Finally, the algorithm moves to SL2, it starts with the entry 7 performing $3 + 1 = 4$ and moves to the entry 3 performing $3 + 1 = 4$. Once all $Dweight_i$ are zero for each SL, the algorithm stops. On the other hand, it could be interesting to study a different approach to find out if there are differences among start from the button and the top of the table. However, it is essential that the system checks during the increasing process if the weight on the entries is enough to deliver a packet/MTU.

4 Performance evaluation

In this section, we evaluate the performance of DTable and SBT proposals against a round-robin scheduler as the baseline reference. We have used our simulator Hiperion which implements the simulation model explained in Sect. 2.2, as well as the QoS mechanisms detailed in Sect. 2.1. Note that although we use OPA for configuring the network parameters, our proposal can be applied to any hierarchical-crossbar-switch based interconnection network.

We have also evaluated the QoS mechanisms in two different scenarios. In the first scenario, the network has been evaluated using a synthetic traffic model composed of several traffic flows. These flows represent the network load generated by applications commonly found in cluster and data centers. In the second scenario, the synthetic HPC flow is replaced by the traffic of real MPI applications using the VEF trace framework [29].

In Sect. 4.1 we present the network model used in the performance evaluation. Section 4.2 presents the synthetic scenario and its results, while Sect. 4.3 includes the evaluation and results obtained using the MPI traces.

4.1 Network model

We have used two different interconnection topologies with two different layouts: a 2D Torus with 8x8 switches, a 3D Torus with 8x8x4 switches, a 8-ary 3-tree with 192 switches and a 24-ary 2-tree with 48 switches. The configuration of each scenario is the following:

- The 2D Torus configuration has 512 endpoints (NICs). Each switch has 48 ports: eight single links to endpoints and four 10x trunk links to neighboring switches.

Table 7 SL2SC and SC2VL tables configuration

SL2SC		SC2VL			
SL	SC	SC	VL	SC	VL
VO	0 1	0	0	5	5
VI	2 3	1	1	6	6
CL	4 5	2	2	7	7
BE	6 7	3	3	8	6
BK	8 9	4	4	9	7

- The 3D Torus configuration has 1024 endpoints connected, the switches have a radix of 28 with 4x trunk links.
- The 8-ary 3-tree has been configured with 512 NICs and 16-port switches.
- The 24-ary 2-tree has a total of 576 endpoints and 48-ports switches.

We have chosen these topologies because they are very common and well known solutions in high performance environments. The detailed explanation about the switch architecture can be found in Sect. 2. The SL2SC and SC2VL tables configuration is shown in Table 7. For instance, the SL VO has two SCs, SC0 and SC1, and they have VL0 and VL1 associated respectively. Further details about SLs will be provided in Sect. 4.2.1.

The switch model implements a credit-based flow control protocol. The packets will be only transmitted when there is enough buffer space in the next network device. Therefore, packets are not dropped when congestion appears. Traffic with similar characteristics is aggregated via SLs, the packet scheduling is performed with SLs and flow control via VCs. According to [17], the GMTU of OPA messages may be up to 8KB, but we have used a GMTU of 1KB in this evaluation for the sake of simplicity. Nevertheless, the evaluation may be performed with greater MTUs using larger buffers. The credit-based flow control unit is 64 bytes, and thus, the GMTU is up to 16 credits.

As stated before, we have used input, output and central buffer queuing architecture. The buffer capacity is 65,536 bytes (64 × GMTU) per input and output ports of switches and 32,768 bytes (32 × GMTU) at the network interfaces. The central crossbar buffer capacity is 131,072 bytes (128 × GMTU) per MPort. If an application wants to inject a packet into a network interface queue but the queue is full, we assume that the packet is stored in the application layer queue.

4.2 Performance evaluation using the synthetic traffic model

In this section we explain the details of the evaluation performed using synthetic traffic. Section 4.2.1 presents the traffic model. The scheduler configurations for the different QoS mechanisms are shown in Sect. 4.2.2. Finally, Sect. 4.2.3 shows and analyzes the obtained results.

Table 8 Set of SCs considered

Type	SL	Description	Traffic pattern	Message size
QoS	Voice (VO)	Audio and online videogames backend traffic	CBR connections	128B
QoS	Video (VI)	Video streaming traffic	CBR connections	256B
QoS	Controlled load (CL)	High performance computing traffic	CBR connections	512B
Best-effort	Best-effort (BE)	Backup protocols, email system, etc.	CBR connections	1024B
Best-effort	Background (BK)	Rest of applications and services	CBR connections	1024B

4.2.1 Traffic model

Table 8 shows each traffic type considered. There are five types of traffic flows, three SLs with explicit QoS requirements such as latency and bandwidth, and two SLs for best effort traffic with slight different levels of priority among them.

The packets from each SL have been simulated using different Constant-Bit-Rate (CBR) distributions. We have selected the following packet payloads for each SL:

- Voice (VO) traffic is generated using a packet payload of 128 bytes. According to [35], the payload value of voice packets ranges from 20 to 160 bytes.
- Video (VI) traffic is generated using a packet payload of 256 bytes. According to [36], a payload ranging from 100 bytes to 64KB is feasible.
- Controlled Load (CL) traffic is generated using a packet payload of 512 bytes, representing a possible average packet payload of many HPC application communications.
- The traffic of the best effort SLs, Best-effort (BE) and Background (BK), is generated using a packet payload of 1024 bytes.

For all cases, the destination pattern is uniform in order to fully load the network. Note that we have chosen a heterogeneous scenario where multiple types of traffic are mixed. However, our proposal is aimed to any environment where flows with different QoS requirements coexist in a high performance network.

4.2.2 Simulated scenario and scheduler configurations

We have supposed a scenario where the goal is to obtain 10% of the egress link bandwidth and the lowest packet latency to the voice traffic; 30% of bandwidth and a higher packet latency than the voice traffic to the video traffic; around 50% of bandwidth and a higher packet latency than voice traffic to the controlled load traffic and the remaining 10% of bandwidth and the highest latency to the best effort traffic. The bandwidth percentages are intended to represent, as close as possible, a realistic combination of traffic and QoS needs from applications with different requirements. We have configured the schedulers according to these traffic requirements.

Table 9 Application of the decoupling methodology

SL	Distance	#entr.	%entr.	MTU_i	$min\phi_i$	$max\phi_i$
VO	2	64	50	128	0.03125	2
VI	4	32	25	256	0.03125	1
CL	8	16	12.5	512	0.03125	0.5
BE	16	8	6.25	1024	0.03125	0.25
BK	16	8	6.25	1024	0.03125	0.25
	Total	128	100		0.15625	4

N = 128, GMTU= 16, w = 8, k = 2

As mentioned in Sect. 3.2, SBT is the simplest QoS algorithm in terms of complexity and configuration. We have filled in the SBT tables with a weight proportional to the percentages mentioned before for each table row. That is, a weight of 10 for the first table row (VO traffic), a weight of 30 for the second table row (VI traffic), etc. SBT does not require any more configurations. Note, however, that the total table weight has to be 100.

In the case of the DTable scheduler, the configuration process is more complex. We have applied the decoupling methodology explained in Sects. 3.3 and in [31], distributing the table entries among SLs according to latency requirements. To do that, we have established the maximum distance of two consecutive table entries of the same SL as follows: a maximum distance of two entries for SL VO and a maximum distance of 16 to SL BE and SL BK. Table 9 also shows the total number of table entries (#entr.) and the proportion of table entries given to each SL (%entr.). For maximum flexibility, the MTU of each SL has been established as small as the expected packet size of each traffic type. Specifically, we have set an MTU of 128 bytes for VO, an MTU of 256 bytes for VI, an MTU of 512 bytes for CL and an MTU of 1024 bytes, which is the maximum, for BE and BK traffic.

Finally, we have configured proper values for w and k parameters. The main condition that we have taken into account is that we want for SL CL a bandwidth several times higher than the proportion of table entries assigned. Moreover, the SL VO has assigned a high proportion of table entries, whilst it requires a small proportion of bandwidth. However, it is important to keep the k parameter value as small as possible in order to obtain good latency performance. We have finally chosen a value of 8 for w and a value of 2 for k. This combination of values allows us to get a $[min\phi_i, max\phi_i]$ range that fits within the bandwidth needed. Table 9 shows the minimum and maximum bandwidth that may be assigned to each SL with this configuration.

Table 10 shows the total amount of traffic that each SL injects, expressed in flits/cycle/NIC (*Inj.* column). This table also shows the total weight (T.W.) that we have distributed among the table entries of each SL and the weight assigned to each table entry (E.W.) of each SL. Note that the SL VO and the SL CL have an E.W. of 6-7 and 130 respectively, due to the DTable bandwidth correction algorithm (Sect. 3.3.3). On the one hand, the SL VO has a $Dweight_0 = -32$ and therefore the first 32 table entries have a weight of 7 and the next 32 table entries have a weight of 6. On the other hand, the SL CL has a $Dweight_2 = 32$ and therefore each table entry

Table 10 Bandwidth configuration of DTable and SBT schedulers

SL	Inj.	Scheduler configuration							
		DTable					SBT		
		#entr.	E.W	T.W.	$R\phi_i$	$F\phi_i$	#entr.	E.W.	T.W.
VO	0.1	64	6–7	416	0.11	0.1	1	10	10
VI	0.3	32	39	1248	0.3	0.3	1	10	10
CL	0.5	16	130	2080	0.49	0.5	1	50	50
BE	0.05	8	26	208	0.05	0.05	1	5	5
BK	0.05	8	26	208	0.05	0.05	1	5	5
Total	1	128		4160			5		100

has a weight of 130. The rest of the SLs are not affected by the DTable bandwidth correction algorithm due to the fact that the obtained bandwidth is equal to the configured bandwidth. Columns $R\phi_i$ and $F\phi_i$ show the bandwidth percentage assigned to each SL before and after applying the bandwidth correction algorithm, respectively. Without the adjustment, SLs VO and CL would get 11% and 49% instead of the desired 10% and 50%, respectively. In this specific example, the bandwidth percentage difference is 1%. Nevertheless, in a scenario where link bandwidths are up to 12.5 GB/s, those differences could have a significant impact in the application execution over time. Besides, without the bandwidth correction algorithm, the system administrators would be forced to find an appropriate combination of parameters, i.e. a combination of MTU, k and w values, that would allow them to obtain the required bandwidth distribution being, in some cases, not possible. Note that in the case of SBT, the scheduler has only one entry for each SL, because entry weights and the total weight are equal.

4.2.3 Simulation results using the synthetic workload

In this section, simulation results are shown. The values shown for each injection rate are the average of 30 different simulations varying the seed of the random number generation. We have used two metrics to evaluate the networks and the different QoS mechanisms:

- End-to-end latency: Message latency from generation to delivery. It is the latency that users will experience.
- Normalized SL throughput: Total amount data expressed in flits/cycle/NIC transmitted through the interconnection network. This metric has been divided by SL and normalized to the total throughput.

Figures 3a, c, 4a and c show the end-to-end latency in the 2D Torus, 3D Torus, 8-ary 3-tree and 24-ary 2-tree topologies, respectively. Note that we have represented each SL in different QoS algorithms with the same color and line pattern, and each SL is represented always with the same point style, e.g. SLs when DTable (DT)

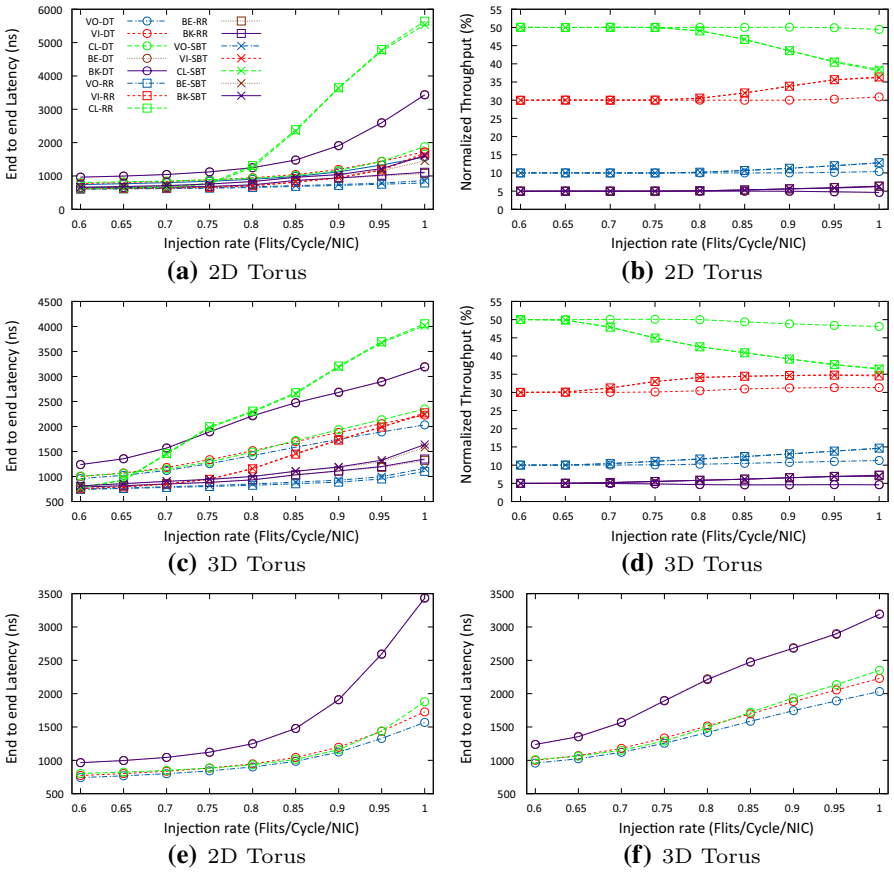

Fig. 3 Performance comparison of each SL using a 2D Torus topology (3a, b and e) and a 3D Torus topology (3c, d and f). Results in Fig. 3a and c refer to end-to-end latency, results in Fig. 3b and d refer to normalized throughput and results in Fig. 3e and f refer to end-to-end DTable latency

is used are represented with a circle and the SL VO is plotted with a line-dot pattern and a blue colour, SLs when SBT is used are represented with a square, SLs when round-robin baseline (RR) is used are represented with the cross symbol.

As explained in Sect. 3.2, SBT and RR algorithms do not provide latency differences, which can be seen in end-to-end latency figures: the more generation ratio is assigned to SLs, the more latency they have. The only exception is in the case of SLs BE and BK, which achieve a slightly higher latency because of sharing the VLs. For instance, the SL VO has the same injection rate as SLs BE and BK combined and the best-effort SLs achieve higher latency values when SBT or RR are used. Referring DTable end-to-end latency, in some cases SLs get more latency than SBT or RR. This is because DTable does not reduce the overall latency to ensure the latency requirements, but it splits the total latency between SLs based on table entries distance. Given that, for example, SL VO using DTable gets a higher latency than the

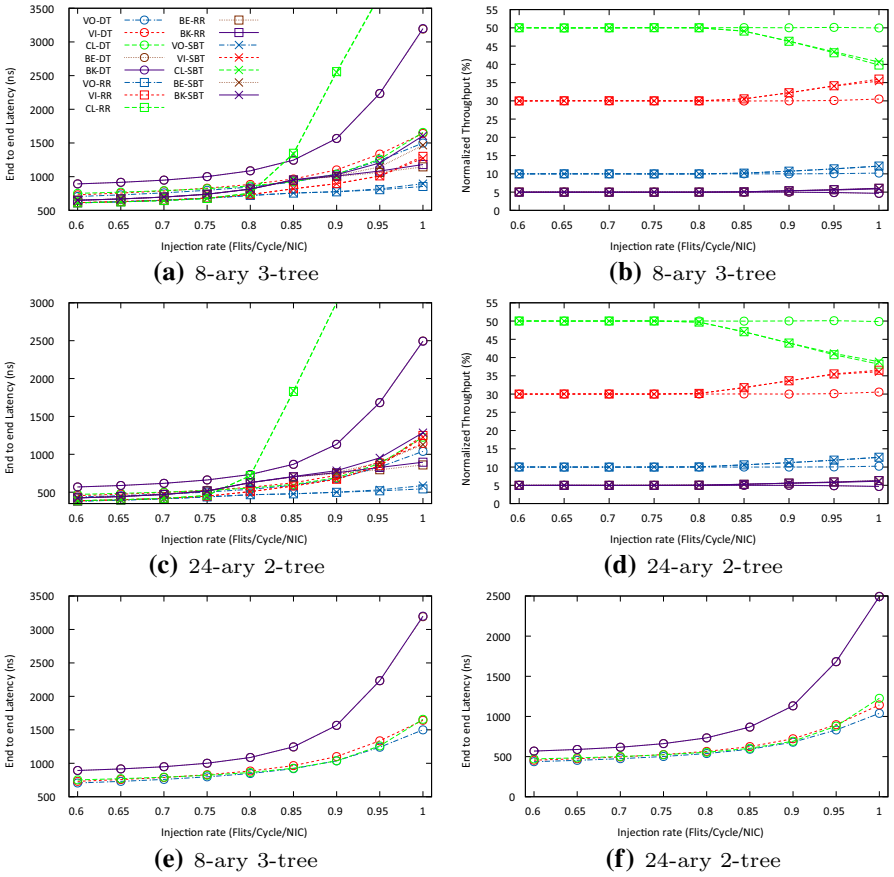

Fig. 4 Performance comparison of each SL using a 8-ary 3-tree topology (4a, b and e) and a 24-ary 2-tree topology (4c, d and f). Results in Fig. 4a and c refer to end-to-end latency, in Fig. 4b and d refer to normalized throughput and in Fig. 4e and f refer to end-to-end DTable latency

same SL when SBT or RR are used, but SL CL with DTable gets lower latency than this SL with SBT or RR. Note that in Fig. 4a and c the SL CL in the SBT and RR tests is off the chart. We have decided to leave them outside for the sake of clarity, otherwise, the rest of the lines would be too close to each other. The end-to-end latency for the injection rates of 1 flit/cycle/NIC is over 4,500 ns in both cases.

Figures 3b, d, 4b and d show the normalized throughput achieved on each topology configuration. DTable obtains a normalized throughput very close to the desired one with an error of $\pm$ 2%. In the case of SBT, it gets an error greater than DTable, specifically, it gets almost the same bandwidth division as the RR scheduler. These results suggest that SBT is not suitable for high-performance interconnection networks. The maximum throughput performance for each configuration is: 0.94 flits/cycle/NIC with DTable and 0.8 flits/cycle/NIC with SBT or RR for 2D Torus; 0.78 flits/cycle/NIC with DTable and 0.68 flits/cycle/NIC with SBT or RR for 3D Torus;

0.95 flits/cycle/NIC with DTable and 0.85 flits/cycle/NIC with SBT or RR for 8-ary 3-tree; and 0.95 flits/cycle/NIC with DTable and 0.8 flits/cycle/NIC with SBT or RR 24-ary 2-tree.

DTable achieves more throughput because the scheduler has a hit rate higher than SBT or RR, and the fact that the scheduler does not try to inject long bursts of packets helps to significantly reduce the head-on-line blocking.

Regarding the topology configurations, in terms of end-to-end latency, Torus scenarios show a higher latency values in all SLs before and after the network reaches saturation point (Fig. 3a, c, e and f). Also, Torus topologies penalize less the best-effort SLs after the saturation point than *k*-ary *n*-tree configurations, i.e. the latency of all SLs increases progressively as the injection rate increases. The expected behavior is that the best-effort SLs increase its latency as much as possible before increasing the latency of high priority SLs. This fact is very obvious in the 3D Torus Fig. 3d. The *k*-ary *n*-tree configurations keep the high priority latencies closer to each other than *n*D Torus topologies, which means that *k*-ary *n*-tree topologies segregate the traffic better than the *n*D Torus scenarios. Results do not show significant differences in terms of achieved throughput per SL. Only the 3D Torus topology in Fig. 3b shows a slight throughput reduction in SLs using DTable scheduler and the network gets congested earlier than others. This happens because the head-of-line blocking on the 3D Torus topology is stronger than on the other networks, due to this topology has more endpoints and thinner trunk links than the 2D Torus topology. Nevertheless, the DTable scheduler is able to keep the bandwidth distribution very close to the expected distribution.

Finally, Figs. 3e, f, 4e and f show the end-to-end latency of DTable SLs. The main aim of these figures is to show the latency differentiation among SLs. We have established that the SL VO must have the lowest latency, the SL VI must have latency higher than the SL VO and so on. As can be seen, SLs get a latency proportional to the desired ones. After the network gets saturated, i.e. the NICs inject more packets per cycle that they are able to deliver, SLs entry distances are more clear and the latency differentiation is more obvious.

SBT and RR achieve similar behaviors before the network gets saturated. Their results are practically the same because both algorithms work in the same way and, before saturation, each SL can inject as much flits as the NICs generate. However, when saturation appears, there are differences because SBT will try to adjust the throughput to the desired, while RR will try to give to each SL $\frac{1}{NumSLs}$ of the available bandwidth. Note that differences are more clear with higher injection rates. Nevertheless, we have decided not to include these ratios because are just theoretical injection rates.

4.3 Scenario using application trace files

As stated in Sect. 2.2, Hiperion includes support for MPI application trace files using the VEF trace framework [29]. The traces are a very representative way to know how a real HPC application will behave in any interconnection network simulator without requiring a complex system to run the applications. Therefore, we have

also used trace files for performing more representative experiments. We expect to see how a poor QoS assignment or the absence of QoS degrades the system performance in terms of application runtime. Hence, those experiments will give us a better perspective on how OPA behaves using traffic from real MPI applications.

We have carried out experiments using multiple trace files obtained from different MPI applications: NAMD (*NAMD*) [37], a parallel application for simulating large biomolecular systems; GROMACS (*gro*) [38], a scientific application to perform molecular dynamics; and LINPACK (*HPL*) and *MPIRandomAccess* applications from of the HPCC Benchmark Suite [39], which is one of the most used benchmark for evaluating supercomputers. These applications have run considering 512 tasks. On each experiment, we have used the SLs CL and BE, one carrying all the trace file traffic and the other with CBR connections detailed in Sect. 4.2.1 and vice-versa. The purpose of the CBR traffic is to introduce background network workload in order to see how the output scheduling algorithm distinguishes between traffic classes. Otherwise, there would be no competition for resources and the trace would occupy them all, making no difference between using QoS or not.

We have performed several experiments for each trace file varying the injection rate of the background traffic (1%, 5%, 10% and 20%) and the SL at which the trace file is injected (SL CL and SL BE). This combination reveals us the scheduler and architecture behavior when the application could use more or less network resources and when the application has to compete harder for resources, because the increase of the background traffic will try to use them. Those injection rate values have been chosen because they allow us to complete the experiments in a reasonable amount of time while significant results can be extracted. Also, we have run the trace file without any background traffic and QoS support to get the execution time baseline.

We have used DTable as the output scheduling algorithm with the configuration shown in Table 10. The results of RR are also included in order to compare the application performance without QoS mechanism. From the results obtained with the synthetic workload, we have considered DTable more interesting for this experimentation than SBT. For this reason, and for the sake of clarity and not overloading the figures with too much information, we have not included the SBT scheduler in the results.

In those experiments, we have used the same network configuration than the exposed in Sect. 4.1. We have only changed the interconnection topologies, since the trace evaluation is limited by the number of tasks of the trace. Since we have only available 512-task traces and this size is not enough to fulfill the systems presented on Sect. 4.1, we have chosen two different topologies: a 2D Torus with 4x4 switches and a 8-ary 2-tree with 16 switches. The configuration of each topology is the following:

- The 2D Torus configuration has 128 NICs. Each switch has 48 ports: eight single links to NICs and four 10x trunk links to neighboring switches.
- The 8-ary 2-tree has been configured with 64 NICs and switches with 16 ports.

To analyse the results of these experiments, we have used the normalized total execution time. It is expressed as the percentage of the execution time between the QoS

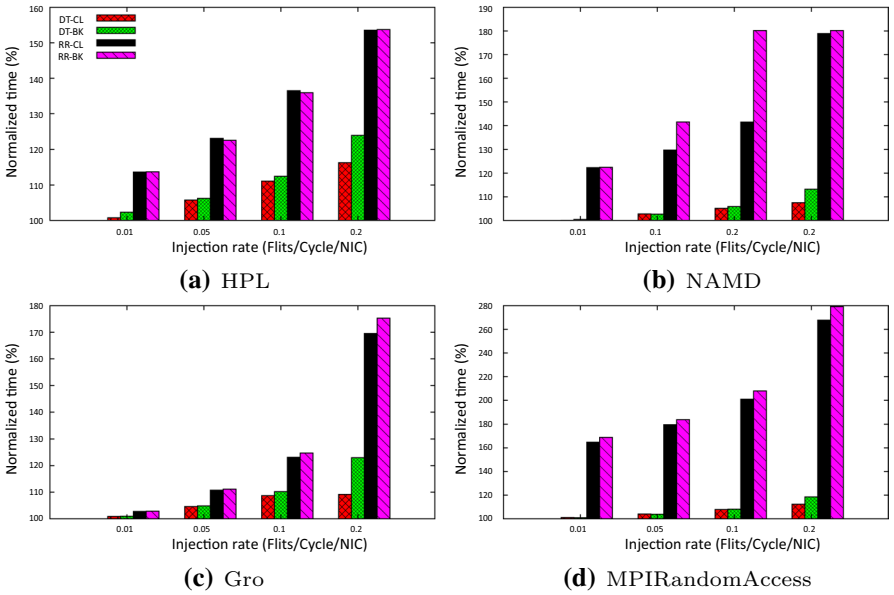

Fig. 5 Performance comparison of 8-ary 2-tree topology using trace files and background traffic

scenario with background traffic and the scenario without background traffic QoS support.

4.3.1 Simulation results

In this section, simulation results using the application traces are shown. Figures 5 and 6 show the bandwidth differences produced by DTable using trace files with the 8-ary 2-tree and the 2D Torus configurations, respectively. Those topologies as well as the experiment configurations are detailed in Sect. 4.3. Each bar in Figs. 5 and 6 represents the normalized execution time, expressed in percentage, between the trace file with background traffic and QoS and the same trace file without QoS enabled. Those percentages have been calculated for each SL. For example, in Fig. 5 *NAMD*-CL result within an injection rate of 0.01 flits/cycle/NIC is calculated running a simulation where: i) the QoS is enabled; ii) the *NAMD* trace file is injected by the SL CL; and iii) the background traffic is generated in the SL BK at injection rate of 0.01 flits/cycle/NIC. The execution time of this simulation is compared with the obtained using the *NAMD* trace disabling the QoS and removing the background traffic. This process has been performed to calculate each result.

Regarding the results shown in Figs. 5 and 6, the total execution time increases with the injection rate, being the *MPIRandomAccess* trace file the most time-consuming in both topologies. On each experiment performed, the results of SLs CL and BK using the same trace file, regardless of the background traffic injection rate, is always lower in the case of the CL SL. This fact is more obvious as the background traffic injection ratio increases. This means that the DTable scheduling

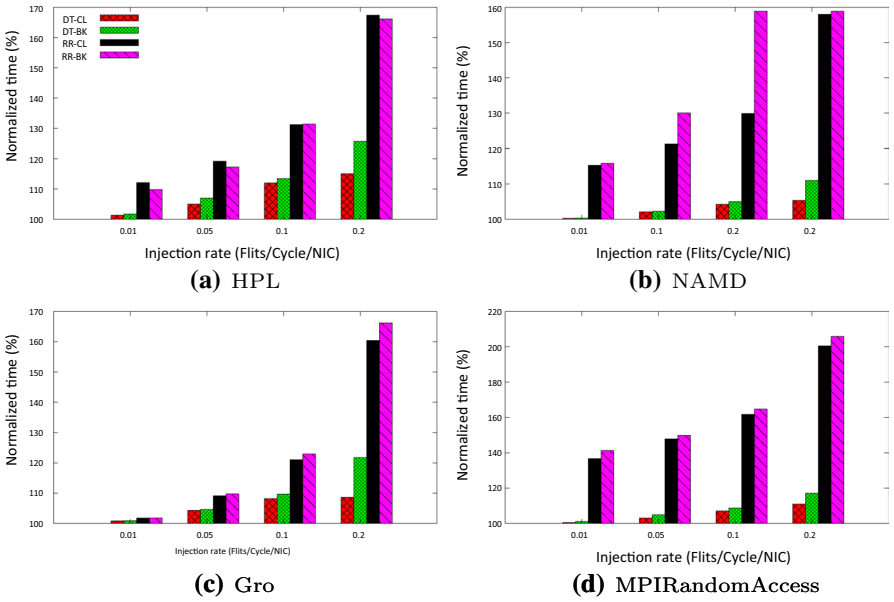

Fig. 6 Performance comparison of 2D Torus topology using trace files and background traffic

output algorithm is able to properly segregate the traffic flows because the SL CL has much more resources assigned than the SL BK. Therefore, although the SL BK is trying to progressively allocate more resources, DTable is properly limiting the amount of resources it can use. As the injection rate of the background workload increases, the differences between the SLs CL and BK are increased. This is due to the fact that as the background workload increases, it tries to use more resources and it is penalized by DTable increasing its execution time. At low background injection rates, the differences between SLs CL and BK are in the range of 1% to 5% because they do not have to compete strongly for resources as the network has sufficient capacity to serve both SLs.

For both topology configurations, in the scenarios where a RR output scheduler, i.e. no QoS is provided, the application execution times are higher than in the scenarios where DTable scheduler is used. Those execution times are even higher at low background traffic injection rates. Hence, the DTable improves the applications performance by distributing the available network resources.

Comparing the results obtained in Figs. 5 and 6, the trend of the results is the same. However, the applications trend to require more time to complete its execution in the 2D Torus topology. This variation is due to the 2D Torus topology has a smaller radix and a larger number of connected NICs than the 8-ary 2-tree. The percentage differences between SLs using DTable in both topologies do not show significant differences. Therefore, the topology configuration does not have any impact on the DTable output scheduler resources distribution. Nevertheless, this is not true in the case of RR were the percentage differences between SLs fluctuate depending on the topology.

Summing up, DTable is able to distribute the resources according to its configuration and outperforms the RR scheduler in terms of total execution time required by the applications and resources distribution among SLs.

5 Conclusion

To enable QoS support in high-performance interconnection networks, the most critical architectural decision is the selection of an adequate output scheduling algorithm, which is in charge of selecting the next packet to be transmitted at any moment. This output scheduling algorithm has to keep the computational complexity as low as possible so it can be realistically implemented. In this paper we have addressed this issue in the context of hierarchical crossbar switch architectures, specifically on OPA switches using SBT and DTable mechanisms. These two table-based output scheduling algorithms keep the computational complexity low and they are able to provide the required QoS differences. Moreover, we have proposed a DTable bandwidth correction algorithm capable of adjusting the bandwidth imprecisions produced by the baseline DTable configuration methodology. This methodology, in conjunction with the bandwidth correction algorithm, allows to set any bandwidth proportion to SLs.

We have evaluated the performance of these scheduling algorithms using a heterogeneous scenario where multiple traffic flows coexist. We have carried out different experiments using several topology configurations and we have compared the results against a round-robin output scheduler, which represents a scenario without QoS provision. On the one hand, results show that SBT is capable of providing bandwidth differences but it is not able to provide latency differences. Moreover, SBT is not able to distribute the bandwidth according to its configuration. On the other hand, DTable is capable of providing bandwidth and latency differences among SLs, and we are able to establish which SL will experience higher and lower end-to-end latency.

We have also carried out several experiments using multiple network topologies with the aim of finding out differences in the behavior of schedulers in terms of QoS provision. Results show that k-ary n-tree topologies are able to provide slightly better end-to-end latency results than the nD Torus configurations. In terms of throughput, there are not significant differences between configurations.

Moreover, we have carried out experiments with two different topologies using communication trace files obtained from real MPI applications and background network workload. These experiments have been performed with DTable as the output scheduling algorithm. Results show that even with real MPI communications, DTable is able to properly segregate the traffic according with the predefined configuration with independence of the network topology and the trace file.

As explained in Sect. 2.2, unlike non-hierarchical switches, sometimes packets have to cross the central crossbar in order to reach the required output buffer. Hence, we are currently evaluating the impact of including another DTable scheduler in these central buffers. We also are planning to perform a deeper hardware study in

order to offer estimates about the silicon area that this middle scheduling algorithm would require.

Acknowledgements This work has been supported by the Junta de Comunidades de Castilla-La Mancha, European Commission (FEDER funds) and Ministerio de Ciencia, Innovación y Universidades under projects SBPLY/17/180501/000498 and RTI2018-098156-B-C52, respectively. It is also co-financed by the University of Castilla-La Mancha and Fondo Europeo de Desarrollo Regional funds under project 2021-GRIN-31042. Javier Cano-Cano is also funded by the MINECO under FPI grant BES-2016-078800.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

References

1. Scott S, Abts D, Kim J, Dally WJ (2006) The blackwidow high-radix clos network. ACM SIGARCH Computer Archit News 34(2):16–28
2. Abts D, Kim J (2011) High performance datacenter networks: architectures, algorithms, and opportunities. Synth Lect Computer Archit 6(1):1–115
3. Ahn JH, Son YH, Kim J (2013) Scalable high-radix router microarchitecture using a network switch organization. ACM Trans Archit Code Optim (TACO) 10(3):17
4. Birrittella MS, Debbage M, Huggahalli R, Kunz J, Lovett T, Rimmer T, Underwood KD, Zak RC (2016) Enabling scalable high-performance systems with the intel omni-path architecture. IEEE Micro 36(4):38–47
5. Scott S (2019) Rossetta: A 64-port switch for Cray's Slingshot Interconnect. In: Proceedings of the 26th Annual Symposium on High-Performance Interconnects (HOTI)
6. Pfister GF (2001) An introduction to the InfiniBand architecture. High Perform Mass Storage Parallel I/O 42:617–632
7. D'Ambrosia J, Law D, Nowell M (2008) 40 Gigabit ethernet and 100 gigabit ethernet technology overview. Nov
8. TOP500 homepage. https://www.top500.org. (Accessed July 18, 2019)
9. Montessoro PL, Pierattoni D (2001) Advanced research issues for tomorrow's multimedia networks. In: Proceedings International Conference on Information Technology: Coding and Computing, pp. 336–340. IEEE
10. Nandy B, Seddigh N, Pieda P, Ethridge J (2000) Intelligent traffic conditioners for assured forwarding based differentiated services networks. networking 2000 broadband communications. high performance networking, and performance of communication networks. Springer, Berlin, Heidelberg, pp 540–554
11. Wilke JJ, Kenny JP (2020) Opportunities and limitations of quality-of-service in message passing applications on adaptively routed dragonfly and fat tree networks. In: 2020 IEEE International Conference on Cluster Computing (CLUSTER), pp. 109–118. IEEE
12. Mubarak M, McGlohon N, Musleh M, Borch E, Ross RB, Huggahalli R, Chunduri S, Parker S, Carothers CD, Kumaran K (2019) Evaluating quality of service traffic classes on the megafly network. In: International Conference on High Performance Computing, pp. 3–20. Springer
13. Demers A, Keshav S, Shenker S (1989) Analysis and simulation of a fair queueing algorithm. In: ACM SIGCOMM Computer communication review, vol. 19, pp. 1–12. ACM

14. Greenberg AG, Madras N (1992) How fair is fair queuing. J ACM (JACM) 39(3):568–598
15. Sivaraman V (2000) End-to-end delay service in high-speed packet networks using earliest deadline first scheduling. University of California, Los Angeles, ???
16. Stiliadis D, Varma A (1998) Latency-rate servers: a general model for analysis of traffic scheduling algorithms. IEEE/ACM Trans Netw 6(5):611–624
17. Birrittella MS, Debbage M, Huggahalli R, Kunz J, Lovett T, Rimmer T, Underwood KD, Zak RC (2015) Intel® Omni-Path Architecture: Enabling scalable, high performance fabrics. In: high-performance interconnects (HOTI), 2015 IEEE 23rd Annual Symposium On, pp. 1–9. IEEE
18. Hiperion repository homepage. https://gitraap.i3a.info/fandujar/\hiperion. (Accessed July 8, 2021)
19. Martinez-Morais R, Alfaro-Cortes FJ, Sanchez JL (2009) Providing QoS with the deficit table scheduler. IEEE Trans Parallel Distrib Syst 21(3):327–341
20. Martínez R, Claver JM, Alfaro FJ, Sánchez JL (2012) Hardware implementation study of several new egress link scheduling algorithms. J Parallel Distrib Comput 72(8):975–989
21. Cornelis Networks homepage. https://cornelisnetworks.com. (Accessed December 14, 2020)
22. CRN Intel Spins Out Omni-Path Interconnect Business Into Stand-Alone Company. https://www.crn.com/news/components-peripherals/intel-spins-out-omni-path-interconnect-business-into-stand-alone-company. (Accessed December 14, 2020)
23. Agarwal N, Krishna T, Peh L-S, Jha NK (2009) Garnet: A detailed on-chip network model inside a full-system simulator. In: 2009 IEEE International Symposium on Performance Analysis of Systems and Software, pp. 33–42. IEEE
24. Böhm S, Engelmann C (2011) xsim: The extreme-scale simulator. In: 2011 International Conference on High Performance Computing & Simulation, pp. 280–286. IEEE
25. Cope J, Liu N, Lang S, Carns P, Carothers C, Ross R (2011) Codes: Enabling co-design of multilayer exascale storage architectures. In: proceedings of the workshop on emerging supercomputing technologies, vol. 2011. ACM
26. Rodrigues AF, Voskuilen GR, Hammond SD, Hemmert KS (2016) Structural simulation toolkit (SST). Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States)
27. Andújar FJ, Coll S, Alonso M, López P, Martínez J-M (2019) Powar: power-aware routing in hpc networks with on/off links. ACM Trans Archit Code Optim (TACO) 15(4):1–22
28. Andújar FJ, Villar JA, Sánchez JL, Alfaro FJ, Duato J (2013) Building 3d torus using low-profile expansion cards. IEEE Trans Computers 63(11):2701–2715
29. Andújar FJ, Villar JA, Sánchez JL, Alfaro FJ, Escudero-Sahuquillo J (2016) An open-source family of tools to reproduce MPI-based workloads in interconnection network simulators. J Supercomput 72(12):4601–4628
30. Tamir Y, Frazier GL (1992) Dynamically-allocated multi-queue buffers for VLSI communication switches. IEEE Trans Computers 41(6):725–737
31. Martinez R, Alfaro FJ, Sanchez JL (2006) Decoupling the bandwidth and latency bounding for table-based schedulers. In: 2006 International Conference on Parallel Processing (ICPP'06), pp. 155–163. IEEE
32. Alfaro FJ, Sánchez JL, Duato J (2004) Qos in infiniBand subnetworks. IEEE Trans Parallel Distrib Syst 15(9):810–823
33. Martínez R, Alfaro FJ, Sánchez JL (2006) Improving the flexibility of the deficit table scheduler. In: International Conference on High-Performance Computing, pp. 84–97. Springer
34. Kahan W (1996) IEEE standard 754 for binary floating-point arithmetic. Lect Notes Status IEEE 754(94720–1776):11
35. Tyagi A, Muppala JK, De Meer H (2000) VoIP support on differentiated services using expedited forwarding. In: Conference Proceedings of the 2000 IEEE International Performance, Computing, and Communications Conference (Cat. No. 00CH37086), pp. 574–580. IEEE
36. Wenger S (2003) H. 264/avc over IP. IEEE transactions on circuits and systems for video technology 13(7), 645–656
37. NAMD Homepage. https://www.ks.uiuc.edu/Research/namd/utilities/. (Accessed July 6, 2021)
38. Van Der Spoel D, Lindahl E, Hess B, Groenhof G, Mark AE, Berendsen HJ (2005) Gromacs: fast, flexible, and free. J Comput Chem 26(16):1701–1718
39. Luszczek PR, Bailey DH, Dongarra JJ, Kepner J, Lucas, RF, Rabenseifner R, Takahashi D (2006) The hpc challenge (hpcc) benchmark suite. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, vol. 213, pp. 1188455–1188677

Authors and Affiliations

Javier Cano-Cano[1] · Francisco J. Andújar[2] · Francisco J. Alfaro-Cortés[1] · José L. Sánchez[1] · Gaspar Mora[3]

Francisco J. Andújar
fandujarm@infor.uva.es

Francisco J. Alfaro-Cortés
Fco.Alfaro@uclm.es

José L. Sánchez
Jose.SGarcia@uclm.es

Gaspar Mora
gaspar.mora.porta@intel.com

[1] Computing System Department, Universidad de Castilla-La Mancha, Campus Universitario s/n, 02071 Albacete, Spain

[2] Computing System Department, Universidad de Valladolid, Campus Miguel Delibes, Paseo de Belén, n 15, 47011 Valladolid, Castilla y León, Spain

[3] Computing System Department, Intel Corporation, Santa Clara, California, USA