

# TDM Virtual-Circuit Configuration for Network-on-Chip

Zhonghai Lu, *Member, IEEE*, and Axel Jantsch, *Member, IEEE*  
 Email: {zhonghai,axel}@kth.se

**Abstract**—In Network-on-Chip (NoC), Time-Division-Multiplexing (TDM) Virtual Circuits (VCs) have been proposed to satisfy the Quality-of-Service requirements of applications. TDM VC is a connection-oriented communication service by which two or more connections take turns to share buffers and link bandwidth using dedicated time slots. In the paper, we first give a formulation of the multi-node VC configuration problem for arbitrary NoC topologies. A multi-node VC allows multiple source and destination nodes on it. Then we address the two problems of *path selection* and *slot allocation* for TDM VC configuration. For the path selection, we use a back-tracking algorithm to explore the path diversity, constructively searching the solution space. In the slot allocation phase, overlapped VCs must be configured such that no conflict occurs and their bandwidth requirements are satisfied. We define the concept of a logical network (LN) as an infinite set of associated (time slot, buffer) pairs with respect to a buffer on a given VC. Based on this concept, we develop and prove theorems that constitute sufficient and necessary conditions to establish conflict-free VCs. They are applicable for networks where all nodes operate with the same clock frequency but allowing different phases. Using these theorems, slot allocation for VCs is a procedure of assigning VCs to different LNs. TDM VC configuration can thus be predictable and correct-by-construction. Our experiments on synthetic and real applications validate the effectiveness and efficiency of our approach.

**Index Terms**—Time Division Multiplexing, Virtual Circuit, Logical Network, Quality of Service, Network-on-Chip

## I. INTRODUCTION

TECHNOLOGY capacity and application complexity have driven bus-based system-on-chip (SoC) towards network-on-chip (NoC). NoC provides a network as a global multi-hop communication platform to allow communications between network nodes. A node is a local computing or storage region, which may contain a processor, a memory, an IP, a configurable hardware or a bus-based subsystem. More importantly, each node contains a switch or router. In the network, node-to-node communication signals are routed as packets via switches instead of being hard-wired. An on-chip network has the potential to solve the scalability problem of buses in bandwidth, clocking frequency and power. However, due to contention for shared links and buffers in the network, routing packets brings about unpredictable performance. To overcome the nondeterminism, researchers proposed various *resource reservation* and *priority-based scheduling* mechanisms to achieve Quality of Service (QoS), i.e., to provide guarantees in latency and bandwidth. The *Æthereal* [1] and *Nostrum* [2] adopt

the resource-reservation strategy to offer guaranteed services. Specifically, both establish *Time-Division-Multiplexing (TDM) virtual circuits (VCs)* in networks operating synchronously. Packets on a VC are consecutively transmitted in a pipeline by the node switches along the VC route. The *Æthereal* VC, which is developed for a network using buffered flow control, is *open-ended*. The *Nostrum* VC, which is designed for a network employing bufferless flow control, is *closed-loop*. The *Mango* [3] NoC realizes guarantees in an asynchronous (clock-less) network by preserving virtual channels for end-to-end connections and using priority-based scheduling in favor of connections in switches. Alternatively, QoS may be achieved through traffic classification in combination with a differentiated service. For example, the *QNoC* [4] characterizes traffic into four priority classes, and switches make priority-based switching decisions. Using resource reservation strategies can achieve hard performance guarantees but resource utilization may be lower because reserved resources may not be fully utilized. In contrast, priority-based schemes can achieve better resource utilization because resources are used on demand in a best-effort fashion, but they can only achieve soft performance guarantees.

In the paper, we focus on the TDM VCs and address the multi-node TDM VC configuration problem. VC is a connection-oriented communication service between communicating entities in a packet-switched network. TDM VC means two or more connections take turns to use shared buffers and link bandwidth using dedicated time slots. In the connection setup or configuration phase, a deterministic path is established and associated resources are pre-allocated. Specifically, each node along a VC route must configure a time-sliced routing table to reserve time slots for input packets to use output links. In this way, VCs multiplex link bandwidth in a time division fashion. As long as a VC is established, packets sent over it, called *VC packets*, encounter no contention and thus have guarantees in latency and bandwidth. In a network delivering both Best-Effort (BE) and guaranteed-service traffic, BE packets utilize resources that are not reserved by VCs. In case the requested resources are not available, the BE packets are buffered if the network uses drop-less buffered flow control. If the network realizes bufferless flow control without dropping packets, the packets are deflected to unfavored links. A VC is simplex, passing at least one source node and one destination node. But, in general, a VC may comprise multiple source and destination nodes (multi-node). In fact, it is important to construct multi-node VCs in order to use network bandwidth more efficiently.

VC configuration is an indispensable process in the application design flow because the establishment of VCs is the precondition for the guaranteed service. Besides, well-planned VCs (for instance, multi-node VCs and minimal routes) can make a better utilization of network resources and achieve better network performance. Configuring VCs involves (1) *path selection*: This has to explore the network path diversity. As a VC has a number of alternative paths, configuring a set of VCs involves an extremely large design space. The space is exponentially increased with the number of VCs; (2) *slot allocation*: Since VC packets can not contend with each other, VCs must be configured so that an output link of a switch is allocated to one VC per slot. Both steps together must ensure that VCs are contention free and equipped with sufficient slots. The network must be free from deadlock and livelock.

Current approaches to the TDM VC configuration problem can be found in [5] and [6], where the traffic model assumes periodic messages and all message flows have the same period. These approaches have three major drawbacks: (1) the techniques of contention avoidance and slot allocation are somewhat ad hoc. Contention is avoided mainly by locally scheduling available slots to a set of sorted VCs one by one. The resulting configurations are sufficient but not necessary. In case of lack of slots along a VC's shortest path, the VC uses a detour, i.e., non-minimal path. To exploit flexible routing in a network to make more VCs configurable, messages within a flow are scheduled individually and may use different routes [6]. Consequently, the message scheduling has also to ensure the correct message ordering; (2) the proposed heuristic algorithms are greedy, exploring only a very small subset of the solution space. In effect, this saves run time but wastes effectiveness. The heuristics limit the number of possible solutions, and for many problems, they can not find a solution or lead to over-dimensioning of the network; (3) a VC specification allows only one source node and one destination node. This makes the network utilization less efficient. Our work addresses these problems. For the first problem, we resort to a formal approach by utilizing the generalized concept of a *Logical Network (LN)* and developing sufficient and necessary conditions to guide the construction of conflict-free VCs. For the second problem, we use a standard technique to systematically search the solution space and constrain paths to minimal routes. For the third problem, we consider a VC specification with multiple source and destination nodes.

The rest of the paper is organized as follows. We outline the related work in Section II, detailing the weakness of the current approaches. In Section III, we first describe the network operation model with TDM VCs and then exemplify the two types of TDM VCs proposed for NoCs, namely, *open-ended* and *close-looped* VCs. Using LNs to construct contention-free, bandwidth-satisfied VCs is exemplified in Section IV. Then we develop sufficient and necessary conditions and prove the theorems for overlapping VCs to be contention-free in Section V. Section VI formulates the multi-node VC configuration problem. In Section VII, we detail the multi-node VC configuration method including the backtracking algorithm. Experimental results are reported in Section VIII. Finally we conclude the paper in Section IX.

## II. RELATED WORK

TDM techniques were proposed in optimizing on-chip bus protocols [7] and in bus-based distributed systems [8]. Wingard et al. [7] proposed a TDM-based interleaved bus coupled with fixed transfer latency and clock alignment, providing bandwidth and latency guarantees to IP cores. In [8], the TDM-based bus access scheme was used for the design of distributed hard real-time applications that require predictability and reliability.

The TDM-based VC configuration is a general problem for reserving time slots to provide QoS in NoCs. As we mentioned previously, possible solutions can be found in [5] and [6]. The UMARS (Unified MAPPING, Routing and Slot allocation) [5] is a single-objective algorithm unifying the IP-to-node mapping, path selection and slot allocation. It is a greedy algorithm that iterates over a monotonically decreasing set of unmapped VCs until all VCs are allocated or until allocation fails. The outer loop of UMARS consists of three steps: (1) select the VC with the highest bandwidth; (2) find a mapping and a path; (3) allocate slots on this path. The second step is guided by a cost function and finds *one locally optimal* path. Although it is fast, it leaves a wide solution space unexplored. Stuijt et al. analyzed the necessity of exploring wider solution space in [6]. Their experiments show that, if re-consideration of the path (path alternative) is allowed, the number of solvable problems in the synthetic benchmark set is improved 209%. However, the improved algorithm is still greedy allowing a limited number of path alternatives. They also demonstrated the significance of performing a proper contention avoidance strategy. Before scheduling message streams, they try to estimate the number of slots that are needed in each of the links. This knowledge of congestion on links is used to guide the path selection process. Together with a slot-sharing allocation strategy, this improves over UMARS by 334% with a small overhead on the run time. Both algorithms in [5] and [6] allow non-minimal routes to increase the solvable problem size. However, permitting detour consumes additional buffering and link bandwidth. It wastes power and may degrade the overall system performance because it affects other traffic in the network. In our opinion, detour can only be justified if the solution space is sufficiently searched. This is particularly true if VC configurations are derived off-line. Furthermore, both works consider VCs only for unicast (one-to-one communication). Therefore they can not efficiently support other communication patterns such as one-to-many, many-to-one, and combinations of these patterns.

We have introduced the concept of LN and presented the LN-based slot allocation method in [9]. The LN concept generalizes the concepts of *admission class* [11] and *Temporally Disjoint Network (TDN)* [2]. Both admission classes and TDNs were proposed to investigate the packet contention problem in deflection-routed networks. They are essentially LNs. Comparing with an admission class, a LN takes not only time slots but also the VC path into consideration, thus the VC path-overlapping scenarios can be studied. Comparing with a TDN, a LN is locally defined for a group of overlapping VCs with both open-ended and closed-loop paths. A TDN can

be viewed as a special case of a LN in the closed-loop VC when a LN is globally set up for all overlapping and non-overlapping VCs. In Section III, when discussing the open-ended VCs, we will introduce a *target class* as a special case of the admission class, and when introducing the closed-loop VCs, we will describe TDNs in detail.

In this paper, we develop the LN concepts further and give a complete account of the VC configuration problem addressing and integrating both path selection and slot allocation. We give a formulation of the multi-node VC configuration problem. In the path selection, we address the node visiting order for multi-node VCs and introduce the back-tracking algorithm for VC path enumeration. Moreover, our configuration method is substantiated with more and detailed experimental results.

### III. TDM-BASED VIRTUAL CIRCUITS IN NoC

#### A. The System Operational Model

The on-chip TDM VCs assume that the network is packet-switched and time-slotted. The network nodes share the same notion of time. They have the same frequency but not phase [10]. VC packet routing is performed by using statically configured routing tables in nodes along the VC path. The entries for the routing tables are globally orchestrated such that no simultaneous use of shared resources is possible. Thus no run-time scheduling is necessary.

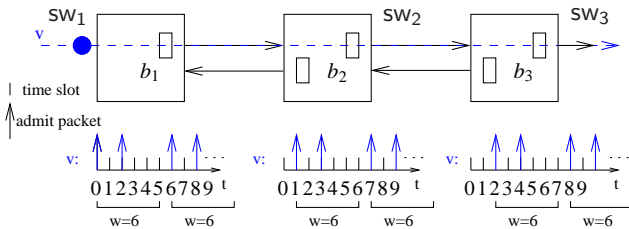


Fig. 1. An example of the system operation

Packets on a VC are admitted into the network in a disciplined way. A certain number of packets are admitted in pre-calculated slots within a given window. This forms an admission pattern that is repeated without change throughout the system execution. In the network, VC packets synchronously advance one hop per time slot. Since a VC packet encounters no contention, it never stalls, using consecutive slots in consecutive switches. As illustrated in Figure 1, VC  $v$  passes switches  $sw_1$ ,  $sw_2$  and  $sw_3$  through  $\{b_1 \rightarrow b_2 \rightarrow b_3\}$ . On  $v$ , two packets are injected into the network every six slots (we say that the window size is 6). Initially, the slots of buffer  $b_1$  at the first switch  $sw_1$  occupied by the packet flow are 0 and 2. Afterwards, this pattern repeats,  $b_1$ 's slots 6 and 8, 12 and 14, and so on are taken. In the second switch  $sw_2$ , the packets occupy  $b_2$ 's slots 1 and 3, 7 and 9, and 13 and 15, and so on. In switch  $sw_3$ , they occupy  $b_3$ 's slots 2 and 4, 8 and 10, and 14 and 16, and so on.

It is the task of VC configuration to determine the exact admission pattern for VC packets and configure the routing tables in nodes. The routing table, by configuration, knows the *time slot* when a VC packet reaches which inport, and *addressing information* about which output to use. In effect, the routing

function partitions the link bandwidth and avoids contention. In implementation, the routing tables may be implemented in the switches. In this case, a VC packet does not need to carry routing information in its head field. Alternatively, the routing tables can be removed from the switches, and the addressing information is embedded in a VC packet. In this case, the information about which time the switches should reserve slots for a specific VC are maintained by the network interface wrapping the switches. This saves area at the expense of communication overhead in each VC packet.

#### B. Open-ended VCs and Target Classes

Figure 2 shows two VCs,  $v_1$  and  $v_2$ , and the respective routing tables for the switches. The output links of a switch are associated with a buffer or register. A routing table  $(t, in, out)$  is equivalent to a routing or slot allocation function  $\mathcal{R}(t, in) = out$ , where  $t$  is time slot,  $in$  an input link, and  $out$  an output link.  $v_1$  passes switches  $sw_1$  and  $sw_2$  through  $\{b_1 \rightarrow b_2\}$ ;  $v_2$  passes switches  $sw_3$  and  $sw_2$  through  $\{b_3 \rightarrow b_2\}$ . The  $\mathcal{A}$ ethereal NoC [1] proposes this type of VC for QoS. Since the path of such a VC is not a loop, we call it an open-ended VC.

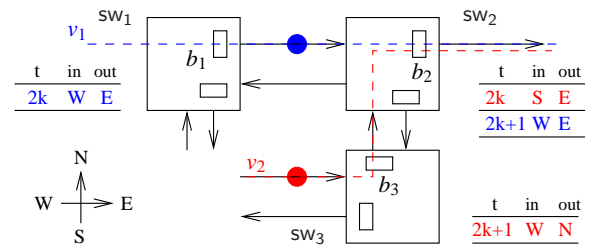


Fig. 2. Open-ended virtual circuits

In open-ended VCs, packets may be partitioned into *target classes* to avoid contention. With respect to a buffer  $b$ , a target class is the set of packets that will occupy slot  $d$  in a slot window  $D$ . This set of packets may come from any network node as long as they will take slot  $d$  in a slot window  $D$  of buffer  $b$ . The formal definition of target class will be given in Definition 2 in Section V. As a target class owns dedicated slots with respect to buffers, packets of different classes do not collide in buffers.

A target class is a special case of an admission class [11] in the sense that a target class has a reference buffer whereas an admission class does not. The union of all the target classes for all buffers in the network gives the corresponding admission class. By globally orchestrating the packet admission, contention can be avoided for packets belonging to different VCs. As illustrated in Figure 2,  $v_1$  and  $v_2$  only overlap in  $b_2$ , denoted  $v_1 \cap v_2 = \{b_2\}$ .  $v_1$  packets are admitted on even slots of  $b_1$ . In  $sw_1$ ,  $(2k, W, E)$  means that  $sw_1$  reserves its  $E$  (East) output link at slots  $2k$  ( $\forall k \in \mathbb{N}$ ) for its  $W$  (West) inport ( $\mathcal{R}(2k, W) = E$ ). As we can also see,  $v_2$  packets are admitted on odd slots  $2k+1$  of  $b_3$ , and  $sw_3$  configures its odd slots for  $v_2$ . Since a  $v_1$  packet reaches  $sw_2$  one slot after reaching  $sw_1$ ,  $sw_2$  assigns its odd slots to  $v_1$ . Similarly,  $sw_2$  allocates its even slots to  $v_2$ . As  $v_1$  and  $v_2$  interleave on the use of the shared buffer  $b_2$  and its associated output link,  $v_1$  and  $v_2$  do not conflict.

### C. Closed-loop VCs with Containers and TDNs

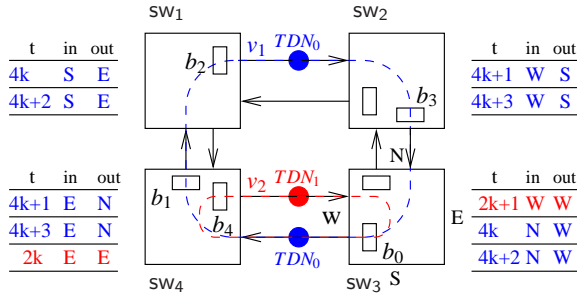


Fig. 3. Closed-loop virtual circuits

The Nostrum NoC [2] also suggests TDM VC for QoS. However, a Nostrum VC has a cyclic path, i.e., a closed loop. On the loop, at least one *container* is rotated. A container is a *special packet* used to carry data packets, like a vehicle carrying passengers. The reason to have a loop is due to the fact that Nostrum uses deflection routing [12] whereas switches have no buffer queues. If a packet arrives at a switch and is not sunk, it must be switched out using one of the switch's output. A Nostrum switch has  $k+1$  inports/outports,  $k$  of which are connected to other switches and one of which is a local duplex port for admitting/sinking packets into/from the network. If  $k$  network packets, none of which reaches its destination, are coming to a switch, all the  $k$  output links will be occupied. At this time, if a local packet is scheduled to be admitted into the network, this situation makes the admission of the packet impossible because there is no output link available. This problem is solved by a looped container. The looped container ensures that there is always an output link available for locally admitting a VC packet into the container and thus the network. VC packets are loaded into the container from a source, and copied (for multicast) or unloaded at the destination, by-passing other switches. Similarly to open-ended VCs, containers as VC packet carriers have higher priority than BE packets and do not contend with each other.

The Nostrum VC [2] uses TDNs to ensure conflict freedom. In [2], TDNs are descriptively rather than formally defined. A TDN consists of a set of regularly partitioned buffers in the network. Packets launched on one TDN never meet those packets launched on another TDN because they always have a different distance to any buffer in the network. Therefore those sets of buffers constitute temporally disjoint networks in the same physical network. One may think that TDNs are the result of coloring network buffers using the least number of colors in such a way that no adjacent buffers have the same color. Buffers with the same color can be viewed as one TDN. The number of colors used is that of TDNs. Apparently, TDNs are globally set up in a network, independent of VC paths. According to [2], the number of TDNs depends on the network topology and the buffer stages in the switches. For example, in a mesh network with one buffer per output in the switches, exactly two TDNs exist,  $TDN_0$  and  $TDN_1$ . To allow more TDNs, more buffers in the switches must be used. For example, placing two buffers in the switches, one at the inport, the other at the output, results in four TDNs. As shown in

Figure 3, two VCs,  $v_1$  and  $v_2$ , are configured.  $v_1$  loops on  $sw_3$ ,  $sw_4$ ,  $sw_1$  and  $sw_2$  through  $\{b_0 \rightarrow b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow b_0\}$ ;  $v_2$  loops on  $sw_3$  and  $sw_4$  through  $\{b_0 \rightarrow b_4 \rightarrow b_0\}$ ; and  $v_1 \cap v_2 = \{b_0\}$ .  $v_1$  and  $v_2$  subscribe to  $TDN_0$  and  $TDN_1$ , respectively. Besides,  $v_1$  launches two containers and  $v_2$  one container. The resulting routing tables for switches are also shown in Figure 3. Since TDNs are temporally disjoint, overlapping VCs allocated on different TDNs are free from conflict.

## IV. VC CONFIGURATION USING LNS

### A. An Overview of Multi-node VC Configuration

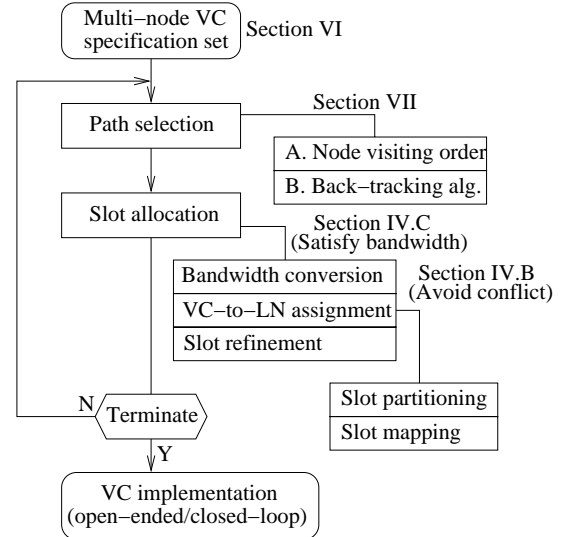


Fig. 4. The multi-node VC configuration flow

Figure 4 sketches our multi-node VC configuration approach. The input to the flow is a VC specification set. The output is a set of TDM VC implementations, one implementation for one specification. The implementation can be either open-ended or close-looped. The configuration consists of two sequential phases: *path selection* and *slot allocation*. In the first phase, paths for VCs are determined. In the second phase, based on the VC paths, slots are allocated to VCs to see if conflict can be avoided and bandwidth can be satisfied. In order to meet the conflict-free and bandwidth requirements, the two-phase procedure is iterative as realized in Algorithm 1 in Section VII.

In the paper, we address both *path selection* and *slot allocation*. The problem formulation is given in Section VI. For the path selection, we first discuss the node visiting order in a multi-node VC. This problem arises when a VC specification contains multiple source and destination nodes and only minimal routes are desired. A multi-node VC can support not only unicast/multicast traffic but also allows to merge multiple unicast/multicast traffic flows (Refer to Section VIII.B for examples). One may logically think of a multi-node VC as a bus, since it allows different kinds of communication patterns. We utilize a back-tracking algorithm to explore the VC path diversity in the network. The path selection is addressed in Section VII. For the slot allocation, we generalize and use

the concept of a LN. This involves *avoiding conflict between overlapping VCs* and *satisfying VC bandwidth requirements*.

In the following of this section, assuming that the path selection is done, we exemplify the LN construction for conflict freedom in Section IV-B. Then we show how to satisfy bandwidth demand using LNs in Section IV-C. We shall see that our method is applicable to both *open-ended* and *closed-loop* VCs. By using the examples, we explain the LN concept and its associated methods intuitively. After that, we present formal definitions and proofs for the conflict freedom in Section V.

### B. Avoid Conflict Using LNs

To convey the basic ideas of a LN, we describe how conflict can be avoided between overlapping VCs by alternatively scheduling VCs on the use of the shared buffer(s). The formal definition of LN will be given in Definition 4 in Section V. As we develop further, we shall see that LNs are the natural result of systematically avoiding collision between overlapping VCs. To be specific, the conflict avoidance is assured through two steps: *slot partitioning* and *slot mapping*. These two steps create LNs and complete assigning VCs to LNs. We describe them with a pair of *closed-loop* VCs ( $v_1, v_2$ ) in Figure 3.

- 1) *Slot partitioning*: As conflicts might occur in a shared buffer, we partition the slots of the shared buffer into slot sets with a regular interval. In Figure 3,  $b_0$  is the only shared buffer of  $v_1$  and  $v_2$ ,  $v_1 \cap v_2 = \{b_0\}$ . We partition the slots of  $b_0$  ( $b_0$  is called the *reference buffer* for  $v_1$  and  $v_2$ ,  $Ref(v_1, v_2) = b_0$ .) into two sets, an even set  $s_0^2(b_0)$  for  $t = 2k$  and an odd set  $s_1^2(b_0)$  for  $t = 2k + 1$ . The notation  $s_\tau^T(b)$  represents pairs  $(\tau + kT, b)$ , which is the  $\tau$ th slot set of the total  $T$  slot sets,  $\forall k \in \mathbb{N}$ ,  $\tau \in [0, T)$  and  $T \in \mathbb{N}$ . The pair  $(t, b)$  refers to the slot of  $b$  at time instant  $t$ . Notation  $s_{\tau_1, \tau_2, \dots, \tau_n}^T(b)$  collectively represents a set of pair sets  $\{(\tau_1 + kT, b), (\tau_2 + kT, b), \dots, (\tau_n + kT, b)\}$ .

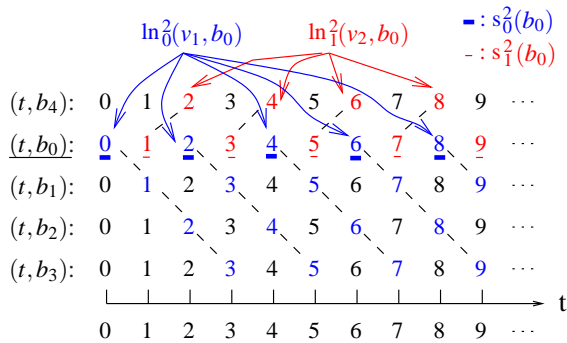


Fig. 5. Creating LNs by mapping slots on VCs

- 2) *Slot mapping*: The partitioned slot sets can be mapped to slot sets of other buffers on a VC regularly and unambiguously because a VC packet or container advances one hop along the VC path each and every slot. In this way, the slot partitionings are propagated to other buffers on the VC. For example, a  $v_1$  packet holding slot  $t$  at buffer  $b_0$ , i.e., pair  $(t, b_0)$ , will consecutively take slot  $t + 1$  at  $b_1$  (pair  $(t + 1, b_1)$ ), slot  $t + 2$  at  $b_2$  (pair

$(t + 2, b_2)$ ), and slot  $t + 3$  at  $b_3$  (pair  $(t + 3, b_3)$ ). After mapping the slot set  $s_0^2(b_0)$  on  $v_1$  and  $s_1^2(b_0)$  on  $v_2$ , we obtain two slot sets  $\{s_0^2(b_0), s_1^2(b_1), s_0^2(b_2), s_1^2(b_3)\}$  and  $\{s_1^2(b_0), s_0^2(b_4)\}$ . We refer to the logically networked slot sets in a set of buffers of a VC as a *LN*. Thus a LN is a composition of associated (*time slot, buffer*) pairs on a VC with respect to a buffer. We denote the two LNs as  $ln_0^2(v_1, b_0)$  and  $ln_1^2(v_2, b_0)$ , respectively. The notation  $ln_\tau^T(v, b)$  represents the  $\tau$ th LN of the total  $T$  LNs on  $v$  with respect to  $b$ . We illustrate the mapped slot sets for  $s_0^2(b_0)$  and  $s_1^2(b_0)$  and the resulting LNs in Figure 5. We can also see that LNs are the result of VC assignment to slot sets, and a LN is a function of a VC. In our case,  $v_1$  subscribes to  $ln_0^2(v_1, b_0)$  and  $v_2$  to  $ln_1^2(v_2, b_0)$ .

As  $ln_0^2(v_1, b_0) \cap ln_1^2(v_2, b_0) = \emptyset$ ,  $v_1$  and  $v_2$  are conflict free, as we shall prove formally in Section V.

### C. Satisfy Bandwidth Using LNs

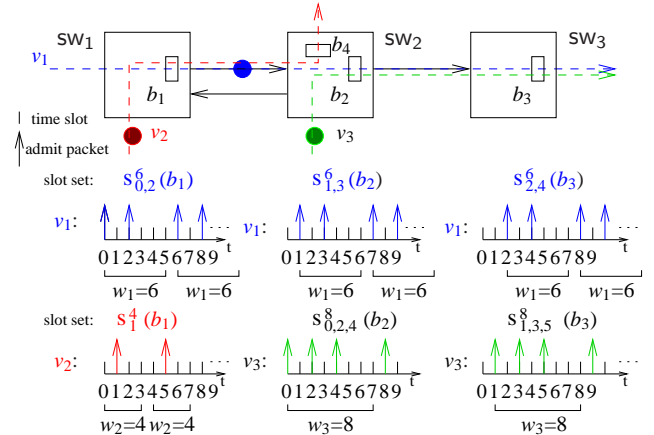


Fig. 6. Packets admitted on slot sets of buffers, i.e., on LNs

VC	Buf. set	bw	N	W	LN	Slot set
$v_1$	$b_1, b_2, b_3$	1/3	2	6	$ln_0^2(v_1, b_1)$	$\{s_{0,2}^6(b_1), s_{1,3}^6(b_2), s_{2,4}^6(b_3)\}$
$v_2$	$b_1, b_4$	1/4	1	4	$ln_1^2(v_2, b_1)$	$\{s_{1,3,5}^4(b_1), s_{0,4}^4(b_4)\}$
$v_3$	$b_2, b_3$	3/8	3	8	$ln_0^2(v_3, b_2)$	$\{s_{0,2,4}^8(b_2), s_{1,3,5}^8(b_3)\}$

TABLE I

VC PARAMETERS AND VC-TO-LN ASSIGNMENT RESULTS FOR FIG. 6. THE BANDWIDTH  $bw$  IS GIVEN IN PACKETS/SLOT.  $N$  IS THE NUMBER OF PACKETS ADMITTED WITHIN  $W$  SLOTS.  $W$  IS THE WINDOW SIZE.

In addition to be contention free, VCs must satisfy their bandwidth requirements. This is achieved in three steps: *bandwidth conversion*, *VC-to-LN assignment* and *slot refinement*. We exemplify the three steps with Figure 6 that shows three *open-ended* VCs,  $v_1$ ,  $v_2$  and  $v_3$ . As can be seen,  $v_1 \cap v_2 = \{b_1\}$ ,  $v_1 \cap v_3 = \{b_2, b_3\}$  and  $v_2 \cap v_3 = \emptyset$ .

- 1) *Bandwidth conversion*: A VC is associated with a bandwidth requirement in bits/second, which can be directly translated into packets/slot. As bandwidth is an average measurement, we can further scale it to the number  $N$  of packets per  $W$  slots.  $W$  is the window size. For example,

we translate  $bw_1 = 1/3$  packets/slot into  $2/6$  packets/slot (2 packets every 6 slots), i.e.,  $N_1 = 2$ ,  $W_1 = 6$ , as listed in Table I.

- 2) *VC-to-LN assignment*: We assign VCs to LNs using the two steps for *conflict avoidance* in Section IV-B. Additionally we must check whether their bandwidth demand can be satisfied. This check is conducted after the first step *slot partitioning*. Given a pair of overlapping VCs, the number  $T$  of partitioned sets with respect to the reference buffer equals the number of LNs. To satisfy the bandwidth requirement of a VC  $v$ , a sufficient number  $N_{in}$  of LNs must be allocated to  $v$ . This number can be derived from  $N_{in} = \lceil NT/W \rceil^1$ , because we must satisfy  $N_{in}/T \geq N/W$ , where  $N_{in}/T$  is the bandwidth supported by the allocated LNs and  $N/W$  the requested bandwidth. The bandwidth requirements of the three VCs in Figure 6 are given in column  $bw$  of Table I.

We first perform VC-to-LN assignment with VC pair  $(v_1, v_2)$ . Since  $b_1$  is the only shared buffer of  $(v_1, v_2)$ ,  $Ref(v_1, v_2) = b_1$ . Let  $T = 2$ , we partition  $b_1$ 's slots into odd and even sets, implying two LNs. Either VC can be allocated to one LN, i.e.,  $N_{in,1} = N_{in,2} = 1$ , offering bandwidth  $N_{in,1}/T = N_{in,2}/T = 1/2$ . Since the bandwidth demand of  $v_1$  and  $v_2$  is less than  $1/2$ , the resulting VC-to-LN assignment will meet the bandwidth constraint. Then we can continue to map the even set on  $v_1$  and the odd set on  $v_2$ , obtaining the even LN  $ln_0^2(v_1, b_1)$  for  $v_1$  and the odd LN  $ln_1^2(v_2, b_1)$  for  $v_2$ . Since  $ln_0^2(v_1, b_1) \cap ln_1^2(v_2, b_1) = \emptyset$ ,  $v_1$  and  $v_2$  are conflict free. Next, we perform VC-to-LN assignment with VC pair  $(v_1, v_3)$ . Let the reference buffer of  $v_1$  and  $v_3$  be  $b_2$ ,  $Ref(v_1, v_3) = b_2$ . Since  $v_1$  already holds even slots in  $b_1$ , it takes odd slots in  $b_2$ , i.e.,  $s_1^2(b_2)$ . We assign the remaining even slots in  $b_2$ , i.e.,  $s_0^2(b_2)$ , to  $v_3$ . Therefore,  $N_{in,3}/T = 1/2 > 3/8$ . We are certain that the supported bandwidth suffices the demand of  $v_3$ . We map the slot set  $s_0^2(b_2)$  on  $v_3$ , obtaining  $ln_0^2(v_3, b_2)$ . As  $ln_0^2(v_1, b_1) \cap ln_0^2(v_3, b_2) = \emptyset$ ,  $v_1$  and  $v_3$  are also conflict free. The VC-to-LN assignment results are shown in column LN of Table I.

- 3) *Slot refinement*: The success of VC-to-LN assignment for all VCs means that all VCs are conflict free and enough bandwidth can be reserved. But, a VC may demand only a fraction of slot sets from its assigned LNs. For instance, " $v_2$  on  $ln_1^2(v_2, b_1)$ " means that  $v_2$  can use one of every two slots. But  $N_2 = 1$  and  $W_2 = 4$ ,  $v_2$  actually demands only one out of four slots. This means that we need to further refine the supplied bandwidth. We first find the candidate slot sets of a reference buffer and then only assign  $N$  of them within window size  $W$  to  $v$ . For example,  $v_3$  has four candidate slot sets over  $b_2$ ,  $s_{0,2,4,6}^8(b_2)$ . We allocate any three of the four to  $v_3$ , for instance,  $s_{0,2,4}^8(b_2)$ . These slot sets are mapped to  $s_{1,3,5}^8(b_3)$ , forming the LN  $ln_0^2(v_3, b_2)$ . The slot sets reserved by the three VCs are illustrated in Figure 6 and listed in the column Slot set of Table I.

After the three steps above, the VCs are constructed without conflict and with bandwidth requirements satisfied. In the following, we do not treat the *Slot refinement* as a separate step. Instead we consider it as the last step of *VC-to-LN assignment* to make the presentation concise.

#### D. Requirements for LN-oriented VC Configuration

We have described so far three techniques: (1) establishing VCs by configuring slot-sliced routing tables; (2) partitioning and mapping slots into LNs; (3) assigning different LNs to VCs. These techniques must promise conflict freedom and provide enough bandwidth. However, there are several key questions that are not yet addressed:

- How many LNs exist when VCs overlap? LN is not global for all VCs. Instead it is local for a group of overlapping VCs. This number is crucial because it defines how to partition and then map slots.
- In the examples, assigning different LNs to overlapping VCs has secured conflict freedom. Is it a sufficient and necessary condition, in general?
- LN is partitioned with respect to a reference LN buffer, which is a shared buffer. As overlapping VCs may have more than one shared buffer, how is this reference buffer selected? Are LNs with respect to all shared buffers equivalent?

In the next section, we answer these questions formally.

### V. FORMAL ANALYSIS

#### A. Assumptions and Definitions

We consider static VCs, meaning that VCs do not change their paths and characteristics throughout system execution. We also assume that one LN is allocated to only one VC. But one VC may subscribe to multiple LNs.

*Definition 1*: A VC  $v$  comprises an ordered set of buffers  $\langle b_0, b_1, b_2, \dots, b_{H-1} \rangle$ . The size  $H$  of  $v$ , denoted by  $H = |v|$ , is the number of buffers.  $d_{b_i b_j}$  is the distance in number of slots<sup>2</sup> from  $b_i$  to  $b_j$ . On  $v$ ,  $d_{b_i b_{i+1}} = 1$ , meaning that the buffers are adjacent, taking one slot to advance a packet from  $b_i$  to  $b_{i+1}$ .

A VC packet or container  $p$  on  $v$ , admitted at slot  $t_0$ , starts from  $b_i$  and visits each buffer in sequence one per slot and never stalls. The *arrival time* of  $p$  at buffer  $b_j$  equals  $t_0 + d_{b_i b_j}$ .

*Definition 2*: The *admission pattern* on a VC, characterized by two integers  $N$  and  $D$ , with  $N \leq D$ , defines that  $N$  packets are admitted in every  $D$  slot period. This gives a bandwidth requirement of  $N/D$  packets/slot, but the exact time slots for admitting the  $N$  packets are not specified. A *packet flow* is defined by infinitely repeating the admission pattern. We call  $D$  the *admission cycle*. With respect to a buffer  $b$  and a natural  $d < D$ , we define a *target class* as an infinite set of packets that arrive at buffer  $b$  at slots  $d + kD$ ,  $\forall k \in \mathbb{N}$ . We call  $d$  the *initial 'distance'* of the target class to buffer  $b$ .

For an open-ended VC,  $D = W$ , where  $W$  is the window size of a VC packet flow; For a closed-loop VC,  $D = H$ .  $H$

<sup>1</sup> $\lceil x \rceil$  is the ceiling function that returns the least integer not less than  $x$ .

<sup>2</sup>As one hop takes one slot to travel, we equivalently measure the distance in number of slots.

is the length of the loop, since  $v$  is a loop and a container revisits the same buffer after  $H$  slots. In this case,  $N$  is the number of containers launched on the VC.

*Definition 3:* Two VCs  $v_1$  and  $v_2$  *overlap* if they share at least one buffer, i.e.,  $v_1 \cap v_2 \neq \emptyset$ . The two VCs *conflict* in buffer  $b$ , denoted  $b \in v_1 \wedge v_2$ , if and only if it is possible that two packets, one from each VC, visit buffer  $b$  at the same time.  $v_1 \wedge v_2 = \emptyset$  means that  $v_1$  and  $v_2$  are conflict free.

*Definition 4:* Given a VC  $v = \langle b_0, b_1, b_2, \dots, b_{H-1} \rangle$ ,  $b_i \in v$ , a natural  $1 \leq T \leq D$  ( $D$  is the admission cycle of the packet flow on  $v$ ) and a natural  $\tau$ ,  $0 \leq \tau < T$ , we define a LN  $ln_\tau^T(v, b_i)$  as an infinite set of (time slot, buffer) pairs as follows:

$$ln_\tau^T(v, b_i) = \{(t, b_j) | t = \tau + d_{b_i b_j} + kT, 0 \leq j < H, \forall k \in \mathbb{N}\}$$

Hence, a LN is defined for a given VC and one of its buffers.  $T$  determines the way the time slots are partitioned into LNs and gives the number of LNs. The motivation of the LN is to precisely define the flow of packets on the VC and each target class is dedicated to exactly one LN. The time when packets visit buffers of the VC is given by the (time slot, buffer) pairs of the LN. On a LN, a VC packet may visit a particular buffer every  $T$  slots. Consequently, the bandwidth possessed by a LN is  $1/T$  packets/slot.

The LNs of a VC have an inherent property: if  $\tau_1, \tau_2 \in [0, T-1]$  and  $\tau_1 \neq \tau_2$ , then packets admitted on different LNs never collide, because

$$ln_{\tau_1}^T(v, b) \cap ln_{\tau_2}^T(v, b) = \emptyset$$

*Definition 5:* A LN-cover is a complete set of LNs defined for a VC  $v$  with respect to a buffer  $b$ ,  $b \in v$ ,

$$LN\text{-cover}(v, b, T) = \{ln_\tau^T(v, b) | 0 \leq \tau < T\}$$

*Definition 6:* VC-to-LN assignment/subscription: a VC  $v$  is assigned to or subscribes to  $ln_\tau^T(v, b)$  if and only if, on  $v$ , a target class, which has an initial distance  $d$  to buffer  $b$  and the admission cycle  $D$ , satisfies  $mod(d + kD, T) = \tau$ ,  $\forall k \in \mathbb{N}$ .

If a VC  $v$  does not overlap with any other VCs, the maximum number of LNs on  $v$  is  $D$ , since  $v$  allows for up to  $D$  target classes and one class uses exactly one LN.

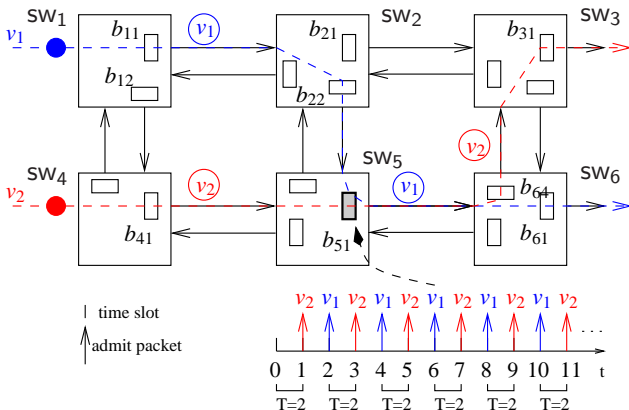


Fig. 7. LNs for  $v_1 = \langle b_{11}, b_{22}, b_{51}, b_{61} \rangle$  and  $v_2 = \langle b_{41}, b_{51}, b_{64}, b_{31} \rangle$  with respect to buffer  $b_{51}$ . It shows a snapshot for the links occupied by  $v_1$  and  $v_2$  at even slots, and  $v_1$  and  $v_2$  packets use the shared buffer  $b_{51}$  alternately.

To illustrate the above definitions, we give an example as shown in Figure 7, where  $v_1$  and  $v_2$  admit one packet every two slots;  $D_1 = D_2 = 2$ . Both VCs overlap in buffer  $b_{51}$ , resulting in two LNs,  $T = 2$ . As can be observed,  $v_1$  subscribes to  $ln_0^2(v_1, b_{51})$ , and  $v_2$  to  $ln_1^2(v_2, b_{51})$ . Therefore,  $v_1 \wedge v_2 = \emptyset$ . The figure shows that  $v_1$  and  $v_2$  packets alternately pass buffer  $b_{51}$  using even and odd slot sets, respectively. It also shows a snapshot at even slots for the links used by  $v_1$  and  $v_2$  packets.

## B. Overlapping VCs

*Lemma 1:* Let  $v_1$  and  $v_2$  be two overlapping VCs and  $D_1, D_2$  be their admission cycles, respectively. Let  $c_1$  and  $c_2$  be any two target classes on  $v_1$  and  $v_2$  with respect to a shared buffer  $b$ , respectively;  $d_1$  and  $d_2$  are the initial distances of  $c_1$  and  $c_2$  to buffer  $b$ , respectively. We have  $b \in v_1 \wedge v_2$  iff  $\exists k_1, k_2 \in \mathbb{N}$  such that  $d_1 + k_1 D_1 = d_2 + k_2 D_2$ .

*Proof:*

(1) Sufficient: We assume that  $\exists k_1, k_2 \in \mathbb{N}$  such that  $d_1 + k_1 D_1 = d_2 + k_2 D_2 (= t)$ . The left-hand side of the equation implies that  $c_1$  enters buffer  $b$  at time slot  $t$ , and the right-hand side implies that  $c_2$  enters  $b$  the same slot. Hence  $b \in v_1 \wedge v_2$ .

(2) Necessary: Suppose, after  $t$  slots,  $v_1$  and  $v_2$  collide in buffer  $b$ ,  $b \in v_1 \wedge v_2$ . For  $c_1$ ,  $t = d_1 + k_1 D_1$ ; for  $c_2$ ,  $t = d_2 + k_2 D_2$ . Therefore  $d_1 + k_1 D_1 = d_2 + k_2 D_2$ . ■

*Theorem 1:* Let  $T$  be the number of LNs, which two overlapping VCs,  $v_1$  and  $v_2$ , can subscribe to without conflict. Then  $T$  is a Common Factor (CF) of their admission cycles,  $D_1$  and  $D_2$ .

*Proof:* Suppose that  $b$  is the reference buffer.

Let  $ln_{\tau_1}^T(v_1, b)$  and  $ln_{\tau_2}^T(v_2, b)$  be the LN subscribed by  $v_1$  and  $v_2$ , respectively. According to Definition 6, we have  $\tau_1 = mod(d_1 + k_1 D_1, T)$  and  $\tau_2 = mod(d_2 + k_2 D_2, T)$ .

We start with  $\tau_1 = mod(d_1 + k_1 D_1, T)$ ,  $\forall k_1 \in \mathbb{N}$ . When  $k_1 = 0$ ,  $d_1 = k_1' T + \tau_1$ ; when  $k_1 = 1$ ,  $d_1 + D_1 = k_1'' T + \tau_1$  and  $k_1'' > k_1'$ . From the last two equations, we get  $D_1 = (k_1'' - k_1') T$ . This means that  $T$  is a factor of  $D_1$ .

Similarly, using  $\tau_2 = mod(d_2 + k_2 D_2, T)$ ,  $\forall k_2 \in \mathbb{N}$ , we can derive that  $T$  is a factor of  $D_2$ .

Therefore  $T$  is a common factor of  $D_1$  and  $D_2$ , i.e.,  $T \in CF(D_1, D_2)$ . ■

By Theorem 1, the number  $T$  of LNs for  $v_1$  and  $v_2$  can be any value in the common factor set  $CF(D_1, D_2)$ . The least number of LNs is 1. However, if the number of LNs for two VCs is 1, only one of the two VCs can subscribe to it. There is no room for the other VC. Therefore we need at least two LNs. In general, if  $n$  VCs overlap in a shared buffer, there must be at least  $n$  LNs, one for each VC, to avoid conflict. In order to maximize the number of options and have finer LN bandwidth granularity, we consider the number  $T$  of LNs to be the Greatest Common Divisor (GCD) throughout the paper. Hence, for the two overlapping VCs,  $v_1$  and  $v_2$ , the number  $T$  of LNs equals the  $GCD(D_1, D_2)$ .

*Theorem 2:* Assigning  $v_1$  and  $v_2$  to different LNs with respect to any shared buffer is a sufficient and necessary condition to avoid conflict between  $v_1$  and  $v_2$ .

*Proof:* By Theorem 1, the maximum number  $T$  of LNs for  $v_1$  and  $v_2$  is  $T = \text{GCD}(D_1, D_2)$ . We can write  $D_1 = A_1T$  and  $D_2 = A_2T$ , where  $A_1$  and  $A_2$  are co-prime.

By Definition 6,  $v_1$  and  $v_2$  subscribe to different LNs  $\Leftrightarrow \text{mod}(d_1 + k_1D_1, T) \neq \text{mod}(d_2 + k_2D_2, T)$ . Since  $D_1 = A_1T$  and  $D_2 = A_2T$ ,  $\text{mod}(d_1 + k_1D_1, T) \neq \text{mod}(d_2 + k_2D_2, T) \Leftrightarrow \text{mod}(d_1, T) \neq \text{mod}(d_2, T)$ .

(1) Sufficient:  $\text{mod}(d_1, T) \neq \text{mod}(d_2, T) \Rightarrow d_1 + k'_1T \neq d_2 + k'_2T, \forall k'_1, k'_2 \in \mathbb{N}$ . When  $k'_1 = k_1A_1$  and  $k'_2 = k_2A_2, \forall k_1, k_2 \in \mathbb{N} \Rightarrow d_1 + k_1A_1T \neq d_2 + k_2A_2T \Rightarrow d_1 + k_1D_1 \neq d_2 + k_2D_2$ . According to Lemma 1,  $v_1$  and  $v_2$  do not conflict, i.e.,  $v_1 \wedge v_2 = \emptyset$ .

(2) Necessary: Suppose  $v_1 \wedge v_2 = \emptyset \Rightarrow d_1 + k_1D_1 \neq d_2 + k_2D_2, \forall k_1, k_2 \in \mathbb{N}$ . But let us assume  $\text{mod}(d_1, T) = \text{mod}(d_2, T)$ . Then we have  $d_1 - d_2 \neq k_2D_2 - k_1D_1$  but  $d_1 - d_2 = kT, k \in \mathbb{Z} \Rightarrow k + k_1A_1 \neq k_2A_2, \forall k_1, k_2 \in \mathbb{N}$ . However, this inequality is not always true, for example, when  $k_1 = A_2; k_2 = A_1 + 1; k = A_2$ . Thus, our assumption cannot be true, and  $\text{mod}(d_1, T) \neq \text{mod}(d_2, T)$ . This means that  $v_1$  and  $v_2$  subscribe to different LNs.

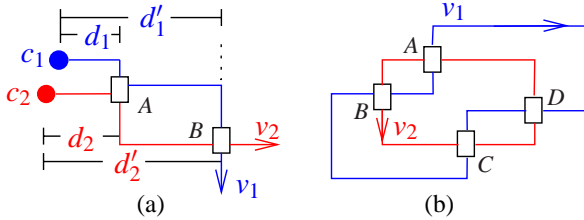


Fig. 8. Two or multiple shared buffers

By Theorem 2, VCs must stay in different LNs referring to *any shared buffer*. However, as overlapping VCs may have multiple shared buffers, LN partitioning might change with a different reference buffer. Figure 8a shows that two open-ended VCs,  $v_1$  and  $v_2$ , overlap in buffers  $A$  and  $B$ . Apparently, no conflict with respect to buffer  $A$  does not necessarily imply no conflict with respect to another buffer  $B$ . We derive the following theorem to check the *reference consistency*.

*Theorem 3:* Suppose that two overlapping VCs,  $v_1$  and  $v_2$ , have two shared buffers  $A$  and  $B$ . Let the distances from buffer  $A$  to  $B$  along  $v_1$  and  $v_2$  be  $d_{AB}^-(v_1)$  and  $d_{AB}^-(v_2)$ , respectively. Let the initial distance of  $c_1$  to  $A$  be  $d_1$ , to  $B$  be  $d_1'$ ; from  $c_2$  to  $A$  be  $d_2$ , to  $B$  be  $d_2'$ . Assume that  $c_1$  on  $v_1$  and  $c_2$  on  $v_2$  do not conflict in  $A$ , then  $d_{AB}^-(v_1) - d_{AB}^-(v_2) = kT$ , where  $T = \text{GCD}(D_1, D_2)$  and  $k \in \mathbb{Z}$ , is a sufficient and necessary condition for  $c_1$  and  $c_2$  to be conflict-free with respect to  $B$ . If so, we say the two shared buffers are *consistent*.

*Proof:* As  $d_{AB}^-(v_1) = d_1' - d_1$  and  $d_{AB}^-(v_2) = d_2' - d_2, \Rightarrow d_{AB}^-(v_1) - d_{AB}^-(v_2) = (d_1' - d_2') - (d_1 - d_2)$ . Further,  $d_{AB}^-(v_1) - d_{AB}^-(v_2) = kT \Leftrightarrow \text{mod}(d_1' - d_2', T) = \text{mod}(d_1 - d_2, T)$ . Condition  $\text{mod}(d_1, T) \neq \text{mod}(d_2, T) \Leftrightarrow \text{mod}(d_1', T) \neq \text{mod}(d_2', T)$ . Thus  $c_1$  and  $c_2$  are conflict free with respect to  $B$ .

By Theorem 3, we can further conclude that if two VCs have multiple shared buffers, all shared buffers must be consistent in order to be conflict-free. For instance, as shown in Figure 8b, if the two closed-loop VCs,  $v_1$  and  $v_2$ , have no conflict, then all shared buffers  $v_1 \cap v_2 = \{A, B, C, D\}$  must be consistent.

If the consistency is checked *pair-wise*, the total number of checking times is  $C_u^2 = u(u-1)/2$ , where  $u$  is the number of shared buffers. However, the check can be done efficiently, as we shall prove below.

*Theorem 4:* Suppose that  $v_1$  and  $v_2$  have at least three shared buffers  $A, B, C \in v_1 \cap v_2$ . If  $A$  and  $B$ , and  $B$  and  $C$  are consistent, then  $A$  and  $C$  are consistent.

*Proof:* As  $A$  and  $B$  are consistent,  $d_{AB}^-(v_1) - d_{AB}^-(v_2) = k_1T$ . As  $A$  and  $C$  are consistent,  $d_{AC}^-(v_1) - d_{AC}^-(v_2) = k_2T$ . By deducting the two equations, we have,  $d_{AB}^-(v_1) - d_{AB}^-(v_2) - (d_{AC}^-(v_1) - d_{AC}^-(v_2)) = (k_1 - k_2)T$ . Further, we have  $d_{BC}^-(v_1) - d_{BC}^-(v_2) = k_3T, k_3 \in \mathbb{Z}$ . According to Theorem 3,  $B$  and  $C$  are consistent. ■

By Theorem 4, reference consistency may be linearly checked. As a result, the total number of checking times is reduced to  $u - 1$ . If all shared buffers are consistent, any shared buffer can be used as a *reference buffer* to conduct LN partitioning and assignment. If they are not consistent,  $v_1$  and  $v_2$  conflict. ■

## VI. THE PROBLEM FORMULATION

We introduce additional definitions, and then define the multi-node VC configuration problem.

### A. Definitions

*Definition 7:* A network is a directed graph  $\mathcal{G} = M \times E$ , where each vertex  $m \in M$  represents a node, and each edge  $e \in E$  represents a link. On  $\mathcal{G}$ , if there is an edge directing from one node to another, the edge is unique.

*Definition 8:* A VC specification set  $\bar{V}$  comprises a set of VCs to be configured on the network  $\mathcal{G}$ . Each VC  $\bar{v}_i \in \bar{V}$  is associated with:

- $m_i \subseteq M$ : a subset of nodes in  $M$  to be visited by  $\bar{v}_i$ . The node set is not necessarily ordered and two consecutive nodes in  $m_i$  do not have to be adjacent in the network.
- $\bar{b}w_i$ : minimum bandwidth requirement (bits/second) of  $\bar{v}_i$ .

*Definition 9:* A VC implementation set  $V$  captures implementation options of  $\bar{V}$  on the network  $\mathcal{G}$ . Each VC implementation  $v_i \in V$ , which implements  $\bar{v}_i$ , is associated with:

- $P_i$ : a set of candidate shortest-distance paths which  $v_i$  may travel. A shortest path  $p_l \in P_i$  is expressed by a set of ordered and *adjacent* nodes on  $\mathcal{G}$ , since a pair of two adjacent nodes on  $\mathcal{G}$  implies exactly the directed link connecting them.  $\forall p_l \in P_i, m_i \subseteq p_l$ .
- $BW_i$ : a set of supported bandwidth (bits/second) of  $v_i$ .  $bw_l \in BW_i$  is the bandwidth of implementation  $v_i$  taking path option  $p_l$ .
- $R_{i,j}$ : a partial routing table created for a visiting node  $n_j$  by  $v_i$ .  $\forall r_z \in R_{i,j}, r_z$  is an entry  $(t, e_{in,x}, e_{out,y})$ , specifying that node  $n_j$  reserves slot  $t$  for a  $v_i$  packet from input link  $e_{in,x}$  to use output link  $e_{out,y}$ .  $R_j$  is the routing table of  $n_j$ , and  $R_j = \sum_i R_{i,j}$ .

*Definition 10:* On a network  $\mathcal{G}$ , a path function:  $\mathcal{P}: M \rightarrow E$  maps  $m_i \subseteq M$  to one path  $p_l \in P_i$ .

*Definition 11:* At node  $n_j$ , a routing function  $\mathcal{R}_j: (\mathcal{J}, E_{in,j}) \rightarrow E_{out,j}$  maps an  $e_{in,x} \in E_{in,j}$  to an  $e_{out,y} \in E_{out,j}$  for slot  $t \in \mathcal{J}$ .

## B. The Problem

Using the definitions above, we formulate the problem as follows:

Given a network  $\mathcal{G}$  and a VC specification set  $\bar{V}$ , find a VC implementation set  $V$  and determine from  $V$  (1) a path function  $\mathcal{P}()$  and (2) a routing function  $\mathcal{R}_j()$  for each node  $n_j$ , such that

$$\forall e_{in,x} \neq e_{in,y}, \mathcal{R}_j(t, e_{in,x}) \neq \mathcal{R}_j(t, e_{in,y}) \quad (1)$$

$$\bar{bw}_i \leq bw_l \quad (2)$$

$$\forall \text{ edge } e_k, Bw(e_k) \leq \kappa_{bw}(e_k) \quad (3)$$

$$\text{where } Bw(e_k) = \sum_i bw_i \text{ if } e_k \in \text{Edge}(p_l)$$

Condition (1) says that VC packets from two different input links of a switch can not be switched to the same output link simultaneously, i.e., VCs must be set up without conflict. Condition (2) expresses that each VC's bandwidth constraint must be satisfied. Condition (3) means that the total normalized (with the link capacity) bandwidth reserved by all VCs on a link cannot exceed the link bandwidth threshold  $\kappa_{bw}$ , which is defined in terms of the link capacity and  $0 \leq \kappa_{bw} \leq 1$ .  $\kappa_{bw}$  can be set for each link. If  $0 \leq \kappa_{bw}(e_k) < 1$ , this means that link  $e_k$  has room for routing BE traffic.

## VII. THE MULTI-NODE VC CONFIGURATION METHOD

### A. The Node Visiting Order of a VC

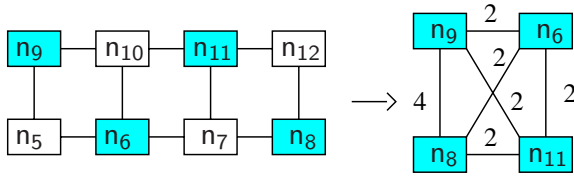


Fig. 9. The node set of a VC specification and its subgraph

A VC specification consists of a set of nodes, which may be un-ordered. To visit the nodes in shortest distance, we must first order them into a sequence, called a *tour*. We constrain that a node appears exactly once in the sequence, implying that we do not consider *forked* shortest paths. For each VC, its node set can be used to construct a subgraph, where an edge is weighted by the shortest distance between two nodes. Figure 9 exemplifies the problem. On the partial mesh, a VC specification has a node set  $\{n_{11}, n_6, n_8, n_9\}$ . The subgraph is drawn on the right side by labeling the edge with the shortest distance between a pair of nodes. If the VC implementation is open-ended, the visiting order following  $\{n_{11} \rightarrow n_6 \rightarrow n_8 \rightarrow n_9\}$  requires 8 hops. A re-ordering of the nodes into  $\{n_{11} \rightarrow n_8 \rightarrow n_6 \rightarrow n_9\}$  leads to 6 hops. This is a shortest path. Supposing that the VC implementation is a closed loop, if the loop visits nodes in order  $\{n_{11} \rightarrow n_6 \rightarrow n_8 \rightarrow n_9 \rightarrow n_{11}\}$ , the traveling distance is 10; If the loop takes the order  $\{n_{11} \rightarrow n_8 \rightarrow n_6 \rightarrow n_9 \rightarrow n_{11}\}$ , the traveling distance is 8 and this is a shortest path. Therefore, for each VC, we need to order the nodes in the node set so that the traveling distance

is shortest. It turns out that, for open-ended VCs, finding a tour visiting each node exactly once is the Hamiltonian Path Problem (HPP) [13]. We use the randomized algorithm to solve it. For closed-loop VCs, finding a tour visiting all nodes only once and back to its starting node is exactly the Traveling Sales Person (TSP) problem [13]. We use a branch-and-bound algorithm to find a shortest tour.

### B. The VC Configuration Algorithm

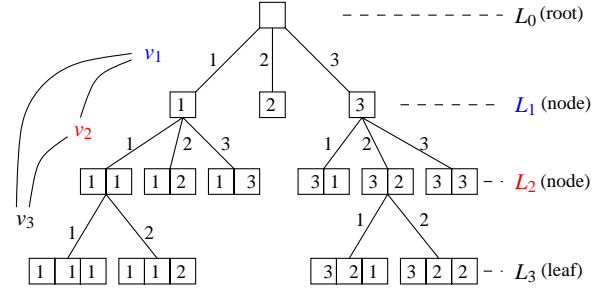


Fig. 10. Solution space

After ordering the node set for each VC specification into a shortest tour, we can then find the path options of the tour. We utilize a standard back-tracking algorithm to explore the path diversity. The algorithm is a recursive function performing a depth-first search, shown as Algorithm 1. The solution space is generated while the search is conducted. Figure 10 shows an example of solution space for three VCs,  $v_1$ ,  $v_2$  and  $v_3$ .  $v_1$  and  $v_2$  have three path options each, and  $v_3$  two options. The tree has three mapped levels plus one initial level ( $L_0$ ). The initial level is a dummy level representing that no VC is mapped yet. At each mapped level, the paths of the corresponding VC are enumerated and slot allocation is then conducted. Specifically, at level  $i$  ( $L_i$ ),  $v_i$  is mapped. Each level's node consists of a path of its own level VC plus a prefix that comprises the paths of its upper level VCs. For example, node 12 at  $L_2$  means that  $v_1$  selects path option 1 and  $v_2$  option 2. The last level nodes, called *leaves*, are solutions found. For instance, leaf 321 at  $L_3$  is a solution in which  $v_1$  takes path option 3,  $v_2$  option 2 and  $v_3$  option 1. From this example, we can see that the solution space is exponentially increased. Suppose that the size of a VC specification set is  $m$  and each VC has  $p$  alternative paths, such a solution tree has  $p^m$  leaf nodes and  $(p^{m+1} - 1)/(p - 1)$  total nodes. As a result, an algorithm that moves through all nodes in the tree must spend  $\Omega(p^m)$  time.

The backtracking algorithm for VC configuration trades runtime for memory consumption. At any time during the search, only the route from the start node to the current expansion node is saved. As a result, the memory requirement of the algorithm is  $O(m)$ . This is important since the solution space organization needs excessive memory if stored in its entirety. Since searching the entire solution space is time-consuming and not scalable, bounding functions are required to cut infeasible branches. In our case, the *pair-wise* slot allocation after path selection serves as a bounding function. While reaching each node in the tree, the paths of corresponding VCs are determined. This function then checks if VC-to-

**Algorithm 1** The pseudo code of VC configuration algorithm

---

**Input:** Q: a queue of all VCs' implementation options.  
**Output:** S: a queue of feasible solutions (s: a solution).  
Initially, cur\_level=0; Q.size=m; S and s are empty;  
Sort Q by a priority criterion;  
void **vc\_configuration**(cur\_level, Q, S, s){  
if (cur\_level==Q.size()){  
// at a leaf, create routing tables  
s.configure\_routing\_table();  
s.allocate\_slots\_to\_non\_overlapped\_vc();  
S.push(s);  
} else {  
// not a leaf, expand subtrees  
V=Q.pop(); // get options of current level's VC  
// try all branches of this level's VC  
for each v in V {  
if (slot\_allocation(v, s)) { //if false, prune the subtree  
s.push(v);  
vc\_configuration(cur\_level+1, Q, S, s);  
else break; } } return;  
}  
int **slot\_allocation**(v, s){  
for each v' in s  
if (VC\_to\_LN(v,v')==0)  
return 0;  
return 1; }

---

LN assignment can be done without conflict and bandwidth constraints can be satisfied. For example, at  $L_2$  nodes, slot allocation for  $(v_2, v_1)$  is performed; at  $L_3$  nodes (leaves), slot allocation for  $(v_3, v_1)$  and then  $(v_3, v_2)$  is conducted. Note that the slot allocation order must be observed. If the slot allocation fails, the algorithm prunes the current expansion node's subtrees, thus making the search efficient.

## C. VC-to-LN Assignment

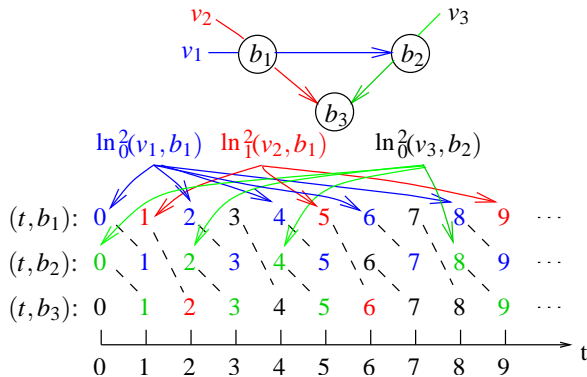


Fig. 11. An example of VC-to-LN assignment

The VC-to-LN assignment is the key step while constructing VCs. It is conducted pair-wise and incrementally. We detail this function in Algorithm 2 (adopted from [9]). The input to the algorithm is a pair of VCs,  $(v_i, v_j)$ <sup>3</sup>, and their paths

<sup>3</sup>VC pairs  $(v_i, v_j)$  and  $(v_j, v_i)$  are equivalent in the paper.

**Algorithm 2** The VC-to-LN assignment procedure

---

```

int VC_to_LN( $v_i, v_j$ ){
if ( $v_i \cap v_j == \emptyset$ ) return 1;
compute the shared number  $T$  of LNs,  $T = GCD(D_i, D_j)$ ;
if (reference_consistency( $v_i, v_j$ )==0) return 0;
//  $v_i$  and  $v_j$  overlap but satisfy reference consistency
take any shared buffer  $b$  as the reference buffer  $Ref(v_i, v_j) = b$ ;
if (state( $v_i$ )==0 && state( $v_j$ )==0) {
// Both states are 0
for  $v$  in  $\{v_i, v_j\}$  {
compute the available LN set for  $v$ ,  $AS_{ln}(v)$ ;
compute the required number of LNs  $N_{ln}(v) = \lceil NT/D \rceil$ ;
if  $|AS_{ln}(v)| < N_{ln}(v)$  return 0;
assign LNs from  $AS_{ln}(v)$  to  $v$ ;
allocate slot sets in the assigned LNs within  $D$  to  $v$ ;
state( $v_i$ )=1; state( $v_j$ )=1; }
return 1; }
if (state( $v_i$ ) != state( $v_j$ )) {
// One state is 0 and the other 1
// suppose (state( $v_i$ )=0 and state( $v_j$ )=1)
map  $v_j$ 's allocated slot sets to the new LN set as the
consumed LN set by  $v_j$ ,  $CS_{ln}(v_j)$ ;
compute the available LN set for  $v_i$ ,  $AS_{ln}(v_i)$ ;
compute the required number of LNs  $N_{ln}(v_i) = \lceil N_i T_i / D_i \rceil$ ;
if  $|AS_{ln}(v_i)| < N_{ln}(v_i)$  return 0;
assign LNs from  $AS_{ln}(v_i)$  to  $v_i$ ;
allocate slot sets in the assigned LNs within  $D_i$  to  $v_i$ ;
state( $v_i$ )=1;
return 1; }
if (state( $v_i$ )==1 && state( $v_j$ )==1) {
// Both states are 1
map  $v_i$ 's allocated slot sets to the new LN set as the
consumed LN set by  $v_i$ ,  $CS_{ln}(v_i)$ ;
map  $v_j$ 's allocated slot sets to the new LN set as the
consumed LN set by  $v_j$ ,  $CS_{ln}(v_j)$ ;
if ( $CS_{ln}(v_i) \cap CS_{ln}(v_j) == \emptyset$ )
return 1;
else return 0;
}
}

```

---

VC	Buf. set	bw	N	W(D)	LN	Slot set
$v_1$	$b_1, b_2$	1/2	1	2	$ln_0^2(v_1, b_1)$	$\{s_0^2(b_1), s_1^2(b_2)\}$
$v_2$	$b_1, b_3$	1/4	1	4	$ln_1^2(v_2, b_1)$	$\{s_1^4(b_1), s_3^4(b_3)\}$
$v_3$	$b_2, b_3$	3/8	3	8	$ln_0^2(v_3, b_2)$	$\{s_{0,2,4}^8(b_2), s_{1,3,5}^8(b_3)\}$

TABLE II

VC PARAMETERS AND VC-TO-LN ASSIGNMENT RESULTS FOR FIG. 11

are known. The function returns 1 if VC-to-LN assignment is done successfully for both VCs, and returns 0 otherwise. Besides, the configuration updates for  $v_i$  and  $v_j$  are stored and may be used in further tests. A VC  $v$  has two configuration states, either 0 or 1. 0 means that VC-to-LN is not performed for  $v$  yet; 1 means that the VC-to-LN is done successfully.

We detail an example on how this VC-to-LN assignment is conducted. Figure 11 shows three VCs,  $v_1$ ,  $v_2$  and  $v_3$ . In the figure, a bubble represents a buffer. Their parameters are

listed in Table II. Their paths are represented by the respective ordered buffer sets. The selection of admission windows ( $W_s$  or  $D_s$ ) is oriented toward finding enough LNs for overlapping VCs and subject to application constraints. For example, if two VCs overlap, at least two LNs must be created, one for each VC. This means the admission windows of the two VCs must be selected such that their GCD is not less than 2. Of course, any selection of admission windows must not violate the application constraints which may limit the minimal and/or maximum size of the admission windows.

The VC-to-LN assignments are performed in order  $(v_1, v_2)$ ,  $(v_1, v_3)$  and  $(v_2, v_3)$ . Whenever the assignment proceeds, the function first check if they overlap. If not, no further assignment is conducted at this stage and it returns 1. If yes, the number of LNs is calculated and the reference consistency is checked. If consistent, any one of the shared buffers may be picked up as the reference buffer to partition LNs. The procedures are detailed as follows:

- 1) VC\_to\_LN( $v_1, v_2$ ): Initially, both VCs,  $v_1$  and  $v_2$ , are not configured yet. For any VC, we compute the available LN set, and the required number of LNs according to its bandwidth requirement. If the available bandwidth is greater than the required one, we allocate enough available LNs to the VC. Otherwise, the allocation fails. Once an allocation for a VC is done, it does not change anymore.

For  $v_1$  and  $v_2$ ,  $Ref(v_1, v_2) = b_1$ . Since  $D_1 = 2$  and  $D_2 = 4$ ,  $T = GCD(D_1, D_2) = 2$ . We can partition  $b_1$ 's slots into two logical sets. Initially,  $state(v_1)=0$  and  $state(v_2)=0$ . The branch of "Both states are 0" is executed. We take  $v_1$  first. The available LN set for  $v_1$   $AS_{in}(v_1) = \{0, 1\}$ , thus  $|AS_{in}(v_1)| = 2$ . The required number of LNs  $N_{in}(v_1) = \lceil N_1 T / W_1 \rceil = 1$ . As  $|AS_{in}(v_1)| > N_{in}(v_1)$ , there are enough LNs to support  $v_1$  bandwidth. We assign  $ln_0^2(v_1, b_1)$  to  $v_1$ . The consumed LN set of  $v_1$   $CS_{in}(v_1) = \{0\}$ . We then allocate slot sets  $s_0^2(b_1)$  and  $s_1^2(b_2)$  to  $v_1$ . The two sets constitute LN  $ln_0^2(v_1, b_1)$ . Next, we take  $v_2$  up.  $AS_{in}(v_2) = \{0, 1\} - CS_{in}(v_1) = \{1\}$ . The required number of LNs of  $v_2$   $N_{in}(v_2) = \lceil N_2 T / W_2 \rceil = 1$ . We assign  $ln_1^2(v_2, b_1)$  to  $v_2$ . Then we allocate slot sets  $s_1^4(b_1)$  and  $s_2^4(b_3)$  to  $v_2$ . After this assignment,  $state(v_1)=1$  and  $state(v_2)=1$ .

- 2) VC\_to\_LN( $v_1, v_3$ ): At this step,  $v_1$  is allocated while  $v_3$  is not. We mainly deal with the un-mapped one  $v_3$ . To calculate the available LN set for  $v_3$ , we must compute the consumed LN set, which is allocated to  $v_1$ . This is done by virtually mapping  $v_1$ 's allocated LN(s) to the current LN set for  $(v_1, v_3)$  with respect to the new reference buffer  $b_2$ .

For  $v_1$  and  $v_3$ ,  $Ref(v_1, v_3) = b_2$ . As  $D_1 = 2$  and  $D_3 = 8$ ,  $T = GCD(D_1, D_3) = 2$ . Since  $state(v_1)=1$  and  $state(v_3)=0$ , the branch of "One state is 0 and the other 1" is executed. We map  $ln_0^2(v_1, b_1)$  with respect to the reference buffer  $b_2$ , resulting in an equivalent LN  $ln_1^2(v_1, b_2)$ . Thus the consumed LN set of  $v_1$   $CS_{in}(v_1) = \{1\}$ . The available LN set of  $v_3$  is  $AS_{in}(v_3) = \{0, 1\} - CS_{in}(v_1) = \{0\}$ . The required number of LNs of  $v_3$

$N_{in}(v_3) = \lceil N_3 T / W_3 \rceil = 1$ . We assign  $ln_0^2(v_3, b_2)$  to  $v_3$ . Then we allocate slot sets  $s_{0,2,4}^8(b_2)$  and  $s_{1,3,5}^8(b_3)$  to  $v_3$ . After this assignment,  $state(v_3)=1$ .

- 3) VC\_to\_LN( $v_2, v_3$ ): At this step, both VCs,  $v_2$  and  $v_3$  are allocated. Since their allocations will not change, we determine whether the intersection of their allocated LN(s) is empty. If it is, both allocations are still valid; otherwise, they conflict. This is done by mapping the allocated LN(s) to the current LN set for  $(v_2, v_3)$  with respect to the new reference buffer  $b_3$ .

For  $v_2$  and  $v_3$ ,  $Ref(v_2, v_3) = b_3$ . As  $D_2 = 4$  and  $D_3 = 8$ ,  $T = GCD(D_2, D_3) = 4$ . Since  $state(v_2)=1$  and  $state(v_3)=1$ , the branch of "Both states are 1" is executed. In this step, we check whether the allocated slot sets for  $v_2$  and  $v_3$  can stay in different LNs after mapping them to the four LNs with respect to the reference buffer  $b_3$ . We map  $s_1^4(b_1)$  of  $v_2$  on  $b_3$ , obtaining an equivalent LN  $ln_2^4(v_2, b_3)$ . Then we map  $s_{0,2,4}^8(b_2)$  of  $v_3$  on  $b_3$ , obtaining LN  $ln_{1,3}^4(v_3, b_3)$ . Because  $ln_2^4(v_2, b_3) \cap ln_{1,3}^4(v_3, b_3) = \emptyset$ ,  $v_2$  and  $v_3$  are conflict free with their slot assignment.

After the above three steps, the VC-to-LN assignments for the three VCs are successful. The slot sets are allocated accordingly, as shown in Table II. These can be used to create routing tables in switches.

#### D. Routing Table Creation

While reaching a leaf ( $cur\_level=Q.size()$ ), a feasible solution is found. With each VC, a switch's partial routing table is created according to the VC's path and the allocated LNs, more accurately, the allocated slot sets within the admission cycle. The slot sets determine when the VC passes a particular buffer in a switch. For instance, if a VC  $v$  with an admission cycle  $D$  subscribes to  $s_{\tau_1}^D(b)$  and  $s_{\tau_2}^D(b)$ , then slots  $\tau_1 + kD$  and  $\tau_2 + kD$  ( $\forall k \in \mathbb{N}$ ) of  $b$  are reserved for  $v$ . The VC path determines the input link  $e_{in}$  and the output link  $e_{out}$  of the switch used by  $v$  packets at the reserved slots. Thus, two routing table entries,  $(\tau_1 + kD, e_{in}, e_{out})$  and  $(\tau_2 + kD, e_{in}, e_{out})$ , can be created in the switch. By composing the partial routing tables of all visiting VCs in a switch, we obtain a complete routing table for the switch. Optimization is also used to shrink the size of the routing tables. For example, entries  $(4k, e_{in}, e_{out})$  and  $(4k+2, e_{in}, e_{out})$  can be reduced to one entry  $(2k, e_{in}, e_{out})$ .

## VIII. EXPERIMENTAL RESULTS

### A. Synthetic Communication Patterns

To investigate the tradeoff between capability and runtime with our approach, we conduct experiments on mesh networks using *open-ended VCs*. Although the experiments are performed on meshes, our method is topology independent. Since the path diversity is the factor that complicates the exploration of the solution space, we realize and compare three different ways of considering alternative minimal paths:

- One Path Option per VC (OPO): A VC considers one and only one locally optimal path. This path is selected from a set of candidates by minimizing the number of overlapped links with the previous VC in the VC set. This is a greedy scheme.

- Half Path Options per VC (HPO): A VC randomly chooses half of its path alternatives.
- Full Path Options per VC (FPO): A VC considers all alternative paths.

The back-tracking algorithm can be used to explore all alternative paths of VCs, allowing us to find all possible solutions. However, finding all solutions consumes long execution time and may not be necessary in some cases, because all solutions are equally good in the sense that VCs are conflict free, VC bandwidth demand and link bandwidth constraints are satisfied, and VC paths are shortest. Therefore we are interested in finding the first solution in this experiment.

$n_{vc}$	OPO			HPO			FPO		
	min.	avg.	max.	min.	avg.	max.	min.	avg.	max.
12	0.31	0.38	0.46	0.36	0.61	1.33	0.44	4.33	22
16	0.46	0.53	0.6	0.49	1.5	6.35	0.57	25.2	138
20	0.8	0.83	1.2	0.99	2.25	30	2.08	48.5	480

TABLE III

RUNTIME (IN SECONDS) WITH OPO, HPO AND FPO

Because there is no standard NoC benchmark available, we build a synthetic traffic generator to create random VC sets. The traffic generator takes as arguments the number of VCs, the bandwidth requirement of VCs and the maximum number of nodes in a VC. Note that a randomly generated VC set may not have a solution, for example, in case that VCs are demanding bandwidth more than the capacity of a particular link. To create only valid problems, we first run our program using FPO as a filter to get the valid problem set. In a  $4 \times 4$  network, we set the VC bandwidth up to 1/2 link capacity, and the maximum number of nodes to 7. We vary the number  $n_{vc}$  of VCs from 11 up to 20. We obtain 200 problems after executing FPO, 20 for each  $n_{vc}$  in the range [11, 20]. Then we run HPO and OPO on the problem set. The percentage of the problems solved by HPO is 80%. The OPO solves only 22% of the problems. We compare their runtime in seconds for  $n_{vc} = 12, 16, 20$  in Table III.

As can be observed, OPO runs fastest, HPO the second, FPO the slowest. Besides, both HPO and FPO show great runtime variation from minimum to maximum. This is due to that a different problem may vary very differently on which leaf a solution exists. Since the solution is found while systematically constructing the solution tree, the runtime variation is high. The OPO does not exhibit much runtime variation because it explores only a linear path in the solution tree. The variation can be attributed to how fast the slot allocation is completed. In summary, back-tracking with FPO is the most powerful but takes longest execution time. The OPO method does no back-tracking. It is the least effective but fastest. Back-tracking with HPO sits in between. We also observed in our experiments, if limiting the program execution time, HPO/OPO can find a solution for some problems for which FPO/HPO cannot find a solution before the time threshold is reached.

From the above analysis, we can see that there is no single “golden” solution when it comes to explore the path diversity in the network. The tradeoffs between the alternatives may help designers to make right choices. In the future, we

anticipate a variety of use cases, *static*, *dynamic*, and *semi-dynamic*. For the static case, VC configurations are computed offline. In this case, optimal solutions weight over faster run time; For the dynamic case, VC configurations may be computed during system execution. In such a case, fast runtime over-weight optimal solutions. Some use cases between the static and the dynamic are also possible. For example, we can image a semi-static case where sets of VCs are pre-computed and the network switches the VC sets during runtime (mode switching), or partial VCs are static and partial VCs dynamic.

## B. An Industrial Case Study

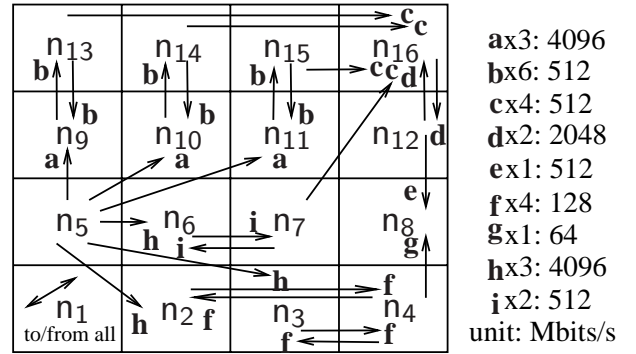


Fig. 12. Node-to-node traffic flows for a radio system

We applied our program to a real application provided by Ericsson Radio Systems. As mapped onto a  $4 \times 4$  mesh in Figure 12, this application consists of 16 IPs. Specifically,  $n_2, n_3, n_6, n_9, n_{10}$  and  $n_{11}$  are ASICs;  $n_4, n_7, n_{12}, n_{13}, n_{14}$  and  $n_{15}$  are DSPs;  $n_5, n_8$  and  $n_{16}$  are FPGAs;  $n_1$  is a device processor which loads all nodes with program and parameters at start-up, sets up and controls resources in normal operation. Traffic to/from  $n_1$  is for system initial configuration and no longer used afterwards. There are 26 node-to-node traffic flows that are categorized into nine types of traffic flows  $\{a, b, c, d, e, f, g, h, i\}$ , as marked in the figure. The traffic flows are associated with a bandwidth requirement. We use *closed-loop VCs* in this case study.

The case study comprises two phases: *VC specification* and *VC configuration*. The VC specification phase consists of *determining link capacity*, *normalizing VC bandwidth demand* and *merging traffic flows*. The VC configuration phase runs the configuration program, exploring the path diversity using FPO. In the following, we detail the case-study steps.

We first determine the minimum required link capacity by considering a heaviest loaded link. For each link  $j$ , suppose that  $U_j$  is a set of traffic flows passing it, we have

$$\sum_{i=1}^{|U_j|} bw_i \leq \kappa_{bw} \cdot bw_{link}$$

where  $bw_i$  is the data rate (bandwidth) of a traffic flow  $i$  and  $bw_{link} = l_w \cdot f_{clk}$  where  $l_w$  is the payload bit width of a flit and  $f_{clk}$  is the network clocking frequency. In the example, **a** and **h** are multi-cast traffic, and others are unicast traffic. The most heavily loaded link may be  $e(n_5, n_9)$ . The **a**-type

traffic passes it and  $bw_a = 4096$  Mbits/s. As all traffic flows are implemented using VCs,  $\kappa_{bw} = 1$  in this case. To support  $bw_a$ ,  $bw_{link}$  must not be less than 4096 Mbits/s. We choose the minimum 4096 Mbits/s for  $bw_{link}$ . This is an initial estimation and subject to adjustment and optimization later on.

After obtaining the link capacity, we normalize the bandwidth demand into a fraction of link capacity. For example, if  $bw_{link} = 4096$  Mbits/s, 512 Mbits/s is equivalent to  $\frac{1}{8} bw_{link}$ .

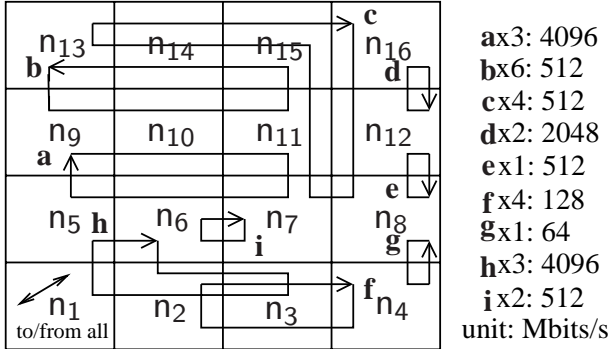


Fig. 13. Merged traffic flows, showing one possible path for each VC

Then we *merge traffic flows* by taking advantage of multi-node VCs. This can be done for multicast, low-bandwidth and round-trip traffic. In the example, for the two multicast traffic **a** and **h**, if a VC specification contains only one source node and one destination node, we have to define 6 VCs for **a** and **h**, specifically,  $\bar{v}_1(n_5, n_9)$ ,  $\bar{v}_2(n_5, n_{10})$ , and  $\bar{v}_3(n_5, n_{11})$  for three **a** flows, and  $\bar{v}_4(n_5, n_6)$ ,  $\bar{v}_5(n_5, n_3)$ , and  $\bar{v}_6(n_5, n_2)$  for three **h** flows. Each of them must provide a normalized bandwidth 1. However, there are only 3 outgoing links for node  $n_5$  on the mesh. Hence it is impossible to build such 6 VCs with the chosen link capacity. As our VC specification considers multiple nodes, we need to build only two multi-node VCs for traffic **a** and **h** as  $\bar{v}_a(n_5, n_9, n_{10}, n_{11})$  and  $\bar{v}_h(n_5, n_6, n_2, n_3)$ . In the example, traffic **b**, **c** and **f** require low bandwidth. We specify a VC to include as many nodes as a type of traffic flow spreads. For traffic **b**, we define a six-node VC,  $\bar{v}_b(n_9, n_{10}, n_{11}, n_{13}, n_{14}, n_{15})$ ; for **c**, a five-node VC  $\bar{v}_c(n_{13}, n_{14}, n_{15}, n_{16}, n_7)$ ; for **f**, a three-node VC  $\bar{v}_f(n_2, n_3, n_4)$ . Furthermore, as we use a closed-loop VC, two simplex traffic flows can be merged into one duplex flow. For instance, for two **i** flows, we specify only one VC  $\bar{v}_i(n_6, n_7)$ . The number  $n_c$  of containers on a VC implementation  $v$  is derived from  $n_c \geq \bar{b}w \cdot |v|$ , where  $\bar{b}w$  is the normalized bandwidth demand of  $\bar{v}$  and  $|v|$  is the length of the loop path of the VC implementation  $v$ . If  $n$  node-to-node flows are specified and implemented with closed-loop VCs,  $n$  VCs must be set up and at least  $n$  containers are required, one for each VC. Performing this traffic-merge step results in 9 multi-node VCs as shown in Figure 13, requiring at least 9 containers. If the 26 node-to-node flows were *specified* with VCs containing only two nodes (one source node and one destination node), 26 VCs would have been defined, demanding at least 26 containers. This tells us that, with the multi-node VC specification, the network can be much more efficiently utilized.

With the three steps above, we complete defining the VC

specification set. While executing the program to configure the VCs, we investigate the impact of VC sorting. Since VC sorting determines the VC levels in the solution tree and the VC-to-LN assignment order, it affects the runtime and the number of solutions. We tried three sorting schemes: *random*, *higher bandwidth first*, *less number of path options first*. In order to compare the potential of the schemes, our algorithm terminates after all solutions are found using FPO. We did not do any tweaking or tuning but used the original IP-to-node mapping and IP communication patterns without change. Corresponding to the three sorting schemes, the number of solutions found is 33, 30 and 76; the run time is 6, 6 and 12 seconds. Sorting by the number of path options is best in this example. This means that VCs with fewer optional paths should be laid out first because they are more constrained. As a result, pruning subtrees and allocating slots are more effective if they are considered in the upper levels in the tree.

## IX. CONCLUSION

Configuring VCs is a general problem for NoC application design with TDM-type guaranteed services. Its complexity arises from the network path diversity and various path overlapping scenarios. In the paper, based on our generalized concept of LN, we develop theorems for the configuration of conflict-free VCs. They are applicable to both open-ended and closed-loop VCs in current NoC proposals. Furthermore, we give a formulation on the multi-node VC configuration problem, and propose a back-tracking algorithm to constructively search for feasible solutions. Our experimental results with synthetic traffic and an industrial case study justify our approach in effectiveness and efficiency.

While our VC configuration algorithm can cut branches which do not possibly lead to a solution, it allows us to find all possible solutions. These solutions can be further evaluated using an objective function, for example, for load balancing, to find optimal ones. This also helps to make the search more efficient by cutting more branches. This potential has not yet been explored, and it remains to be done in the future.

In the paper, we have considered *synchronous* TDM VCs on which VC packets use consecutive slots in consecutive switches, i.e., they are not stalled. The network is synchronous with switches operating at the same frequency but potentially with different phases. This type of TDM VC guarantees both bandwidth and latency, and the switches require only a single buffer (no queue) per link. However, it couples the latency requirement with the bandwidth requirement. For low-bandwidth low-latency traffic, it leads to overbooking bandwidth in order to satisfy the low latency constraint. In the future, we will extend our framework to cover *asynchronous* TDM VCs in order to make more efficient use of link bandwidth and to allow asynchronous network communication. With the asynchronous TDM VCs, VC packets are stallable. Switches communicate asynchronously with each other. They are equipped with queues to store VC packets and a scheduling strategy guarantees the allocated link bandwidth. In contrast to the synchronous TDM VCs, the VC bandwidth can be guaranteed but latency may vary.

## GLOSSARY

**Admission pattern and Admission cycle:** A certain number  $N$  of VC packets are admitted into the network over a sequence of  $D$  time slots. This is called an admission pattern. An infinite VC packet flow enters the network by repeating this pattern.  $D$  is the admission cycle or admission window.

**Admission class** [11]: the admission class of a packet admitted at time  $t_0$  with initial destination distance  $i(t_0)$  is defined as  $\text{mod}(t_0 + i(t_0), D)$ , where  $D \in \mathbb{N}$ .

**Container** [2]: A container is a *special packet* used to carry data packets, like a vehicle carrying passengers.

**Logical Network (LN):** an infinite set of associated (time slot, buffer) pairs with respect to a buffer on a given VC.

**Quality of Service (QoS):** the capability of performance guarantees provided by a network.

**Target class:** With respect to a buffer  $b$  and a natural  $d < D$ , we define a *target class* as an infinite set of packets that arrive at buffer  $b$  at slots  $d + kD$ ,  $\forall k \in \mathbb{N}$ . We call  $d$  the *initial distance* of the target class to buffer  $b$ .

**Temporally Disjoint Network (TDN)** [2]: A TDN consists of a set of regularly partitioned buffers in a network. Packets launched on one TDN never meet those packets launched on another TDN because they always have a different distance to any buffer in the network.

**Time Division Multiplexing (TDM) VC:** Two or more VC packet streams take turns to share buffers and link-bandwidth. The time domain is divided into slots and VCs own dedicated slots to use the shared resources.

**Virtual Circuit (VC):** a connection-oriented communication service between communicating entities in a packet-switched network.

**VC packets:** Packets delivered on a VC.

## ACKNOWLEDGMENT

The authors appreciate helpful discussions on the algorithms with Dr. Christian Schulte and Mikael Lagerkvist, and would like to thank Dr. Ingo Sander, Mikael Millberg and Dr. Tarvo Raudvere for their comments in improving the paper.

We thank the anonymous reviewers for their insightful comments.

The research is partially supported by the EU FP6 project SPRINT under contract 027580.

## REFERENCES

- [1] K. Goossens, J. Dielissen, and A. Rădulescu, "The  $\text{\AA}$ ethereal network on chip: Concepts, architectures, and implementations," *IEEE Design and Test of Computers*, vol. 22, no. 5, pp. 21–31, Sept-Oct 2005.
- [2] M. Millberg, E. Nilsson, R. Thid, and A. Jantsch, "Guaranteed bandwidth using looped containers in temporally disjoint networks within the Nostrum network on chip," in *Proceedings of the Design Automation and Test in Europe Conference*, February 2004.
- [3] T. Bjerregaard and J. Sparso, "A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip," in *Proceedings of the Design, Automation and Test in Europe Conference*, 2005, pp. 1226–1231.
- [4] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny, "QNoC: QoS architecture and design process for network on chip," *The Journal of Systems Architecture*, December 2003.
- [5] A. Hansson, K. Goossens, and A. Rădulescu, "A unified approach to constrained mapping and routing on network-on-chip architectures," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, Sept. 2005.

- [6] S. Stuijk, T. Basten, M. Geilen, A. H. Ghamarian, and B. Theelen, "Resource-efficient routing and scheduling of time-constrained network-on-chip communication," in *Proceedings of the 9th Euromicro Conference on Digital System Design*, Aug. 2006.
- [7] D. Wingard and A. Kurosawa, "Integration architecture for system-on-a-chip design," in *Proceeding of the IEEE 1998 Custom Integrated Circuits Conference*, May 1998.
- [8] P. Pop, P. Eles, and Z. Peng, "Bus access optimization for distributed embedded systems based on schedulability analysis," in *Proceedings of the Design, Automation and Test in Europe Conference*, March 2000.
- [9] Z. Lu and A. Jantsch, "Slot allocation using logical networks for TDM virtual-circuit configuration for network-on-chip," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD'07)*, Nov. 2007.
- [10] E. Nilsson and J. Öberg, "Reducing peak power and latency in 2D mesh NoCs using globally pseudochronous locally synchronous clocking," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, September 2004.
- [11] J. T. Brassil and R. L. Cruz, "Bounds on maximum delay in networks with deflection routing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 7, pp. 724–732, July 1995.
- [12] A. Borodin, Y. Rabani, and B. Schieber, "Deterministic many-to-many hot potato routing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 6, pp. 587–596, 1997.
- [13] J. Bang-Jensen and G. Gutin, *Digraphs: Theory, Algorithms and Applications*. Springer-Verlag, 2000.



**Zhonghai Lu** received BSc. from Beijing Normal University, China in 1989. Since then he had worked extensively in industry for several electronic, communication and embedded systems companies as a system engineer and project manager for eleven years. Afterwards, he entered the Royal Institute of Technology (KTH), Sweden in 2000. From KTH, he received MSc. and PhD. in 2002 and 2007, respectively. He is currently a researcher at KTH, supervising a number of MSc. students and co-supervising several PhD. students.

Dr. Lu has published over 25 peer-reviewed technical papers in journals, book chapters and international conferences in the areas of networks/systems on chips, embedded real-time systems and communication networks. He is a reviewer for a number of journals and conferences. His research interests include computer systems and VLSI architectures, interconnection networks, system-level design and HW/SW codesign, reconfigurable and parallel computing, system modelling, refinement and synthesis, and design automation.



**Axel Jantsch** received a Dipl.Ing. (1988) and a Dr. Tech. (1992) degree from the Technical University Vienna. Between 1993 and 1995 he received the Alfred Schrdinger scholarship from the Austrian Science Foundation as a guest researcher at the Royal Institute of Technology (KTH). From 1995 through 1997 he was with Siemens Austria in Vienna as a system validation engineer. Since 1997 he is with the Royal Institute of Technology, Stockholm, Sweden. Since December 2002 he is full professor in Electronic System Design.

A. Jantsch has published over 140 papers in international conferences and journals and one book in the areas of VLSI design and synthesis, system level specification, modelling and validation, HW/ SW codesign and cosynthesis, reconfigurable computing and networks on chip. He has served on a number of technical program committees of international conferences such as FDL, DATE, CODES+ISSS, SOC, and HDLCON and others. He has been TPC chair of SSDL/FDL 2000, TPC cochair of CODES+ISSS 2004 and general cochair of CODES+ISSS 2005. Since December 2002 he is Subject Area Editor for the Journal of System Architecture. At the Royal Institute of Technology A. Jantsch is heading a number of research projects involving a total number of 10 Ph.D. students, in the areas of system level specification, design, synthesis, validation and networks on chip.