LOFT: A High Performance Network-on-Chip Providing Quality-of-Service Support

Jin Ouyang* and Yuan Xie
Computer Science and Engineering Dept., the Pennsylvania State University
*jouyang@cse.psu.edu

Abstract

Providing quality-of-service (QoS) for concurrent tasks in many-core architectures is becoming important, especially for real-time applications. QoS support for on-chip shared resources (such as shared cache, bus, and memory controllers) in chip-multiprocessors has been investigated in recent years. Unlike other shared resources, network-on-chip (NoC) does not typically have central arbitration of accesses to the shared resource. Instead, each router shares the responsibility of resource allocation. While such distributed nature benefits the scalable performance of NoC, it also dramatically complicates the problem of providing QoS support for individual flows. Existing approaches to address this problem suffer from various shortcomings such as low network utilization and weak QoS guarantees. In this work, we propose LOFT NoC architecture which features both high network utilization and strong QoS guarantees. LOFT is based on the combination of two mechanisms: a) locally-synchronized frames (LSF), which is a distributed frame-based scheduling mechanism that provides flexible QoS guarantees to different flows and b) flit-reservation (FRS), which is a flow-control mechanism integrated in LSF that improves network utilization. The experimental results show that LOFT delivers flexible and reliable QoS guarantees while sufficiently utilizes available network capacity to gain high overall throughput.

1. Introduction

The diminishing return from instruction level parallelism (ILP) has led the semiconductor industry toward utilizing task level parallelism (TLP) with chip-multiprocessors (CMP) and multiprocessor system-on-chip (MPSoC). For both commercial products [1]–[3] and research prototypes [4], [5], the number of processors on a single chip is increasing. For future large scale CMPs and MPSoCs, delivering sufficient bandwidth for on-chip communication is both critical to performance and challenging. Among various on-chip interconnect candidates, network-on-chip (NoC) has excellent scalability in that the throughput it can provide is proportional to its complexity and energy consumption. Therefore it is envisioned to be a solution for interconnecting tens to hundreds cores for future many-core architectures.

Besides good overall throughput, it is becoming increasingly important to provide QoS support in many-core architectures for individual tasks, especially for real-time tasks. QoS support helps real-time tasks to maintain the expected performance and functionalities. It also enables composability and simplifies system integration during design time. Finally, QoS support enables performance isolation, which benefits the effectiveness of system-level task scheduler. Consequently, we have seen significant recent research efforts on QoS support in on-chip shared resources such as caches [6]–[8] and memory controllers [9]–[11].

However, providing QoS guarantees with NoC turns out to be difficult, because tens and hundreds of tasks compete for bandwidth while the scheduling is distributed to each router. The lack of global knowledge hinders implementing robust QoS guarantees with straightforward schemes. To address this difficulty, existing approaches to provide QoS support rely on resource reservations or relaxing guarantees, resulting in either low network throughput or weak guarantees.

The focus of this work is to design a new NoC architecture that can provide strong and flexible QoS support, and at the same time can sufficiently utilize network bandwidth to achieve high overall performance. Our proposed LOFT network-on-chip architecture achieves the above goal based on the combination of two mechanisms:

- **Locally-Synchronized Frame (LSF)** is a frame-based scheduling mechanism implemented locally in each router, in order to provide strong and flexible service guarantees to respective flows. In LSF, the distributed scheduling within each router only requires local information exchange between adjacent routers, without the need of global knowledge. Therefore, LSF is well scalable, especially suitable for future large scale multiprocessor architectures.
- **Flit-Reservation (FRS)** is a flow-control mechanism bound to LSF and improves overall network utilization and throughput. In FRS, a look-ahead flit is sent before the data flits, to reserve bandwidth and buffers along the path to the destination. The data flits follow the look-ahead flit and reclaim resources booked by the look-ahead flit to make progress. Look-ahead flits are routed on a separate look-ahead network without interfering with data flits. Since look-ahead flits are much shorter than data flits, the look-ahead network is lightly loaded and quite fast.

The contribution of LOFT can be generalized as follows:

- **1)** LOFT can provide robust throughput guarantee to each rate-observing flow even with interference of malicious aggressors
that try to exhaust network bandwidth; b) LOFT also guarantees delay bound at much finer granularity than similar approaches [12, 13]; c) LOFT can fully exploit under-utilized bandwidth, and achieve excellent overall performance; d) the extra hardware introduced by LOFT is light-weight, incurring very low overheads.

2. Background

In this section, we first describe the required features in a network-on-chip architecture that can provide quality-of-service. We then review related work on quality-of-service (QoS) and guaranteed service (GS) support for network-on-chip (Guaranteed service support can be viewed as a special problem of QoS). However, some NoC literatures use the terms interchangeably, both referring to mechanisms that provide bandwidth or latency guarantee).

2.1. Quality-of-Service in Network-on-Chip

We assume a general model as follows. In a multi-hop NoC connected by routers, multiple processing elements (PE) send data to each other. A flow refers to the unique traffic sent from one PE (source) to another PE (destination). Flows are uni-directional. We use flow_{ij} to refer to the traffic from node i to node j and vice versa. A single node can be sources as well as destinations of multiple flows. Some links in NoC are used by several flows, which effectively share the link bandwidth. For a certain flow requiring quality-of-service, say flow_{ij}, it exposes its requirements as minimum throughput r_{ij} and maximum delay bound b_{ij}. With this model, the desired features of QoS support in NoC is specified as follows:

(a) Guaranteed minimum throughput: For flow_{ij} requiring quality-of-service, its data rate is at least r_{ij} regardless of other flows contending for bandwidth.

(b) Guaranteed maximum delay: During design phase, it should be possible to calculate the maximum packet delay b_{ij} according to the path taken by flow_{ij}. This can be used by various design-time procedures such as task binding and route computation. During run-time, the actual maximum packet delay should be always less than the previously calculated bound.

(c) Fairness in throughput allocation: It should allow contending flows to specify either equal or differentiated data rates (fair allocation or differentiated allocation). During run-time the actual data rates obtained are proportional to the specified data rates. As a priori, requirements (a) and (b) should be satisfied.

(d) Low level of under-utilization: It is not likely that a flow will be constantly using its booked throughput. Also, the full capacity of the network might not be totally booked. In either case, other flows should be able to scavenge the excess bandwidth to help performance.

(e) Low hardware complexity: The QoS mechanism is desirable to introduce little hardware overheads.

2.2. Related Work

A number of previous studies have developed mechanisms to provide quality-of-service and guaranteed service in NoC. Time-division-multiplexing (TDM) circuit-switching is proposed to provide bandwidth and latency guarantees in Æthereal NoC [14]. Each guaranteed flow is mapped to a virtual circuit, by reserving a time slot for each link along the path. By time-slot reservation, Æthereal architecture can provide the exact bandwidth booked by the flow and analyzable delay bounds. However, it does not allow guaranteed flows to use excess bandwidth when the network is under-utilized. Also, how to arbitrate amongst throughput requests is not clear from the original work. Nostrum NoC [15] also exploits the idea of TDM to create virtual circuits and uses the so-called “containers” to provide bandwidth guarantee. It only works for defective routing and shares the same problem with Æthereal.

In MANGO NoC [16], a flow needs to reserve the virtual channels along its path to construct a virtual circuit before sending packets on the virtual circuit. Therefore MANGO implements a large amount of buffers and complex switch modules. In addition, this connection-based approach tends to penalize short-lived transactions and compromises the overall throughput. In SonicsMX [17], sources insert interval markers to acquire bandwidth. The benefit of this approach is the absence of circuit setup. However, it does not provide hard guarantees of either minimum bandwidth or maximum delay.

Our work is closely related to Globally-Synchronized Frames (GSF) [12] and its extension [13]. GSF adopts the frame-based scheduling principles proposed in Rotate Combined Queues (RCQ) [18] and offers excellent throughput guarantees. In GSF, time is coarsely quantized into frames, and each flow can reserve a fraction in the frame to inject data flits. Frames are prioritized according to their ages, and the data flits belonging to older frames are switched first in any router. GSF also allows bursty flows to utilize excess bandwidth by providing multiple on-the-fly frames and fast frame recycling. Without circuit setup and dedicated virtual channels, GSF has relatively low hardware complexity and good scalability.

However, GSF suffers from several problems. First, as suggested by Grot et al [13] and Das et al [19], to maintain a good performance GSF needs large frame sizes and source queues. For example, to achieve good throughput in a 64-node NoC, GSF needs a 2000-flit source queue [13], [19] which translates to a significant buffer overhead. In addition, the use of a large frame size causes the delay bounds to be too loose.

Second, the globally synchronized frame recycling in GSF limits its ability to utilize excess bandwidth. Figure 1 shows an example pathological scenario. The grey nodes on the first column of the network are sending packets to the black node (hotspot) at the center of the network. The stripped node, in contrast, is sending to its nearest neighbor. Without prior knowledge of the actual traffic pattern (which is typically the case for CMP), all the nodes may be assigned equal reservations in the frame. In the actual traffic pattern, the stripped node could exploit full link speed since it experiences...
no contention. However, the pressure created on the hotspot node slows down the global frame recycling, which indirectly reduces the accepted throughput of the stripped node.

Third, GSF reduces the efficiency of virtual channel flow-control. To prevent priority inversion, GSF does not allow flits belonging to different packets to reside in the same virtual channel. This requirement essentially elongates the time to return virtual credits and further compromises the network throughput.

Our work is also related to the original work on flit-reservation by Peh et al. [20]. In Peh’s work, flit-reservation is used entirely to improve the average performance, while in our work it is integrated into frame-based scheduling to offer guaranteed service.

3. Principles of LSF and FRS

Our proposed LOFT architecture is based on the integration of locally-synchronized frames (LSF) and flit-reservation flow-control (FRS). In this section, we separately present the principles for LSF and FRS, but defer the discussion of integration of LSF and FRS to the next section.

3.1. Frame-Based Scheduling

Both GSF [12] and locally-synchronized frames (LSF) are based on the idea of grouping time slots into frames, to coarsely approximate the ideal deadline-based scheduling [21]. The term “scheduling” in the NoC sense is to determine the switching order of flits contending for an output physical link. Figure 2a showcases how contentions in the NoC can be abstracted by a server model. As can be seen, “server” here simply refers to each physical link, and the “service” it provides is the link bandwidth. At each server, queues (buffers) are provided to account for dynamic mismatch between arrival and service rates. Scheduling is performed conceptually by a “MUX”, which at each cycle dequeue one flit from a certain queue to service. Therefore the de queuing order determines the service order.

Using this model, frame-based scheduling can be explained with the help of Figure 2b and 2c. First, Figure 2b shows the ideal setup. At each “MUX” (the scheduling point), the buffers are divided into isolated queues, the frames. Each frame has a size of $F$ flits. Frames are associated with a frame number, and the smaller this number is, the higher priority the corresponding frame has. The “MUX” services the flits in the frames in the decreasing order of their priorities. That is, “MUX” will first dequeue flits from frame 0, and only after all flits in frame $k$ are drained, the flits in frame $(k + 1)$ can be served. A head frame pointer points to the frame currently being serviced, the head frame. Figure 2c shows a more practical setting, where there is a finite window of frames, which has $WF$ frames. The head frame pointer walks across frames in the round-robin order.

Incoming flits are queued into frames as follows. For each contending $flow_{ij}$ at “MUX”, it is assigned a fixed number of buffer slots, denoted by its allocated reservation $R_{ij}$, from each frame. The sum of $R_{ij}$s cannot be larger than $F$. $flow_{ij}$ can only inject up to $R_{ij}$ flits of its incoming flits into a frame, starting from the highest priority frame. The flow is throttled if its injection frame hits the head frame; that is, its reservations in all the frames have been used up. A throttled flow can restart to inject flits when the head frame pointer advances, at which time the frame window is shifted by 1. Frame window shifting effectively recycles the oldest frame to a fresh one.

Allocating buffer slots inside each frame essentially divides the total bandwidth into shares. If we assume that the “MUX”
can dequeue 1 flit at a cycle, and ∑ R_{ij} ≤ F, then the throughput allocated to flow_{ij} is R_{ij}/∑ R_{ij} of the ideal bandwidth. In addition, since at most only ̃F flits can be injected into a frame, the time to drain the head frame is upper bounded.

While based on the same principle, GSF and LSF fundamentally differs in where frames are managed. GSF abstracts the whole NoC as a single “MUX”. In GSF, frame window shifting is globally synchronized by a barrier network. While this saves some complexities in the router, it also incurs serious problems as discussed in Section 2.2. In contrast, in LSF each output link of a router manages frame recycling of its own frames regardless of other output links or routers. The benefit of doing so will become clear from the following sections. In addition, LSF allows for integration of flit-reservation, an efficient flow-control mechanism to improve performance.

3.2. Flit-Reservation

Flit-reservation flow-control (FRS), originally proposed by Peh et al [20], is an integral part of LOFT. It meshes with LSF well since it is also a distributed resource scheduling mechanism. As [20] presented a thorough discussion, here we only briefly reiterate the key ideas.

FRS proposes using look-ahead flits to pre-schedule bandwidth and buffers for data flits, and thus obliterates the need of arbitration and flow-control for data flits. A look-ahead network is dedicated to route look-ahead flits. The look-ahead network is simply a second physical network overlaid on the data network. Before the injection of any data flit, a leading look-ahead flit must be injected into the look-ahead network to reserve resources for the data flit.

Figure 3 shows the format of look-ahead flits. The look-ahead flit contains the departure times (t_d) and the flit numbers (flt_no) for the data flits it leads. A look-ahead flit can lead only data flits belonging to a single flow. A source field uniquely identifies the source node of the flow, and a destination field identifies the destination node. Each flt_no field uniquely identifies a data flit led by the look-ahead flit, and each t_d following the flt_no records the departure time of that data flit from the previous router. While originally in [20] flt_no and the source field are not needed, for integration with LSF we introduce these two fields, whose usage will be clear in Section 4.

The router architectures for both the look-ahead network and the data network are shown in Figure 4. The router of the look-ahead network closely resembles the generic wormhole router with virtual channels and a sequence of routing stages: route computing (RC), virtual channel allocation (VA), switch allocation (SA), and switch traversal. In addition, each output/input port of the router contains an output/input scheduler. The output scheduler schedules the departure times of data flits to the next router. The input scheduler allocates space in the input buffer for data flits, and schedules data flits to traverse the switch to output ports in time before departure. Both schedulers program the scheduled events in the output and input reservation tables respectively.

The data network router contains an output reservation table in each output port, and an input reservation table in each input port. These tables contain the scheduling results from the schedulers to manipulate the movement of data flits. In addition, the input buffer is arranged as a central buffer, rather than virtual channels.

The formats of output reservation table and input reservation tables are shown in Figure 5. The output reservation table keeps track of the status of the output port within a time window, which in this figure has 8 time slots. For each time slot, it contains a busy flag to record if the output port is busy. It also contains a field to record the number of free buffer slots.
The look-ahead flit arrives at a router, and writes the frame to the input buffer of the next router. We refer to this field “virtual credits” as it represents free buffers in future time slots rather than the current time slot. The grey arrow in the figure indicates the current time slot. The input reservation table is organized similarly, recording the information of arriving data flits (flow and flit number). In addition, it stores buffer allocation, output ports, and switching times of arriving data flits.

The procedure of scheduling the departure time of a data flit is sketched as follows:
1) The look-ahead flit arrives at a router, and writes the information of data flits it leads to the input reservation table. Buffers are also allocated for the data flits.
2) The look-ahead flit passes the router pipeline as in an ordinary router.
3) When the look-ahead flit reaches the output port, it tries to schedule the earliest departure times (when the output port is not busy and the virtual credit count is positive). Upon successful scheduling, it updates the status of the output reservation table.
4) The output scheduling result is returned to the input scheduler, which updates the switching time and the output port. The input scheduler also returns virtual credits to the previous router.
5) When data flits arrive, the input scheduler lazily allocates buffer slots for them.

For example, Figure 5 shows the status of reservation tables after scheduling one data flit which arrives at cycle 2 and is scheduled to departure at cycle 5 from the east port.

Figure 6 presents a comparison of flow-control mechanisms in conventional wormhole switching, GSF, and FRS, which highlights the efficiency of FRS. This time graph shows the back-to-back transfer of 4 packets each having 4 flits between two routers when the input buffer is close to full. As can be seen, even assuming 1-cycle credit turn-around time, flow-control introduces a non-negligible throughput overhead to wormhole switching. This overhead is even worse for GSF. On the other hand, FRS can achieve zero turn-around time and has the highest efficiency. In addition, as the look-ahead flits are short and a single look-ahead flit can schedule multiple data flits, the look-ahead network is lightly loaded and fast.

4. Integration of LSF and FRS

Locally-synchronized frames and flit-reservation share fundamental similarities. Both mechanisms are forms of distributed resource scheduler that allocates buffer space and link bandwidth. In addition, the central buffer used by FRS can be readily used to emulate the frame queues used by LSF. The similarities motivate us to combine the two mechanisms for a high performance network-on-chip providing guaranteed services. We discuss the integration of LSF and FRS in this section.

4.1. Combining Local Frames and Reservation Tables

The first step we take is to divide the output reservation table, rather than input buffers, into frames (Figure 7). The frame size $F$ now equals to the number of time slots in each segment of the framed output table. A share of time slots, denoted as $R_{ij}$, from each frame is assigned to each flow $ij$ contending for the output. Using the same terminology as frame-based scheduling, successfully scheduling a time slot in a frame $k$ is referred to as injecting a data flit into frame $k$. As with frame-based scheduling, each flow $ij$ can only inject up to $R_{ij}$ data flits into each frame, and it is required that $\sum R_{ij} \leq F$.

Each output scheduler maintains a current time slot pointer (denoted as $CP$) indicating the current time slot, and a head frame pointer (denoted as $HF$) pointing to the frame being serviced. Each flow $ij$ contending for the output link maintains its current injection frame $IF_{ij}$ and remaining reservation $C_{ij}$. $WF$ is used to denote the frame window size and $WT$ is used to denote the time window size, where $WT = F \times WF$.

Algorithm 1 formally shows the injection procedure for $flow_{ij}$. When the network is powered-up, $IF_{ij}$ is initialized to $HF$ and $C_{ij}$ is initialized to $R_{ij}$. Upon each scheduling request from look-ahead flits belonging to $flow_{ij}$, it is first checked if $C_{ij}$ is positive. If so, procedure $try\_schedule()$ is invoked to try to inject the data flit into frame $IF_{ij}$. Upon successful scheduling, $C_{ij}$ is decremented by one; otherwise, $IF_{ij}$ is incremented and $C_{ij}$ is updated to the minimum of
Algorithm 1 Injection procedure for flow

1: Initialize: IF_{ij} ← HF
2: Initialize: C_{ij} ← R_{ij}
3: UPON EVERY REQUEST FROM LOOK-AHEAD FLITS:
4: scheduled ← FALSE
5: while scheduled = FALSE do
6: if C_{ij} > 0 then
7: scheduled ← try_schedule(IF_{ij})
8: end if
9: if scheduled then
10: C_{ij} ← C_{ij} - 1
11: else
12: if (IF_{ij} + 1) Mod WF ≠ HF then
13: C_{ij} ← MIN(R_{ij}, C_{ij} + R_{ij})
14: IF_{ij} ← (IF_{ij} + 1) Mod WF
15: else
16: break
17: end if
18: end if
19: end while

Algorithm 2 try_schedule(IF_{ij})

1: if IF_{ij} = HF then
2: candidate ← CP + 1
3: else
4: candidate ← F × IF_{ij}
5: end if
6: scheduled ← FALSE
7: while scheduled = FALSE and candidate ≠ ((IF_{ij} + 1) Mod WF) × F do
8: if output_table(candidate).busy = FALSE and
   output_table(candidate).virtual_credits > 0 then
9: scheduled ← TRUE  // Scheduling is successful if a valid
   slot is found
10: else
11: candidate ← candidate + 1
12: end if
13: end while
14: if scheduled then
15: Update output and input reservation tables
16: end if
17: return scheduled

Algorithm 3 Manipulation of CP and HF

1: Initialize: CP ← 0 HF ← 0
2: AT EVERY CLOCK TICK:
3: CP ← (CP + 1) Mod WF
4: IF CP Mod F = 0 then
5: for all flow_{ij} containing for the output link do
6: if IF_{ij} = HF then
7: IF_{ij} ← (IF_{ij} + 1) Mod WF
8: C_{ij} ← MIN(R_{ij}, C_{ij} + R_{ij})
9: end if
10: end for
11: HF ← (HF + 1) Mod WF
12: end if

Fig. 8. An example showing output scheduling anomaly. The entry tagged by “N” shown in (c) indicates buffer underflow.

4.2. Output Scheduling Anomaly

While the construction in the previous subsection seems plausible, it has an inherent severe problem, which is explained by the following example.

Consider a simplified case where two flows flow_{ij} and flow_{mn} contend for an output link. Suppose F = 4, WF = 4, and the size of input buffer is 4 flits. We equally distribute the link bandwidth to the two flows; that is, R_{ij} = R_{mn} = 2. The initial status of the output reservation table is shown in Figure 8a.

Now suppose two look-ahead flits from flow_{ij} arrive in the first two cycles, and each of them leads two data flits. The first look-ahead flit will successfully schedule two time slots in the first frame. Then the second look-ahead flit finds its reservation in frame 0 is used up. Therefore it advances IF_{ij} and injects two data flits in the second frame. Furthermore, the virtual credits consumed in the first frame are not returned by slot 7 (this is possible, for example, due to contention in the next hop). The output reservation table immediately after the previous scheduling events is shown in Figure 8b (shaded entries indicate changes). It shows that the next input buffer

R_{ij} and (C_{ij} + R_{ij}), until the reservations in all frames have been exhausted.

Algorithm 2 defines procedure try_schedule(). In the algorithm, we use output_table(k).busy and output_table(k).virtual_credits to refer to the busy flag and the virtual credit count of entry k in the output reservation table. We define a valid time slot as the slot with a false busy flag and a positive virtual credit count. Procedure try_schedule() only returns true if a valid time slot is found in the given injection frame.

Algorithm 3 shows the procedure to shift the frame window. Initially CP and HF are both assigned “0”. At each cycle CP is updated to (CP + 1) Mod WT, while HF is updated to (HF + 1) Mod WF for every F cycles.

It is worth to mention that GSF [12] does not allow flows to inject into the head frame, in order to guarantee that the head frame is drained in a finite time. In contrast, in our work injection to head frame is permitted, as it is guaranteed that the head frame can be recycled for every F cycles.
is full at slot 5 with zero virtual credits. A problem may occur if a look-ahead flt from \( flow_{mn} \) arrives in the third cycle, leading one data flt. This look-ahead flt will inject to frame 0, since it has \( C_{mn} > 0 \) and finds slot 3 is valid. However, the virtual credit count at slot 5 will become negative after \( flow_{mn} \) injects to slot 3 (Figure 8c)! The outcome is disastrous since the buffer is “silently” overbooked without any flow being aware. At cycle 5, the actual delivery of the scheduled data flt may fail due to insufficient buffer.

We call this problem output scheduling anomaly, which is a direct consequence of out-of-order scheduling as shown in the above example. An aggressive flow can schedule buffer slots in more distant future, as forced by frame-based scheduling. A more moderate flow not aware of the aggressive flow may inject to a more imminent time slot, which may cause buffer underflow in the future. There are two straightforward approaches to address the problem. The first approach prevents injection into a time slot if any future time slot has zero credits. The second approach enforces in-order scheduling by requiring new data flts to be scheduled in more distant time slots than any already scheduled data flts. While either approach eliminates the anomaly, both of them will force a moderate flow to discard unused reservations in more imminent frames. This would cause bandwidth under-utilization and break the fairness of scheduling. A better solution to the anomaly problem should let aggressive flows voluntarily yield buffer space to moderate flows.

Our solution to the problem introduces an additional counter \( \text{skipped}(i) \) associated with each frame \( i \), where \( i \in [0, WF - 1] \). Informally, \( \text{skipped}(i) \) records the total yielded reservations by all flows. The manipulation of \( \text{skipped}(i) \) is as follows:

1. Upon initialization, \( \text{skipped}(i) \) \( \leftarrow 0 \).
2. When the injection pointer \( IF_{ij} \) of a \( flow_{ij} \) advances, \( \text{skipped}(IF_{ij}) \) is incremented by \( C_{ij} \). That is, we add \( \text{skipped}(IF_{ij}) \) \( \leftarrow \text{skipped}(IF_{ij}) + C_{ij} \) after the if statement in line 12 of Algorithm 1.

The newly added counter \( \text{skipped}(i) \) is used as follows. For an arbitrary flow \( flow_{ij} \), let \( Prior \) be the index of the entry immediately prior to the injection frame \( IF_{ij} \) in the output reservation table. That is, \( Prior \) equals to \( (IF_{ij} \times F + WT - 1) \mod WT \). Then \( flow_{ij} \) can only inject flts into frame \( IF_{ij} \) if the following condition holds:

\[
F - \text{skipped}(IF_{ij}) \leq \text{out\_table}(Prior).\text{virtual\_credit} \quad (1)
\]

Condition (1) is appended to the conditions of the if statement at line 6 of Algorithm 1. With this change, output scheduling anomaly is eliminated when the size of input buffer is \( F \) flts, as stated by Theorem 1 in the Appendix. In addition, aggressive flows will voluntarily yield buffer space to moderate flows to ensure fair allocation in buffer space. Detailed discussion of condition (1) can be found in the Appendix.

Reconsider our example with the addition of condition (1). Now \( flow_{ij} \) cannot inject into frame 1 since \( F - \text{skipped}(1) = 4 > \text{out\_table}(3).\text{virtual\_credit} = 2 \). Therefore it will advance \( IF_{ij} \) to 2, and as a result \( \text{skipped}(1) = C_{ij} = 2 \). Since now \( F - \text{skipped}(2) = 4 \leq \text{out\_table}(7).\text{virtual\_credit} = 4 \), \( flow_{ij} \) can inject two data flts into frame 2. After that, when \( flow_{mn} \) schedules slot 3, no buffer underflow will occur.

### 4.3. Optimizations

The techniques presented so far frame output reservation tables to provide fair bandwidth allocation. However, although each output scheduler independently manages output reservation tables and frames, frame recycling is still globally synchronized across all routers using Algorithm 3. Consider the stripped node in Figure 1. Despite the ample bandwidth it could leverage, on average it can only inject \( R_{ij} \) flts for every \( F \) cycles and the bandwidth is under-utilized. In this subsection, we propose optimizations to improve utilization and overall throughput. The first optimization speculatively forwards flts to reduce latency. The second optimization breaks the global synchronization among schedulers to exploit excess bandwidth.

#### 4.3.1. Speculative Flit Switching

In the present switching mechanism, data flts are switched at the time slot scheduled by look-ahead flts, even when no other flts are scheduled before them. This unnecessarily prolongs the delivery latency when the network is lightly loaded.

To address this problem, we modify the switching mechanism to forward data flts as soon as possible. A data flt can be forwarded to the next router if there is no on-going transmission on the output link and if the next buffer has free space. This requires two changes to the data network router shown in Figure 4. First, additional signals are added to return actual credits in the input buffer. Second, an output arbiter similar to the switch arbiter is needed to pick one flt from the ready data flts to forward. This arbiter can be a simple round-robin arbiter, since its choice will not compromise scheduling made by LSF and FRS, except for one case explained below.

The input scheduler picks a data flt that has arrived (input buffer has been allocated) with earliest scheduled departure time as the candidate for switching. In practice, it just picks the first non-empty entry in the “buffer out” row of the input reservation table. If the candidate wins the output arbitration, it will be forwarded and its entry in the input and output reservation tables will be cleared. The input scheduler will mark a candidate as emergent if it is scheduled to be forwarded in the current cycle. Emergent candidate is guaranteed to win arbitration. This causes no problem since an output port can only have at most one emergent candidate for each cycle. This is the only exception for the round-robin arbiter.

Two caveats deserves explanation. First, as now the data flts may arrive ahead of the scheduled time (but still after the look-ahead flt), the input reservation table needs to record the flow number and flt number, in order to uniquely identify arriving data flts. This information is provided by the look-ahead flt.

The second problem is more subtle but severe. With the modified switching mechanism, data flts may be serviced in a different order other than the scheduled order. This brings...
back the problem of output scheduling anomaly, where out-of-order forwarding may cause buffer underflow at a later time. The outcome is the potential risk that emergent flits miss their switching times due to insufficient buffer. We solve this problem by adding a “speculative buffer” to each input port, as shown in Figure 9. The speculative buffer is used to hold data flits forwarded out of order, and the original non-speculative buffer is used for in-order flits. The output reservation table only tracks buffer usage of the non-speculative buffer. For a winning candidate, the output scheduler checks if it is the first scheduled flit in the output reservation table. If so, the data flit is forwarded to the non-speculative buffer; if not, it is forwarded to the speculative buffer. In either case, if the corresponding buffer is full the data flit will be denied access to the output link. At the other end of the link, the input scheduler allocates space in the speculative buffer for speculative flits and vice versa. Using a separate speculative buffer prevents out-of-order flits from blocking emergent flits.

4.3.2. Local Status Reset. Speculative forwarding only saves latency but not improves throughput. A flow cannot inject more data flits once its reservation in a frame window is used up, even if all scheduled data flits have been delivered by speculative switching. The intrinsic bottleneck to throughput is the constant frame recycling rate for both lightly loaded and heavily loaded links. Thus we propose to do local status reset when an output link is idle. During a local status reset event, \( a) \CP_i, \HF_i \xleftarrow{0}, b) \IF_i, \C_i \xleftarrow{1}, \text{for all contending flows}\), and \( c) \text{out_table(i).virtual\_credit} \xleftarrow{B_N}, \forall i \in \{0, \WT - 1\}, \text{where } B_N \text{ is the size of the non-speculative buffer.} \) The reset event is triggered on an output link when the conditions below are both met to ensure the reset is safe:

- All the busy flags in the output reservation table are \{false\}.

- The non-speculative buffer in the next input port is empty. This can be checked by the credits returned from the non-speculative buffer.

Local status reset essentially recycles all the frames in the frame window to be fresh frames. It reduces the idle time of output link and improves utilization. Note that local status reset also breaks the global synchronization of frame recycling, allowing lightly loaded links to recycle frames at faster rates.

5. Simulation Setup

5.1. LOFT

We model LOFT adhering to the principles presented in Section 4 in a cycle-accurate NoC simulator [22]. Our current model simulates a 8×8 mesh topology. Each node is numbered by \((x + y \times 8)\) according to its coordinates \((x, y)\). The injection rates of the source nodes are variable for the purpose of performance study. In contrast, the ejection rate of any destination is constant, 1 flit/cycle.

The detailed specifications are generalized in Table 1. Each data packet has 4 data flits, which are further partitioned to two 2-flit data quantum. Each look-ahead flit leads a single data quantum, which is scheduled in its entirety to simplify scheduling. With \( F = 256 \) and \( WF = 2 \), this implies a reservation table size of \( F \times WF/2 = 256 \) entries. In addition, we assume that the maximum number of flows contending for a link is 64. With deterministic routing, we further assume that a flow uses the same reservation \( (R_i) \) for all links of its path, and that for all contending \( \text{flow}_{ij} \) of a link, \( \sum R_{ij} \leq F \).

5.1.1. Look-ahead Network. The router architecture of look-ahead network adheres to that shown in Figure 4. We add one output scheduling stage (input scheduling is not on the critical path) to a baseline 2-stage wormhole router with look-ahead routing and speculative switch allocation [23]. Each output scheduler maintains the status (injection frame \( I F_i, \text{allocated reservation } R_{ij}, \text{and remaining reservation } C_{ij} \)) for 64 flows as well as the head frame pointer \( HF \) and current slot pointer \( CP \) shared by all flows.

In the current setup, each look-ahead flit leads a single data quantum composed by 2 data flits. Therefore the flit number (departure time) simply becomes the quantum number (departure time). The 32-bit look-ahead flit contains a 6-bit destination field, a 6-bit flow number field, a 10-bit quantum number field, and a 10-bit departure time field.

5.1.2. Data Network. The router in the data network also has 3 stages: input buffer allocation, output request and arbitration, and switch traversal. The depth of the central buffer (non-speculative) is set to the same as the frame size to eliminate
output scheduling anomaly (see Section 4.2). The depth of the speculative buffer is varied from 0 to 16 flits, and the impact of its size is studied.

Each data flit is 128-bit wide, same as the link width. The unique flow number and flit number is stored in the first 16 bits. Note that data flits do not contain any routing and scheduling information, as routing and scheduling has been done by the leading look-ahead flits.

5.2. GSF

For comparison purpose, we also implement GSF in the simulator. The suggested parameters in [13] and [19] are used, which are shown in Table 1. The network size, topology, injection and ejection rates, and routing algorithm are the same as LOFT. To maintain an acceptable performance, GSF requires a source queue as large as the frame size (2000 flits) for each node.

5.3. Delay Bounds and Hardware Cost

From the specifications in Table 1, we can calculate delay bounds and hardware cost of both LOFT and GSF.

5.3.1. Delay Bounds. As explained by [12], injected packets in GSF are guaranteed to be drained in one frame window. However, a tight bound of the period of the frame window is difficult to estimate, and the original work [12] relies on simulation to obtain an empirical bound. Nevertheless, a worst-case estimation can be made by assuming a frame window full of flits are injected to node 0 and destined to node 63. Then the worst-case time to drain a frame window is over \((k \times WF \times F)\). The factor \(k\) is due to the flow-control overhead (Section 3.2) and equals to 2 with the router architecture we model. Therefore the worst-case latency amounts to 24000 cycles. This bound is not only too long but regardless of paths taken by data flits. In contrast, LOFT provides much tighter bounds. In the worst case, all links in the network are heavily loaded, such that speculative switching and local status reset are not effective. In this case the worst-case end-to-end latency of LOFT is the same as that of RCQ [18]:

\[
F \times WF \times Num_{Hops}
\]

which amounts to 512 cycles per hop in our setting. Compared to GSF, this worst-case latency is not only much tighter but related to the paths taken by the flows.

5.3.2. Hardware Cost. As shown by Table 2, LOFT uses 32% less storages than GSF (assuming a 12-flit speculative buffer for LOFT). In addition, for a rough estimation of area and power, we use McPAT [24] with configurations to emulate the LOFT router (e.g., a worm-hole NoC router with one virtual channel and 256-flit input buffer depth). The area and the power of a 64-node LOFT NoC are estimated to be 32mm² and 50W, which is 7% of a 64-node CMP [25] and 19% of total chip power (265W estimated by McPAT).

<table>
<thead>
<tr>
<th>Source queue</th>
<th>Virtual channels</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSF</td>
<td>256000</td>
<td>15360</td>
</tr>
<tr>
<td>LOFT</td>
<td>40660</td>
<td>2308</td>
</tr>
</tbody>
</table>

6. Experiment Results

We measure and compare the quality-of-service metrics of LOFT and GSF, using the simulation setup presented in the previous section. In the simulation we run 4 types of synthetic traffic patterns: uniform, hotspot, and two pathological cases that evaluate the quality of performance isolation. For uniform traffic, each source is treated as a separate flow, while in other cases each source-destination pair is treated as a distinct flow. For hotspot traffic, all sources send packets to node 63. The configurations of the two pathological cases will be discussed later with their results. We run each simulation until a stable network state is reached. For clarity, in this section we use \(flow_{i \rightarrow j}\) to refer to \(flows_{ij}\).

6.1. Fairness

We first evaluate the general fairness of LOFT. Figure 10a, 10b and 10c show the results for equal and differentiated allocations respectively for hotspot traffic. For equal allocation, the bandwidth is distributed equally to all flows; in the two differentiated allocation cases, the network is divided into 4 and 2 partitions, and differentiated services are provided to different partitions. It can be seen that LOFT achieves excellent fairness in bandwidth allocation as GSF [12]. We repeat the same experiment for uniform pattern and confirm that the actual bandwidths received by different flows also comply with the allocation.

6.2. Performance

For performance evaluation, two metrics are used: a) average packet latency against offered load rate, and b) network throughput. We obtain the metrics from the simulation of uniform and hotspot traffics. For LOFT, we also vary the size of the speculative buffer to study its impact on performance. Note that setting the speculative buffer size to 0 is equivalent to turning off all optimizations proposed in Section 4.3. For comparison purpose, the performance of GSF is also measured and included in the results.

Figure 11a shows the result for uniform traffic pattern. The first observation from the result is that the average packet latency levels out when the offered load increases beyond a certain point for both LOFT and GSF. This is an expected result as to provide guaranteed service, both LOFT and GSF regulate the injection rates of flows and prevent unbounded packet latency. Second, we observe that increasing the speculative buffer size helps improve LOFT’s performance, and LOFT outperforms GSF when the speculative buffer size is at least 8. However, increasing the speculative buffer size beyond 12 flits sees only marginal gains.
Fig. 10. Fairness of throughput allocation for hotspot traffic pattern. The tables show the maximum, minimum, average, and standard deviation of throughputs for each group of flows.

Figure 11b shows the results for hotspot traffic pattern. The performance of LOFT is even better than GSF in this case, where network contention is more fierce. The performance of LOFT is better for all speculative buffer sizes. However, different from the uniform case we notice that the speculative buffer size does not significantly impact the performance. This is due to the fact that with high contention on the links speculative switching is least effective.

According to the result above, we choose a 12-flit speculative buffer size, and use this size for the rest of experiment.

6.3. Case Studies of Performance Isolation

We study two cases to exam the effectiveness in performance isolation of LOFT compared with GSF.

a) Case Study I is adapted from the hotspot traffic to model the denial-of-service attack. In this case, only nodes 0, 48, and 56 actively send packets to the hotspot, node 63. Each flow is allocated 1/4 of the link bandwidth, that is, 0.25 flits/cycle. flow\_{0\rightarrow63} is a regulated flow, injecting at a constant average rate of 0.2 flits/cycle. flow\_{48\rightarrow63} and flow\_{56\rightarrow63} are aggressors. The injection rates of them varies significantly, and may be well beyond the allocated rate. We assume that both aggressors are injecting at same rates, and measure the performance of each flow versus the injection rate of aggressors.

The results for GSF is shown in Figure 12a. We notice that with the injection rate of aggressors increasing, the average packet latency of all three flows rises significantly. The latency for the regulated flow increases from 60 cycles up to 2000 cycles, which shows that the performance of the regulated flow is severely degraded by the aggressors. In addition, the aggressors also cause the accepted throughput of the regulated flow to slightly decline. Finally, we see that the aggregate throughput is always below 60% of link bandwidth, due to the inefficiency of the flow-control mechanism.

Compared to GSF, the robustness of LOFT is more satisfying as shown in Figure 12b. The packet latencies of all flows do not significantly increase until the aggressor injection rate reaches 0.4 flits/cycle. Beyond that point, all flows see a packet latency increase. However, the latency increase (from 42 cycles to 55 cycles) of the regulated flow is to a much less extent than the aggressors (from ~25 cycles to >=400 cycles). Therefore LOFT provides better performance isolation than GSF, and tends to penalize aggressors that try to exhaust bandwidth. In addition, the network utilization of LOFT (over 90%) is much higher than GSF.
7. Conclusion

Providing quality-of-service in network-on-chip is challenging due to the distributed scheduling nature. In this paper we present LOFT, a high performance network-on-chip that provides guaranteed service and overcomes the weaknesses of previous approaches. LOFT combines locally-synchronized frames, where each output port independently implements frame-based scheduling, and flit-reservation, a pre-scheduling mechanism for efficient flow-control. We compare LOFT with globally-synchronized frames, another state-of-the-art QoS framework for NoC. The experiment results show that LOFT achieves equally satisfying fairness, better robustness against denial-of-service attack, and higher network utilization than GSF. We believe LOFT serves as a promising candidate for guaranteed service NoC architectures.

Appendix

Discussion of Condition (1)

In this section, we first formally prove the following theorem.

**THEOREM I.** With the constraint of condition (1), if the input buffer size is \( F \) flits, then at any moment of time and any output port, \( \text{out\_table}(i)\text{.virtual\_credit} \geq 0, \forall i \in [0, W T − 1] \).

Before proving Theorem I, we introduce some necessary mathematical tools:

**Preliminaries**—First, for the moment we assume infinite frame window and time window. This transformation does not affect the soundness of proof, since we can view the infinite frame/time window as the concatenation of infinite fixed-size frame/time windows. It only removes the ambiguity introduced by wrap-arounds and allows us to conveniently and uniquely identify a frame/time slot. Also for convenience, we use *time slot* \( (X, a) \) \((a \in [0, F − 1])\) to refer to the \((a + 1)\)th time slot in frame \( X \).

Furthermore we define

\[
\begin{align*}
    b(X, a) &= \text{number of data flits scheduled in slot } (X, a) \\
    B(X) &= \sum_{a=0}^{F-1} b(X, a)
\end{align*}
\]

Note that for all valid \( X \) and \( a \), \( b(X, a) \) is either 0 or 1, and \( B(X) \geq 0 \). We further define

\[
    u(X, a) = \text{number of returned virtual credit at slot } (X, a) \\
    U(X) = \sum_{a=0}^{F-1} u(X, a)
\]

and for all valid \( X \) and \( a \), \( u(X, a), U(X) \geq 0 \). Given the above notations and supposing the input buffer size is \( F \) flits, the virtual credit count of some time slot \( (X, a) \) can be calculated as:

\[
\text{out\_table}(X, a)\text{.virtual\_credit} = F - \sum_{i=0}^{X-1} B(i) - \sum_{j=0}^{a} b(X, j) + \sum_{i=0}^{X-1} U(i) + \sum_{j=0}^{a} u(X, j) \tag{3}
\]

Using equation (3), condition (1) can be transformed into the following equivalent form

\[
\text{skipped}(1F_j) \geq \sum_{i=0}^{1F_{i,j}-1} B(i) - \sum_{i=0}^{1F_{i,j}-1} U(i) \tag{4}
\]
Frame Region—We define frame regions as non-overlapping groups of adjacent frames. Figure 14 shows an example of three frame regions. By our convention, a frame region has to contain at least one frame. In addition, we number the frame regions with the frame numbers of the first frames they contain.

Furthermore, we introduce the following notations:

\[
FR(k) = \text{Frame Region } k \\
B^{FR}(k) = \sum_{X \in FR(k)} R_{ij} \\
U^{FR}(k) = \sum_{X \in FR(k)} U(X)
\]

Due to space limits, we state the following lemma without a formal proof.

**Lemma I.** With the constraint of condition (1), if the input buffer size is \( F \) flits, then for frame region \( FR(0) \) containing the first frame (frame 0), \( R^{FR}(0) \geq B^{FR}(0) - U^{FR}(0) \).

Now we are ready to prove Theorem I with the help of Lemma I.

**Proof of Theorem I**—Take an arbitrary entry \((X, a)\) in the output scheduling table. We construct a frame region \( FR(0) \) containing frame 0 to frame \((X - 1)\), and a frame region \( FR(X) \) containing only frame X. From Lemma I, we know that \( R^{FR}(0) \geq B^{FR}(0) - U^{FR}(0) \). In addition, it is straightforward to see that \( \sum_{j=0}^{a} b(X, j) \leq B(X) \leq R^{FR}(0) + R^{FR}(0) - \text{skipped}(X) \). Recall that with condition (4), we have \( \text{skipped}(X) \geq B^{FR}(0) - U^{FR}(0) \). In summary, the virtual credit count of slot \((X, a)\) can be calculated as follows.

\[
\text{out_table}(X, a).\text{virtual_credit} = F - B^{FR}(0) + U^{FR}(0) - \sum_{j=0}^{a} b(X, j) + \sum_{j=0}^{a} w(X, j) \\
\geq F - B^{FR}(0) - R^{FR}(X)
\]

Since by construction \( R^{FR}(0) + R^{FR}(X) \leq \sum R_{ij} \leq F \), \( \text{out_table}(X, a).\text{virtual_credit} \geq 0 \) is true for any valid \( X \) and \( a \) and Theorem I is proven.

In addition, condition (4) implies that aggressive flows cannot inject into a frame if the virtual credits consumed in previous frames have not been returned; instead, the aggressive flow \( flow_{ij} \) will advance its injection frame pointer and thus \( \text{skipped}(IF_{ij}) \) is increased by \( C_{ij} \). Effectively, **aggressive flows voluntarily yield buffer space to moderate flows when condition (1)/(4) is not satisfied.**

**References**


