

# A 4.6Tbits/s 3.6GHz Single-cycle NoC Router with a Novel Switch Allocator in 65nm CMOS

Amit Kumar<sup>†</sup>, Partha Kundu<sup>‡</sup>, Arvind P. Singh<sup>§</sup>, Li-Shiuan Peh<sup>†</sup> and Niraj K. Jha<sup>†</sup>

<sup>†</sup>Dept. of Electrical Engineering, Princeton University, Princeton, NJ 08544

<sup>‡</sup>Microprocessor Technology Labs, Intel Corp., Santa Clara, CA 95052

<sup>§</sup>Intel Technology India Pvt Ltd., Airport Road, Bangalore, India 560017

<sup>†</sup>{amitk, peh, jha}@princeton.edu, <sup>‡</sup>{partha.kundu}@intel.com, <sup>§</sup>{arvind.p.singh}@intel.com

## Abstract

*As chip multiprocessors (CMPs) become the only viable way to scale up and utilize the abundant transistors made available in current microprocessors, the design of on-chip networks is becoming critically important. These networks face unique design constraints and are required to provide extremely fast and high bandwidth communication, yet meet tight power and area budgets. In this paper, we present a detailed design of our on-chip network router targeted at a 36-core shared-memory CMP system in 65nm technology. Our design targets an aggressive clock frequency of 3.6GHz, thus posing tough design challenges that led to several unique circuit and microarchitectural innovations and design choices, including a novel high throughput and low latency switch allocation mechanism, a non-speculative single-cycle router pipeline which uses advanced bundles to remove control setup overhead, a low-complexity virtual channel allocator and a dynamically-managed shared buffer design which uses prefetching to minimize critical path delay. Our router takes up 1.19 mm<sup>2</sup> area and expends 551 mW power at 10% activity, delivering a single-cycle no-load latency at 3.6GHz clock frequency while achieving a peak switching data rate in excess of 4.6Tbits/s per router node.*

## 1 Introduction

There is wide consensus, both in industry and academia, that multi-core chips are the only efficient way for utilizing the billions of transistors available in future technologies. CMPs have thus emerged as the *de facto* architecture for current and future microprocessors [4]. As we scale to many-core chips, buses and crossbars are unable to deliver the required bandwidth within a reasonable area or power envelope. On-chip networks have thus emerged as a promising interconnect fabric [5].

In this paper, we walk through the detailed design of our on-chip network targeted at a 3.6GHz shared-memory 36-node CMP in 65nm technology. Shared-memory workloads are highly sensitive to on-chip interconnect latency [11] while demanding high on-die bandwidth [9]. The specifications for the on-chip network are thus highly challeng-

ing: the fabric needs to be ultra-low-latency while supplying high bandwidth, at an aggressive clock of 3.6GHz, while being highly power- and area-efficient. These aggressive specifications prompt a design that differs fairly significantly in router pipeline, microarchitecture and circuit from existing state-of-the-art [15], as follows:

- A novel switch allocation scheme that achieves close to ideal matching efficiency (high bandwidth) while being implementable within a single cycle.
- A non-speculative single-cycle low-load router pipeline that uses advanced bundles to set up switch allocation in advance.
- A simplification of virtual channel allocation into a design where selection is done after switch allocation, which shortens the pipeline.
- A shared buffer design that uses register file cells and prefetching to shorten access latency.

Our 3.6GHz 6×6 mesh network router takes up 1.19 mm<sup>2</sup> area and expends 551 mW power at 10% activity while delivering a single-cycle no-load latency with a peak switching data rate in excess of 4.6Tbits/s per node and a peak injection/ejection data rate in excess of 920Gbits/s per node. Delay and power estimates of each router component are obtained using a detailed analysis in a 65nm process [2].

Prior works on on-chip network prototypes [3, 12, 18] for SoC platforms have very different design concerns from ours which is targeted at general-purpose CMPs. The most notable differences lie in our choice of a custom design flow versus a synthesis-based design flow [10], our targeting of an aggressive 3.6GHz clock in a 65nm process versus typically low-frequency, unpipelined routers in synthesized NoCs [16] and our being faced with shared-memory traffic which exerts stringent latency-throughput demands. While there have been prior network chip prototypes aimed at CMPs [8, 14, 20], their design targets are different and significantly less aggressive. Mullins et al. [14] propose a speculative single-cycle router clocked at 250MHz. In contrast, we propose a non-speculative single-cycle router pipeline clocked at 3.6GHz. In [20], a five-stage router design at 4GHz is presented with a peak injection/ejection data rate of 256Gbits/s per node. Our design is able to achieve a higher corresponding data rate of 920Gbits/s while using a

shorter router pipeline. The design in [8] uses control flits which travel ahead of data and help in hiding setup delay similar to advanced bundles in our design. However, the design in [8] targets an operand network, with the router pipeline directly integrated within the processor pipeline, so pipeline latency and complexity are much more critical than network throughput whereas our design is interfaced through cache controllers at each core and needs to handle the very high throughput demands of memory traffic.

The rest of the paper is organized as follows. Section 2 presents the proposed router pipeline and compares it to a state-of-the-art baseline. We then delve into the switch allocation pipestage and present our novel switch allocation mechanism in Section 3. Section 4 presents implementation details of the proposed design, analyzing in detail the critical path within each pipestage. In Section 5, we evaluate our design while Section 6 concludes the paper.

## 2 Proposed router pipeline

In this section, we first describe the router pipeline of a state-of-the-art baseline packet-switched design followed by our proposed pipeline. Figure 1 shows this packet-switched baseline router pipeline. When a flit arrives at a router, it is written into the buffer in the *switch setup* (SS) stage. In parallel, the flit sets up a request to the switch allocator based at its output port. Lookahead routing [7] is used to remove route computation from the critical path by calculating packet routes one hop in advance. The flit then goes through *switch allocation* (SA) which is the process of matching a flit to its requested output port. Speculation [15] is used to enable the header flit of the packet to arbitrate for both the switch port along with trying to acquire a free virtual channel (VC). If speculation succeeds, the flit enters the next pipestage. However, when speculation fails, the flit goes through these stages again, depending on where the speculation fails. If the flit was unable to acquire a VC, it goes through both VC allocation (VA) and SA again, while if it was successful in VA but not in SA, it has to retry SA only. Both VA and SA use a separable allocation scheme similar to the one in [15]. To avoid throughput degradation due to speculation, non-speculative requests (corresponding to flits which have already won a VC) are prioritized over speculative requests. Since each input port places a single output port request every cycle in the separable allocation mechanism used, the flit can be read out of the buffer in parallel with SA. This is followed by the *switch traversal* (ST) stage where the flit traverses the crossbar switch in this stage, followed by *link traversal* (LT), where it traverses the link to reach the next router. Under no-load, when router ports are empty, pipeline bypassing can be used to reduce the pipeline to two stages through the router: SS where the crossbar is set up for flit traversal, followed by ST. As opposed to the header flit, body and tail flits do not need to go through VA and inherit the VC allocated by the header. The tail flit deallocates the VC on leaving the router.

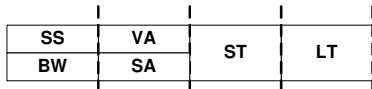
Figure 2 shows our proposed router pipeline. The different pipestages are described below:

**SA stage:** In our design, SA is done immediately after the flit arrives. This requires the output port of the packet to be known *a priori*. To accomplish this, our design uses an advanced bundle signal (sent for every flit) which arrives one cycle earlier than the actual flit payload. The function of this bundle is to set up requests to the switch allocator by communicating the output port information of an arriving flit. The advanced bundle encodes a flit’s entire route from its source to destination, assuming deterministic XY routing that is used in our design (use of advanced bundles, however, is not limited to XY routing). Hence, this bundle can be sent separately over a relatively narrow channel (for a  $6 \times 6$  mesh, this wiring overhead is six bits, which is around 4.6% of the forward wiring, assuming 16 byte wide links). However, the header flit of the packet now does not need to carry the packet’s route. In parallel with SA, the flit is also written to the buffer. In our design, VA and SA are done in the same cycle, but VCs are assigned non-speculatively after SA, as opposed to speculatively in the state-of-the-art scheme. VA simply involves finding a free VC for each output port from a pool of free VCs every cycle and assigning it to the flits which win SA. The selection of free VCs for each output port is done in the same cycle as SA but is not on the critical path (this is explained in Section 4.3). If there is no VC available, the flit goes to the back of the SA queue and retries. This has two distinct advantages: firstly, it leads to a much simpler hardware implementation of VA (using just a FIFO per port) as compared to the state-of-the-art schemes, and secondly, it reduces VC occupancy by avoiding cases where a flit wins VA but not SA (and hence keeps the VC occupied). For SA, we propose a novel high-throughput SA algorithm as described in Section 3.

**Buffer read (BR) stage:** Our proposed SA scheme allows each input port to place multiple output port requests every cycle. This disallows flit BR from taking place in parallel with SA, as our design uses single-ported input buffers to minimize router area and power. Hence, a separate BR stage is used where a flit is read out from its buffer after the SA result is known and set up into a latch in front of the crossbar. Simultaneously, the crossbar switch is set up by propagating control signals based on the input-output connections determined by SA in the previous cycle. In parallel, an advanced bundle carrying the route information of the winning flit traverses the switch so as to travel ahead and set up control at the next router.

**ST stage:** In this stage, the flit traverses the crossbar. We use a custom-designed crossbar, as described in [19]. The data are read out of the input latch in front of the crossbar and in the worst case need to traverse the complete horizontal (X) and vertical (Y) dimensions of the crossbar channel (this is explained in Section 4). It is then latched into a latch prior to LT. The flit’s advanced bundle traverses the link in this cycle to reach the next node.

**LT stage:** The flit traverses the link in this cycle to reach the next router along its path. Simultaneously, the advanced bundle sets up the control for the switch allocator at the next



**Figure 1.** Baseline router pipeline [SS: Switch Setup, BW: Buffer Write, SA: Switch Allocation, VA: VC Allocation, ST: Switch Traversal, LT: Link Traversal]

router in preparation for the arriving flit.

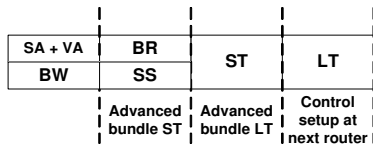
## 2.1 Optimized single-cycle pipeline

At low loads, when router output ports are mostly free, the advanced bundle is used to further optimize the router pipeline by cutting down the number of router pipestages to a single cycle, as opposed to the two-stage no-load bypass pipeline in the state-of-the-art baseline. Figure 3 shows an example of this scenario where a flit arriving from router 0 (where it goes through the normal pipeline) is able to bypass the pipeline at router 1 by directly traversing the crossbar switch upon arrival. This is accomplished by allowing the advanced bundle, which arrives one cycle earlier than the flit, to set up the control signals for the crossbar *a priori*. This, however, is allowed only if the following three conditions are satisfied: (a) there is no flit already in the buffer at the input port where the advanced bundle arrives, (b) there is no output port conflict with existing flits, i.e., there is no flit already in the SA stage of the router which is waiting to use the same output port as the one requested by the advanced bundle, and (c) there is no conflict with new flits, i.e., there is no output port conflict between multiple advanced bundle signals arriving in the same cycle, i.e., not more than one advanced bundle signal comes, simultaneously requesting the same output port. If any of the above conditions are not met, the flit has to go through SA as in the normal pipeline of Figure 2. However, in case of no conflicts with existing or new flits, a pipeline delay of just one cycle (corresponding to ST) through the router is realized.

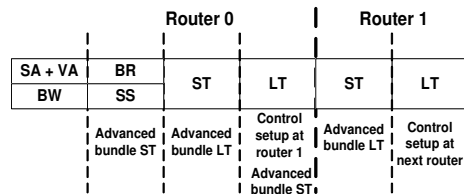
## 3 SPAROFLO switch allocation

SA refers to the mechanism of allocating output links to waiting input requests. It is a critical determinant of network throughput: an efficient allocator leads to better matching efficiency and hence avoids idle cycles on links for which traffic exists. In the context of on-chip networks, apart from creating a good match, SA needs to adhere to several unique design constraints. More specifically, tight area/power budgets translate to the use of single-ported input buffers and a single port from each input port into the crossbar (no speedup), thereby restricting the number of possible matches per cycle from each input port to one. Tight delay constraints imply single-cycle allocation, prohibiting the use of algorithms which require multiple iterations to find a better matching.

Past work on practical SA mechanisms has looked at separable allocators [6] which use two separate stages of arbitration: once across the input ports (local allocation) and the other across the output ports (global allocation). To improve



**Figure 2.** Proposed router pipeline



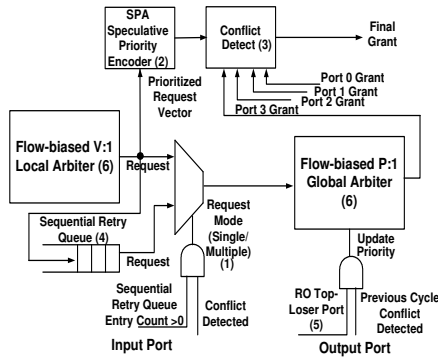
**Figure 3.** Optimized pipeline

the matching efficiency of separable allocation, parallel iterative matching (PIM) was proposed in [1], which uses multiple iterations of three basic phases: the first phase called the *request* phase involves each unmatched input port nominating a flit for each output port for which it has a request, followed by the *grant* phase where each unmatched output port randomly selects one input request from the ones it receives, and finally the *accept* phase where each unmatched input port selects one grant randomly from the ones it receives after the grant phase. PIM1, which uses a single-iteration of PIM, is more amenable to on-chip designs, as it avoids the delay incurred due to multiple iterations. To further reduce the allocation delay of PIM, Mukherjee et al. [13] proposed the simple pipelined arbitration algorithm (SPAA), where each input port nominates just one request in the request phase instead of nominating a request for each output port, thereby removing the possibility of conflicts, where multiple output ports grant the same input port, and hence obviating the need for the accept phase. However, this removes the possibility of conflicts at the expense of presenting a sparse request vector with fewer requests to the output arbiters as compared to PIM, which may result in fewer matchings under low/medium network load.

### 3.1 SPAROFLO

In this work, we present SPAROFLO, a novel SA mechanism which improves allocation efficiency in the context of on-chip designs. SPAROFLO incurs low critical path delay and can be implemented with simple hardware while using single-ported buffers and crossbars. The basic idea behind SPAROFLO is to use a separable allocation mechanism which adapts to switch contention by dynamically varying the number of requests presented by each input port to the global allocation phase. Moreover, SPAROFLO prioritizes past requests, which failed to use the switch because of allocation conflicts, over new requests. SPAROFLO also helps create a communication flow in the network by giving higher priority to successive requests from flits of the same packet which helps reduce the average resource occupancy per packet and hence improves throughput. Figure 4 presents the overall microarchitecture of SPAROFLO. The different features of SPAROFLO are implemented as a set of three priority rules which are explained in the following subsections.

**Speculative priority assignment (SPA):** Conventionally, each input port presents a single request per cycle (from the local arbiters) to the global arbiters, in *single-request per cycle mode*. One way to improve allocation efficiency at low/medium network load is to allow each input port to op-

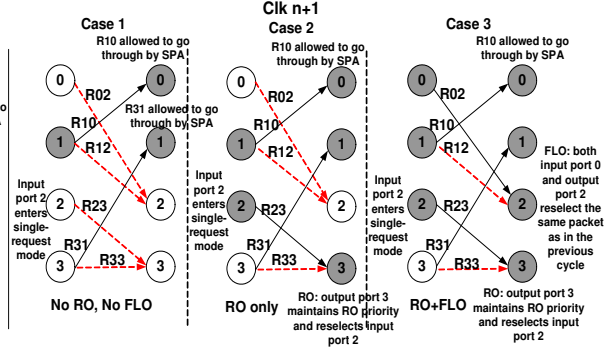
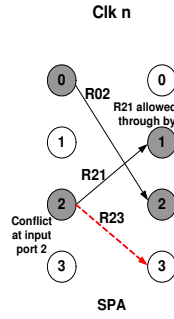


**Figure 4.** SPAROFLO switch allocation (V = number of VCs; P = number of ports)

erate in *multiple-request per cycle mode* instead, i.e., where each input port selects a dense request vector during local arbitration (one for each output port) to present to the global arbiters. This leads to a higher number of successful input-output matches. Correspondingly, there is a decrease in the number of times an output port goes idle even though there was a request for it. However, as network load increases, dense request vectors from each input can hurt throughput by leading to a higher number of input port conflicts, which happens when more than one output port selects the same input port. Hence, a mechanism to transition between these two modes, depending on contention for switch ports, can help adapt to traffic conditions. SPA builds on such an intuition by allowing an input port to operate in the multiple-request per cycle mode under normal conditions, but forcing it to transition to the single-request per cycle mode when it gets involved in a conflict (Module 1 in Figure 4).

Moreover, SPA mitigates the delay overhead of conflicts. Specifically, SPA speculates that a conflict will happen at an input port and generates a priority assignment for requests from each input port in parallel with global allocation (Module 2). In case when a conflict actually happens at an input port [detected by the conflict detect unit (Module 3)] and the number of requests from that input port is at most two, the highest-priority request determined by the generated priority assignment is chosen to go through. However, when the number of requests from a conflicting input port is greater than two, it is not possible to determine (in parallel with global allocation) the actual set of requests which are granted output ports and hence none of the requests is allowed to go through. Secondly, the conflicting requests which could not go through are put in a sequential retry/conflict queue (in the generated priority order) at the conflicted input port and re-tried one-by-one in subsequent cycles in the single-request per cycle mode (Module 4). This mode is maintained for as long as the sequential retry queue is non-empty, thereby favoring past conflicted requests over newer requests and hence reducing the chances of further conflicts from the same input port.

**Recreate old (RO):** The RO priority rule further complements SPA in clearing past conflicted requests before servicing new requests by biasing global allocation towards past requests which had conflicted. As mentioned earlier, in



**Figure 5.** SPAROFLO walkthrough

case of a conflict at an input port, the conflicted requests are put in the sequential retry queue in the priority order determined by SPA, and the local arbitration at that input enters the single-request mode, picking requests from the head of this retry queue every cycle. The request, which is initially at the head of this queue, is the top-loser from the last cycle or, in other words, the request which had the highest rank (in priority order determined by SPA) among the requests which could not go through the switch in the previous cycle because of the conflict. In the cycle immediately following the conflict, the conflicted input port, being in the single-request mode, is guaranteed to choose this top-loser request during local arbitration. RO guarantees that this request will also win global arbitration by simply recreating the past at the corresponding output port arbiter. This is done by maintaining the priority of the global matrix arbiter at the top-loser's output port (Module 5). Hence, RO further complements SPA in clearing past conflicting requests before servicing new ones.

**Flow (FLO):** Another intuition behind improving network throughput is to bias the allocation mechanism to create communication flows in the network by keeping flits of the same packet together as much as possible. This would result in packets being staggered across fewer routers and hence holding fewer network resources (like VCs) on average at any given instant of time. FLO is a priority rule which builds on this intuition by creating a hybrid of wormhole and virtual-cut through switching, i.e., allocating output ports in units of flits, but prioritizing flits belonging to the same packet during both local and global arbitration phases. In other words, both local and global arbiters favor flits belonging to the same packet which was selected in the previous cycle (Module 6). To avoid throughput degradation, this bias is maintained only if such contiguous requests from flits of the same packet exist. Moreover, this biasing does not suffer from any starvation occurrences, as the bias for flits from the same packet is broken once the tail flit of the packet wins allocation after which flits belonging to other packets can be served. Thus, flows are created across the network where flits of the same packet remain together for as long as possible and occupy fewer resources at a time. FLO also complements SPA in reducing the total number of input port conflicts since by reselecting flits from the same

packet, the conflict-free allocation from the previous cycle is maintained.

### 3.2 Walkthrough example

Figure 5 walks through how SPAROFLO works. Shaded circles represent successfully-matched ports, non-dashed lines represent successful requests while dashed lines represent unsuccessful requests. In the first phase of cycle  $n$ , local arbitration at input port 0 nominates a request R02 for output port 2, while input port 2 nominates requests R21 and R23 for output ports 1 and 3, respectively. In the global arbitration phase, requests R21, R02 and R23 are granted by output ports 1, 2 and 3 respectively. In parallel, SPA at input port 2 generates a priority order between requests R21 and R23, assuming a conflict will happen (R21 assumed to have a higher priority). As can be seen, a conflict at input port 2 happens which gets selected by more than one output port. Hence, the SPA priority is used to allow R21 to go through output port 1 while output port 3 goes unutilized. Figure 5 shows three different scenarios in the next cycle: the first one without RO or FLO, the second with RO only and the third with both RO and FLO. In all three scenarios, local arbitration at input port 0 selects R02 for output port 2, input port 1 selects requests R10 and R12 for output ports 0 and 2, respectively, while input port 3 selects requests R31 and R33 for output ports 1 and 3, respectively. Input port 2, on the other hand, enters the single-request mode due to a conflict in the previous cycle and nominates only one request R23 (the top-loser from the last cycle) for output port 3. In the first scenario, requests R10, R31, R12 and R33 get selected during global arbitration, leading to a conflict at input ports 1 and 3. Again, the priority order of SPA at each input port is used to allow one among the conflicting requests (R10 and R31 in this case) to go through output ports 0 and 1 while output ports 2 and 3 go unutilized. In the second scenario, the global arbiter at output port 3, which was involved in a conflict in the previous cycle, maintains its priority (only for this cycle) due to RO and reselects R23 from input port 2 (the top-loser from the last cycle). In the third scenario, assuming R02 is a request from a flit of the same packet as the one which was granted in the previous cycle, both the local arbiter at input port 0 and the global arbiter at output port 2 reselect R02. Hence, both RO and FLO lead to conflict-free successful matching of additional output ports.

## 4 Design and implementation

### 4.1 Buffer microarchitecture

In our design, we use dynamically-managed buffers, similar to the one proposed in [17], where the entire buffer pool is shared among all VCs, as opposed to assigning a fixed set of buffers to each VC which may lead to under-utilization for smaller-length packets. However, our buffer design targets wormhole flow control with flit-level buffering, as opposed to virtual cut-through flow control with packet-level buffering that is used in [17]. Our targeting of a very aggressive clock frequency also mandates careful design of the buffer microarchitecture. Figure 6 shows the

microarchitecture block diagram of the buffer data path and control structure. It consists of the following major blocks.

**Flit payload buffer:** This unit consists of buffers at each input port which are addressed in units of flits (128 bits each) and store all flits arriving at an input port before they are switched out of the router.

**Header control block (HCB):** In our dynamically-managed buffer design, since flits of a particular packet may be assigned non-contiguously within the payload buffer, the header control block contains an entry for each packet which serves to track the flits within that packet by storing the pointers where the respective flits are stored. In addition, the header control block also stores routing information related to a packet including its message (or traffic) class and output port. When a flit wins SA, its buffer pointer needs to be retrieved from the header control block in order to read the flit out of its buffer. In order to remove this extra indirection of accessing the header control block before BR, we use prefetching to read out the flit's buffer pointer at the time when the flit is placing a request for the switch port in anticipation that the flit will win SA.

**Buffer management unit:** This unit consists of the buffer allocator and the flow control manager. The buffer allocator is responsible for assigning a free buffer to each incoming flit. This is done by maintaining a list of free buffer entries and assigning the buffer at the top of this list to any incoming flit. The flow control manager is responsible for signaling buffer availability to the upstream router. We use on/off flow control [6] for this purpose. A free buffer count is maintained to count the total number of free buffers. This count is decremented when a new flit occupies a free buffer, and incremented when a flit leaves the router, with its buffer slot added to the free list pool. The thresholds for on/off signaling are calculated based on the round-trip delay between adjacent routers. In addition, one buffer is marked as reserved for each packet and is not returned to the free pool until the tail flit of that packet has been transmitted. This is done to avoid deadlocks and ensure forward progress of flits within a packet even if there are no buffers available in the free buffer pool.

**Packet forwarding unit:** This helps implement the FLO feature of our switch allocator (where successive requests from flits of the same packet are prioritized) by retrieving successive buffer pointers from the same entry of the header control block corresponding to flits within the same packet.

### 4.2 Switch allocator microarchitecture

Figure 7(a) shows the SPAROFLO switch allocator microarchitecture. We use a distributed allocator design where the same allocation logic is replicated at every input port. As described in Section 2, the request setup to the SA stage is removed from the critical path using the advanced bundle which arrives one cycle ahead of the data. Each flit on arrival places a request in a FIFO queue corresponding to its output port. Local arbitration at every input port simply involves picking the first entry out of this FIFO queue for each output port. This has a simpler hardware implementation

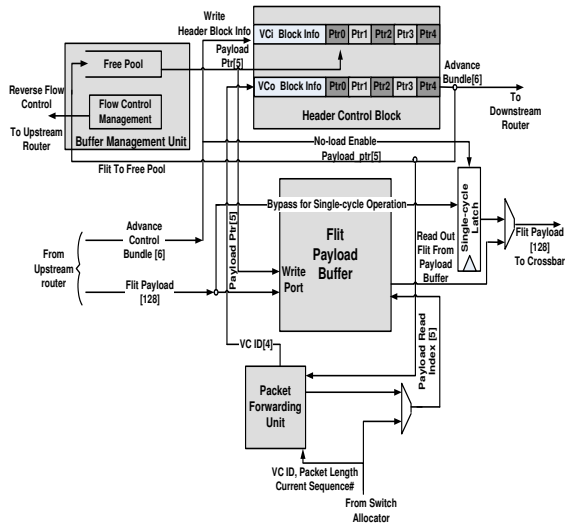
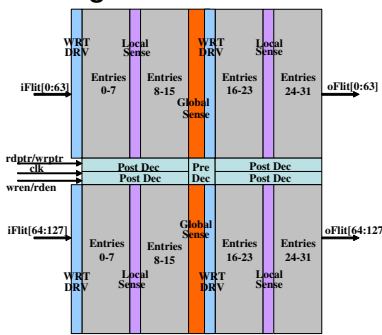
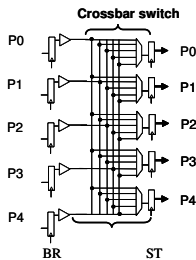


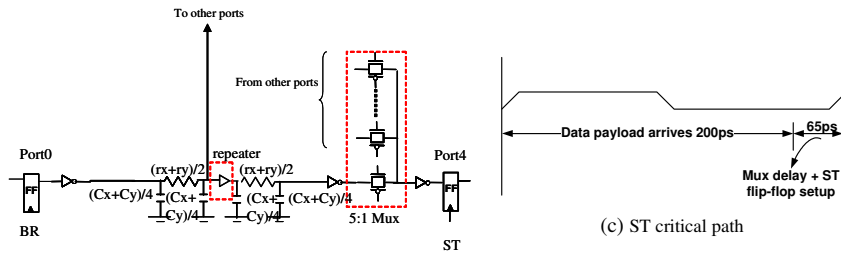
Figure 6. Buffer microarchitecture



(a) Input buffer layout



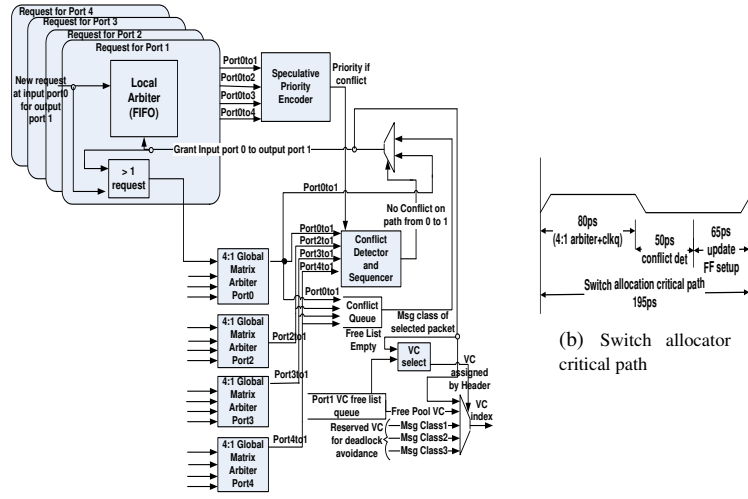
(a) ST logical path



(b) ST physical path

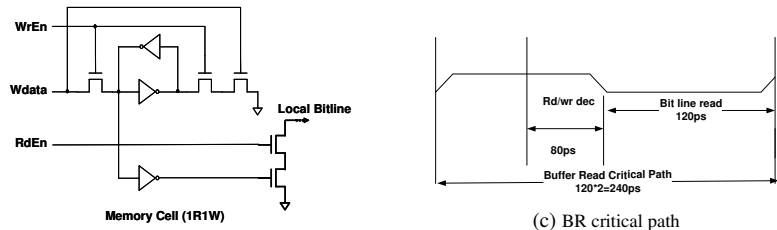
Figure 9. Switch traversal analysis

as compared to using a queuing arbiter which uses complex hardware with CAMs [6]. In order to start global arbitration in parallel at the beginning of the cycle itself, an indicator is used for each output port to track if there is at least one request in local arbitration. This indicator signal is valid at the beginning of the cycle and is used to trigger global arbitration in parallel. The global arbitration consists of a 4:1 matrix arbiter at each output port. Speculative priority encoding of requests occurs in parallel with global arbitration and hence does not fall on the critical path. In order to detect conflict conditions (when multiple output ports select the same input port), a conflict detector is used. In the absence of conflicts, the global arbitration output is deemed to be final and the local arbiter state is updated appropriately.

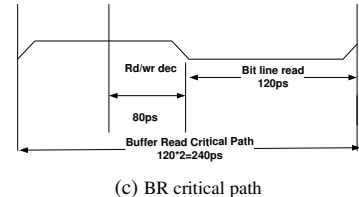


(a) Switch allocator microarchitecture

Figure 7. Switch allocator analysis

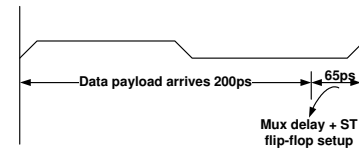


(b) Register file cell



(c) BR critical path

Figure 8. BR stage analysis



(c) ST critical path

However, in case of a conflict, the conflicting requests are placed in the conflict queue at the input port and serviced sequentially in subsequent cycles, with the local arbiter not generating additional requests during this time. VA is done by picking a free VC for each output port. The entire pool of VCs is partitioned into a reserved pool and an unreserved pool. The reserved pool has one VC for each protocol message class (MC) and helps avoid protocol-level deadlocks. The unreserved pool consists of the remaining bulk of VCs which can be allocated to packets belonging to any MC. During VA, first the unreserved pool is checked at each output port to pick a free VC. If no unreserved VC is available, a free VC is picked from the reserved pool corresponding to the flit's MC.

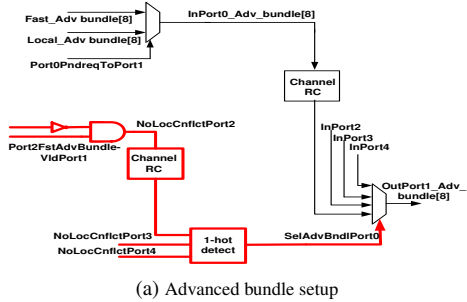


Figure 10. Advanced bundle setup and switch traversal

### 4.3 Critical path analysis

**SA stage:** Figure 7(b) shows the critical path in the SA stage. As can be seen, it consists of a 4:1 matrix arbiter followed by conflict detect and input state update. BW, which involves wordline decode and register cell write, takes place in parallel and does not affect the critical path delay. VA is done in parallel with SA. During VA, if a VC needs to be picked from the reserved pool (in case when there is no VC available in the unreserved pool), the selection can only happen after the MC of the flit, which won SA, is known. This selection is done in parallel with input state update and hence does not add to the critical path.

**BR stage:** Figure 8(a) shows the layout of input buffers which are implemented using register file cells [shown in Figure 8(b)]. The critical path in this stage [shown in Figure 8(c)] consists of read wordline decode in the first phase of the cycle followed by bitline read in the second phase.

**ST stage:** Figure 9(a) shows the logical data path for crossbar traversal while Figure 9(b) shows the corresponding wire RC delay. The worst-case delay corresponds to traversing the entire X and Y dimensions. The critical path is set by this wire RC delay, followed by the multiplexer and ST flip-flop setup time, shown in Figure 9(c).

**Advanced bundle setup and ST stage:** The critical path in this stage is set in the case when the advanced bundle enables the single-cycle pipeline discussed in Section 2.1. In order to do so, the advanced bundle needs to check conflict conditions mentioned in Section 2.1 as well as traverse the switch in the same cycle. To minimize the critical path, these two things are done in parallel. Conflict conditions, with pending flits already in the router, are checked first, followed by a 1-hot detect logic to check if exactly one advanced bundle arrives asking for a particular output port (shown in Figure 10(a)). If there are no conflicts, the advanced bundle crossbar setup multiplexer is enabled. The critical path is shown in Figure 10(b). The advanced bundle uses a narrow channel in the middle of the crossbar and hence incurs a shorter wire RC delay in both X and Y dimensions as compared to the flit payload ST RC delay. This slack is used to accommodate most of the additional logic for checking conflicts.

## 5 Evaluation

In this section, we present evaluation results of our proposed router design. Table 1 shows the network and process

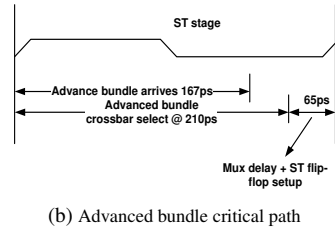


Figure 11. SA latency comparison

parameters along with the traffic pattern used. To model network performance, we used a cycle-accurate microarchitectural simulator which models all major components of the router pipeline at clock granularity, viz., buffer management, routing algorithm, VA and SA, and flow control between routers, and tracking the state of each flit every cycle as it travels through the router pipeline. Our circuit simulation uses a bit-slice approach for estimating area, power and timing. The estimated area of each sub-block is derived by laying out the bit slices. A floorplan is created for the router based on area estimates at the sub-block level. The global wire RC components are estimated from the top-level floorplan. A switch-level circuit simulator is used for power estimation at the sub-block level. Power estimation for each sub-block is done at nominal operating conditions (1.0V, 110°C) with 10% activity at each interface of the sub-block. For each sub-block, a transistor-level netlist is provided to the simulator. Power simulation is done using the pre-layout schematic netlist. The tool does RC estimation during power calculation. The router power is derived by adding up all sub-block level power components. Complete circuit simulations with the estimated RC is done in SPICE to get the delay of the critical paths.

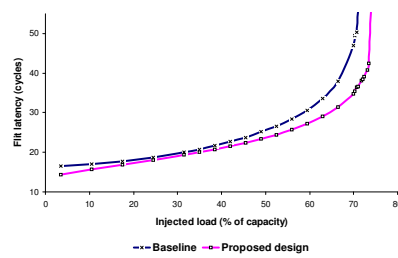
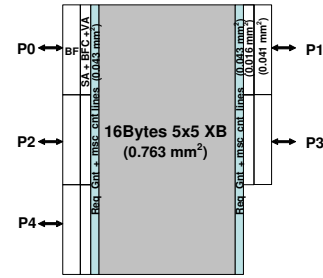
### 5.1 Network-level results

**SPAROFLO switch allocator performance:** To study the effectiveness of the proposed SPAROFLO SA mechanism, we compare it with both PIM1 and a separable allocator similar to SPAA. In order to isolate the impact of SA, in this study we consider a large number of VCs and buffers so that the performance is dominated by SA alone. Figure 11 plots flit latency as a function of network load for the different allocation schemes. It can be seen that while SPAA outperforms PIM1, SPAROFLO further improves this performance at all levels of network loads.

**Overall router performance:** Figure 12 plots flit latency of the proposed design as a function of network load assuming the parameters presented in Table 1, comparing it against the baseline design of Figure 1 which incorporates state-of-the-art techniques such as lookahead routing [7], speculation [15] and no-load pipeline bypassing. It can be seen that our design outperforms the baseline both in terms of latency and throughput with the latency reduction near saturation being 21% and that at no-load being 13%.

**Table 1.** Network parameters

|                         |                   |
|-------------------------|-------------------|
| Technology              | 65 nm             |
| $V_{dd}$                | 1.0 V             |
| Topology                | 6-ary 2-mesh      |
| Routing                 | Dimension-ordered |
| Traffic                 | Uniform random    |
| Number of router ports  | 5                 |
| Number of MCs           | 3                 |
| Reserved VCs per MC     | 1                 |
| Unreserved VCs          | 12                |
| Buffers per port        | 32                |
| Flit size/channel width | 128 bits          |

**Figure 12.** Network-level performance**Figure 13.** Router layout [BF: Buffer, BFC: Buffer control, VA: VC allocator, SA: Switch allocator]**Table 2.** Critical path delay of different router pipestages

| Router pipestage             | Critical path delay |
|------------------------------|---------------------|
| Advanced bundle setup and ST | 275ps               |
| SA                           | 195ps               |
| BR                           | 240ps               |
| Data ST                      | 265ps               |

**Table 3.** Router power breakdown  
Simulated power @(1.0 V, 110°C, 65 nm)

|              |        |
|--------------|--------|
| Crossbar     | 192 mW |
| Input buffer | 255 mW |
| Clock buffer | 90 mW  |
| Allocation   | 14 mW  |

## 5.2 Circuit simulation results

**Router layout:** Figure 13 shows the floorplan of the proposed router. As can be seen, the height (Y dimension) of the router is set by the buffer height whereas the length (X dimension) is governed by the crossbar dimension (wire pitch). The router area was estimated to be  $1.19 \text{ mm}^2$ .

**Timing:** Table 2 shows the critical path delay of each router pipestage. It can be seen that advanced bundle setup and data ST stages have comparable critical path delay, with the former seeing the longest critical path delay (275ps) which is used to set the router pipeline frequency of 3.6GHz.

**Power estimates:** Table 3 shows the power estimate (including both dynamic and leakage power) of each router component, assuming an activity factor of 10% at the interface. It can be seen that router power is mainly dominated by the crossbar and input buffers which constitute 34.8% and 46.2% of the overall power, respectively.

## 6 Conclusion

As technology scaling leads to an increasing number of computation cores on-chip, a high performance and energy-efficient on-chip network for communication among these cores is becoming increasingly critical. In this paper, we presented an in-depth design of an on-chip network router which targets a high clock frequency CMP system and achieves low latency and high throughput, while at the same time being area- and power-efficient, highlighting the design challenges and microarchitectural and circuit-level innovations used to overcome them.

## Acknowledgments

We would like to thank D. N. Jayasimha, Aniruddha S. Vaidya and David V. James of Intel Corp. for valuable discussions on the microarchitecture. This work was supported in part by the MARCO Gigascale Systems Research Center, an Intel PhD Fellowship and NSF under grant no. CNS-0613074.

## References

- [1] T. E. Anderson *et al.* High speed switch scheduling for local area networks. *ACM Trans. Computer Systems*, 11(4):319–352, Nov. 1993.
- [2] P. Bai *et al.* A 65nm logic technology featuring 35nm gate lengths, enhanced channel strain, 8 Cu interconnect layers, low-k ILD and 0.57 SRAM cell. In *Proc. Int. Electronic Devices Meeting*, pages 657–660, Dec. 2004.
- [3] L. Benini and G. D. Micheli. Networks on chips: A new SoC paradigm. *IEEE Computer*, 35(1):70–78, Jan. 2002.
- [4] S. Borkar. Performance, power and the platform. *Technology @ Intel Magazine*, Nov. 2005.
- [5] W. J. Dally and B. Towles. Route packets not wires: On-chip interconnection networks. In *Proc. Design Automation Conf.*, pages 684–689, June 2001.
- [6] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers, 2004.
- [7] M. Galles. Scalable pipelined interconnect for distributed endpoint routing: The SGI SPIDER chip. In *Proc. Hot Interconnects 4*, pages 141–146, Aug. 1996.
- [8] P. Gratz *et al.* Implementation and evaluation of a dynamically routed processor operand network. In *Proc. Int. Symp. Networks-on-Chip*, pages 7–17, May 2007.
- [9] K. Hughes *et al.* Physical simulation for animation and visual effects: Parallelization and characterization for chip multiprocessors. In *Proc. Int. Symp. Computer Architecture*, June 2007.
- [10] A. Jalabert *et al.* XpipesCompiler: A tool for instantiating application specific networks-on-chip. In *Proc. Design Automation and Test in Europe Conf.*, Feb. 2004.
- [11] N. E. Jerger, M. Lipasti, and L.-S. Peh. Circuit-switched coherence. *Computer Architecture Letters*, 6(1):5–8, Jan. 2007.
- [12] K. Lee, S.-J. Lee, and H. J. Yoo. Low power network-on-chip for high performance SoC design. *IEEE Trans. VLSI Systems*, 14(2):148–160, Feb. 2006.
- [13] S. S. Mukherjee *et al.* A comparative study of arbitration algorithms for the Alpha 21364 pipelined router. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, pages 223–234, Oct. 2002.
- [14] R. Mullins, A. West, and S. Moore. The design and implementation of a low-latency on-chip network. In *Proc. Asia and South Pacific Design Automation Conf.*, pages 164–169, Jan. 2006.
- [15] L.-S. Peh and W. J. Dally. A delay model and speculative architecture for pipelined routers. In *Proc. Int. Symp. High Performance Computer Architecture*, pages 255–266, Jan. 2001.
- [16] A. Pullini *et al.* NoC design and implementation in 65nm technology. In *Proc. Int. Symp. Networks-on-Chip*, May 2007.
- [17] Y. Tamir and G. L. Frazier. Dynamically-allocated multi-queue buffers for VLSI communication switches. *IEEE Trans. Computers*, 41(6):725–737, June 1992.
- [18] S. Tota, M. R. Casu, and L. Maccahiarulo. Implementation analysis of NoC: A MPSoC trace-driven approach. In *Proc. ACM Great Lakes Symp. on VLSI*, pages 204–209, 2006.
- [19] S. Vangal, N. Borkar, and A. Alvandpour. A six-port 57GB/s double-pumped nonblocking router core. In *Proc. Symp. VLSI Circuits*, pages 268–269, June 2005.
- [20] S. Vangal *et al.* An 80-tile 1.28TFLOPS network-on-chip in 65nm CMOS. In *Proc. Int. Solid-State Circuits Conf.*, Feb. 2007.