VLSI Micro-Architectures for High-Radix Crossbar Schedulers

Giorgos Passas Manolis Katevenis Dionisios Pnevmatikatos

Institute of Computer Science (ICS)
Foundation for Research and Technology – Hellas (FORTH)
Heraklion, Crete, Greece
{passas, kateveni, pnevmati}@ics.forth.gr

Abstract

We study the scaling of parallel-matching crossbar schedulers to radices above 100. First, we examine a traditional microarchitecture that implements the matching decision of each input and each output of the crossbar in a separate arbiter block and communicates the matching decisions between the input and the output arbiters through global point-to-point links. Using simple models and experimentation with 90nm CMOS layouts, we show that this architecture is expensive because the global point-to-point links take up $O(N^4)$ area, where N the radix of the crossbar. Next, by observing that the wiring of an arbiter fits in a minimal $O(NlogN)$ area, we propose a novel microarchitecture that inverts the locality of wires by orthogonally interleaving the input with the output arbiters, thus lowering the wiring area of the scheduler down to $O(N^2log^2N)$. Using this architecture, the scheduler for a radix-128 FIFO, VOQ, or 2-VC crossbar becomes gate limited, fitting in 3.6, 7.2, and 7.2mm^2 respectively, which is a 40, 50, and 70% improvement compared to the traditional. Moreover, the proposed schedulers find a new match in less than 10ns, thus allowing a minimum packet below 30Bytes at 24Gb/s line rate. Based on these findings, we conclude that crossbar schedulers are feasible even for radices above 100.

General Terms

Algorithms, Design

Keywords

Crossbar, Parallel Iterative Matching, ASICs

Giorgos Passas and Manolis Katevenis are also with the University of Crete, Computer Science Department (CSD). Dionisios Pnevmatikatos is also with the Technical University of Crete, Department of Electronic & Computer Engineering, Microprocessor & Hardware Lab.

NOCS'11 May 1-4, 2011 Pittsburgh, PA, USA

1. INTRODUCTION

Designers generally believe that many-core chips need a multi-hop Network on Chip (NoC) topology to interconnect a few tens of cores because a crossbar would be prohibitively expensive [4][13]. In contrast, we showed that the datapath of a crossbar interconnecting 128 cores in a single hop, with 24Gb/s (unidirectionally) per core, fits in just 7.6mm^2 of silicon in 90nm CMOS [12].

Our approach is also applicable in router chips for processor-memory or I/O interconnection networks and in packet-switching fabrics for the Internet. It shows that crossbar speedup is not overly expensive, even for radices above 100. Thus, combined input-output queueing (CIOQ) [12] is advantageous over crosspoint queueing (bufXbar) [8][9][11], as it significantly reduces the costly partitioning of buffer memories [12].

In this paper, we contribute the control part of the crossbar, focusing on the crossbar scheduler, which is the most intricate component thereof. We show that the scheduler of a radix-128 crossbar with single-FIFO, Virtual Output Queues (VOQs), or two Virtual Channels (VCs) per input fits in less than 8mm^2 of silicon. Moreover, the scheduler makes a new decision in less than 10ns, thus allowing a minimum packet as small as 30Bytes at 24Gb/s line rate.

Hence, combined with our previous work, we prove that a radix-128 6Tb/s *Crossbar Network on Chip* is feasible, fitting in less than 16mm^2 of silicon, even in a conservative 90nm process.

In this paper, we are concerned with the popular parallel-matching crossbar schedulers, such as the PIM [1] and the *i*SLIP [10] scheduler. Running these schedulers, the inputs and the outputs of the crossbar consent on a bipartite match by exchanging matching decisions, which they make locally and in parallel. Decisions are implemented using one arbiter circuit per input and one per output, and communication using point-to-point links between the input and the output arbiters [1][10].

A traditional microarchitecture [1][10][6] places each arbiter in a separate block, thus localizing the wires of the arbiters and globalizing the point-to-point links. We observe that the point-to-point links require $O(N^4)$ area, where N the crossbar radix. On the other hand, the area of the arbiters is much smaller: Their aggregate area is $O(N^2)$ for gates and $O(N^2logN)$ for wiring. Thus, the traditional mi-

croarchitecture has a handicap at high radices.

We fix this handicap by inverting the locality of wires: By *orthogonally interleaving* the input with the output arbiters, we localize the point-to-point links, while only globalizing the wiring inside the arbiters. Thus, we lower the wiring area of the scheduler down to $O(N^2 log^2 N)$.

We call the proposed microarchitecture *cross* and the traditional microarchitecture *block*.

We compare the above two microarchitectures on the *i*SLIP scheduler. Using simple models and experimentation with 90nm CMOS layouts, we show that for N above 64, a block *i*SLIP scheduler is limited by the area of the point-to-point links: For N=128, it takes up 14mm^2 –twice our crossbar datapath, 70% of which is due to the point-to-point links. On the other hand, a cross *i*SLIP scheduler remains gate limited even for radices above 128: For N=128, it fits in just 7mm^2, which is a 50% improvement.

Next, we adapt the two microarchitectures to schedulers for FIFO and VC crossbars. By extrapolation, we find that for N=128, a block FIFO or 2-VC scheduler is again wire limited, occupying 6 and 25mm^2 respectively. On the other hand, the cross schedulers fit in 3.5 and 7mm^2, which is a 40% and 70% improvement respectively.

Below, (1.1) we summarize the results of our previous work on the datapath of the crossbar, (1.2) elaborate on the position of the scheduler on the die, and (1.3) outline our paper.

1.1 Synopsis of Previous Work

In [12], we designed and laid out a radix-128 750MHz 32-bit bit-sliced crossbar in a 90nm CMOS process [5]. The bit slice was an array of 128 128:1 multiplexor trees, placed in an area of 154μm × 1.1mm = 0.169mm^2 (with 95% density), and routed in the lower four metal layers. The crossbar was two stacks of bit slices, placed in an area of (16×180μm) × (2×1.15mm) = 6.6mm^2, and routed in the upper five metal layers, on top of the bit slices.

We clocked the crossbar at 750MHz using three pipeline stages: *(i)* one from the inputs of the crossbar to the inputs of the bit slices, *(ii)* one inside the bit slices, and *(iii)* one from the outputs of the bit slices to the outputs of the crossbar.

We surrounded the crossbar with 128 "user IP" tiles, of size 1mm×1mm each, arranged in a 12×12 = 144 matrix, where the crossbar and its control replace the 16 = 4×4 center-most tiles. Assuming that three fourths of each user tile contain SRAM blocks (e.g. cache memories next to a processor or queues in a switch chip), we found out that all 128 input and 128 output links of the crossbar can easily be routed over the SRAM blocks, thus incurring virtually no area overhead to the user tiles for wiring. Finally, we clocked the IO links at the crossbar frequency using a two-stage pipeline. Including the repeaters and pipeline registers on the input and the output links, the total area of the crossbar datapath was 7.6mm^2.

1.2 Scheduler Floorplan

We allocate the remaining 9.4mm^2 (60%) of the central area to the crossbar scheduler. The scheduler can be connected to the user tiles through global links, running in parallel with the input and the output links of the crossbar. As we show in section 4.3, the links of the scheduler have much lower throughput, thus introducing only a small overhead, again fitting over the user tiles.

In the rest of this paper, we will be referring to the user tiles simply as crossbar inputs and outputs. We will be also considering that the inputs and the outputs have instant access to the crossbar –the latency on the global links can be pipelined.

1.3 Paper Outline

We start by reviewing the *i*SLIP scheduler in section 2. In section 3, we analyze the area cost of the block *i*SLIP microarchitecture, showing the wiring limitations at high radices. In section 4, we propose the cross *i*SLIP microarchitecture, and in section 5, we describe the CMOS layout of a radix-128 cross *i*SLIP scheduler, which is gate limited. We adapt the block and the cross microarchitectures to FIFO and VC schedulers in section 6. Finally, section 7 is a conclusion.

2. iSLIP REVIEW

We first review the *i*SLIP algorithm; next, we describe the *i*SLIP circuit.

2.1 The iSLIP Algorithm

The *i*SLIP algorithm [10] runs in the following four steps:

Step 1: Request. Each input sends a request to every output for which it has at least one queued packet.

Step 2: Grant. If an output receives any requests, it decides round-robin which one to grant and communicates back to each input whether or not its request was granted.

Step 3: Accept. If an input receives any grants, it decides round-robin which one to accept and communicates back to each output whether or not its grant was accepted.

Step 4: Slip. If an output is accepted by the input it granted (or equivalently, by any input), it increments (modulo N) its round-robin pointer to one location beyond that input; if an input accepts any output, it increments (modulo N) its round-robin pointer to one location beyond that output.

After the first three steps have been executed, a bipartite match between the inputs and the outputs has been found. The fourth step ensures that subsequent runs of the algorithm will give *fair* and often maximal matches. If the transmission time of a minimum packet is larger than the running time of the above steps, the first three ones may be iterated between the unmatched inputs and outputs to improve the size of the match.

2.2 The iSLIP Circuit

Figure 1(a) depicts a top-level view of the *i*SLIP circuit [6]. In the first stage, 1-bit point-to-point links communicate the requests from the inputs to the outputs; in the second stage, per-output round-robin arbiters decide which input to grant and 1-bit point-to-point links communicate the grants back to the inputs; in the third stage, per-input arbiters decide which grant to accept and 1-bit point-to-point links communicate the accepts back to the outputs; finally, the arbiters update their round-robin pointers accordingly. Thus,

the scheduler comprises $2N$ arbiters and $3N^2$ point-to-point links.

The arbiters can be instances of the same design. This design takes as input N request signals, finds the winning request in a round-robin schedule, and outputs the one-hot encoding of the winning request on N grant signals. Moreover, it decides whether to advance the round-robin pointer, or maintain its current value, by ORing together N *update* signals –for an output arbiter, these are the accepts it gets from the input arbiters, while for an input arbiter they are its own grant signals[1].

The arbiter uses two strict priority-encoder blocks, as in figure 1(b). A priority-encoder block takes as input N request signals and outputs: *(i)* N grant signals equal to the one-hot encoding of the first non-zero request in a strict-priority schedule, or zero when all inputs are zero; *(ii)* an *any* signal indicating whether there is at least one non-zero grant signal; and *(iii)* N pointer signals equal to the thermometer encoding of the winning request.

The arbiter prioritizes the grant and the pointer outputs of the lower over the upper block whenever the *any* signal of the lower block is non zero. In the next arbiter decision, the lower block sees the requests masked by the pointer. Thus, the arbiter keeps granting the requests round robin until the granted request is the one with the lowest priority. Then, the lower block sees no request active, and the arbiter selects the upper block. On the other hand, the upper block sees the requests as they are, thus the round-robin pointer wraps around (modulo operation).

The above design improves the vintage circular-token circuits, which are non-synthesizable. Observe that it still contains a cycle, which is broken by the register though.

Finally, the output (input) arbiters encode the id of the winning input (output) on $1+logN$ signals, which connect to the homonym crossbar output (input). The encoders introduce small area overhead –actually, they can be removed from the critical path by registering their inputs, thus we will be ignoring them until section 5.

3. iSLIP AREA COST ANALYSIS

Figure 1(a) describes the traditional architecture for parallel matching schedulers. For example, Anderson et al. [1] implemented PIM using a separate FPGA for each arbiter and running the point-to-point links off chip. On chip, the FPGAs become blocks, while the off-chip links become global links.

Using this architecture, the total area of the scheduler is $2N\times$ the area of the arbiters, plus the area of the point-to-point links. Below, (3.1) we describe simple models for the area of the arbiters and (3.2) the point-to-point links, and (3.3) we validate our models by experimenting with CMOS layouts.

3.1 Arbiter Area

An arbiter (see figure 1(b)) uses N two-input AND gates for the pointer mask, $3N$ two-input AND/OR gates for each of the two multiplexors, N two-input OR gates for the OR of updates, plus the gates of the two priority-encoder blocks. The simplest implementation of a priority-encoder block uses the linear circuit of figure 2(a) –a chain of ORs computes the thermometer encoding of the winning request, followed by

[1]Or equivalently, its own request signals.

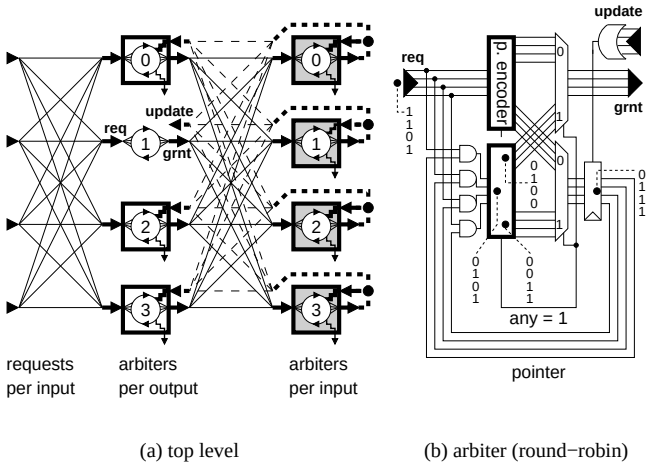


Figure 1: The *i*SLIP circuit, illustrated for an example N = 4. In (a), each downward-pointing arrow encodes the winning input (output) and connects to the corresponding output (input); the encoder is omitted in (b) for brevity.

an AND operation between the (inverted) thermometer code and the requests, which gives the grants; this accounts for $2N$ two-input AND/OR gates. Summing up, the arbiter uses a total of $12N$ two-input gates.

The corresponding area can be estimated by multiplying the gate count with the average area of a two-input gate. Adding the area of the N pointer flip flops, we get a model of the total gate area of the arbiter. We give numbers in section 3.3.

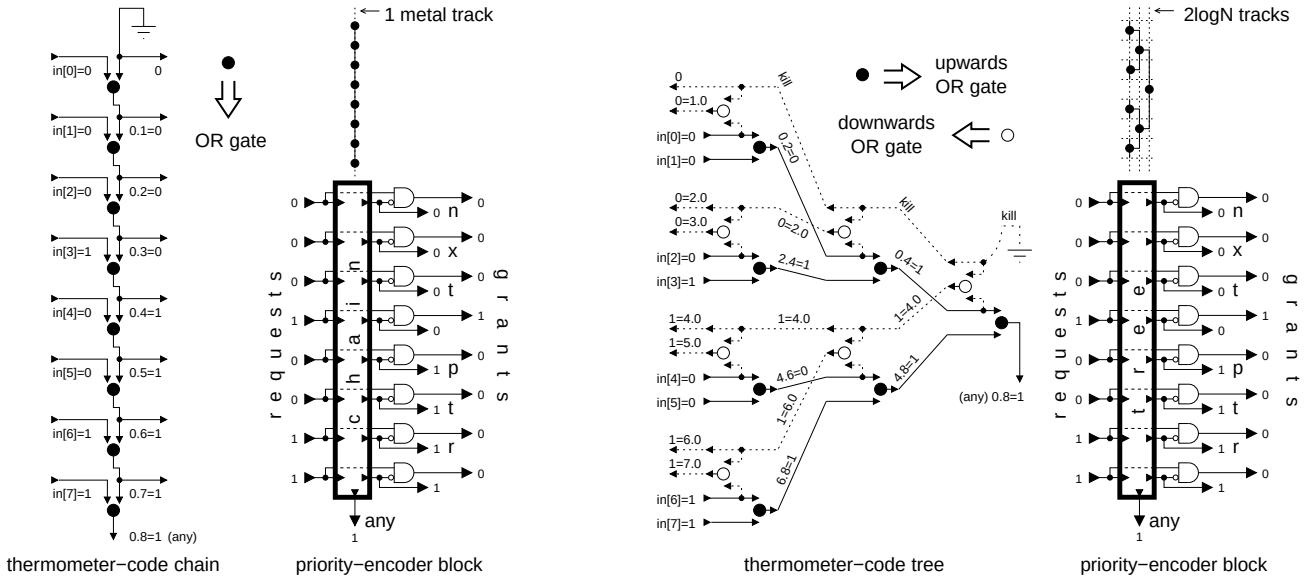
On the other hand, the wiring of the arbiter is due to few linear structures –for symmetry, we assume that the OR of updates is implemented using a chain of two-input ORs. Thus, the corresponding metal-track area is $k(NR/L\times R/L)$, where k the small number of linear structures, R the average routing pitch, and L the number of metal layers per dimension.

We call the above an *area-optimized* arbiter. Though too slow, it gives a lower bound on both gate and wiring area.

To speed up the design, *(i)* we replace the chain in each priority-encoder block with a tree structure, as in figure 2(b) –an upwards tree of OR gates computes parts of the thermometer code, followed by a downwards tree that combines them, thus reducing delay to $2logN$ stages; *(ii)* we implement the OR of updates using a tree; *(iii)* we amplify the fanout of the *any* signal using two trees of N buffers each (the *any* signal fans out to $2N$ 2:1 multiplexors); *(iv)* we amplify the fanout of the OR of updates using one tree of N buffers (the OR of updates fans out to N flip flops).

Thus, a *speed-optimized* arbiter uses a total of $14N$ two-input gates, $3N$ buffers, and N flip-flops. Moreover, it uses eight trees of N nodes, thus requiring a metal-track area of $8(NR/L\times(logN)R/L)$.

Finally, the tree of figure 2(b) was proposed by Brent and Kung [2]. Faster structures exist, but in our case, they offer only a minimal speedup, at significant area cost [7]. Besides, we observe that the delay overhead of the Brent-Kung structure is due to the delay of the upwards tree only –the traversal of the downwards tree can be overlapped with the traversal of the buffer trees of the *any* signal.



(a) (b)

Figure 2: The priority-encoder block of a round-robin arbiter, optimized for (a) area or (b) speed, illustrated for an example N = 8; The symbol $i.j$ denotes the prefix $in[i]\ OR\ in[i+1]\ \ldots\ OR\ in[j-2]\ OR\ in[j-1]$, where $i \in [0 : N-1]$ and $j \in [1 : N-1]$.

3.2 Point-to-Point Link Area

In ASICs, diagonal connections are implemented using "Manhattan routing", thus each point-to-point link uses one metal track per dimension. Hence, a model of the total area of the point-to-point links is $(3N^2R/L)^2$. Taking into account that metal tracks are half utilized on average, a more realistic model is $(1.5N^2R/L)^2$.

The basic architecture of figure 1(a) leaves the metal tracks on top of the arbiters unexploited. This is a drawback, especially when the arbiters utilize a small portion of that area and when the total area of the point-to-point links is not much larger than the total area of the arbiters.

However, we can exploit the metal tracks on top of the arbiters by arranging the arbiters in two dimensions. Figure 3(a) shows an example. Then, as long as the point-to-point links fit on top of the arbiters, their area overhead is virtually zero. Otherwise, one has to provide additional metal tracks by spreading the arbiters. Notice that wiring area remains $(1.5N^2R/L)^2$, as each link still requires half a metal track per dimension.

Finally, in figure 3(a), there are $1.5N^2/\sqrt{2N}$ wires on top of each arbiter per dimension. Because each arbiter has $3N$ IO pins, there are $3N$ *via* cuts in the proximity of each arbiter, connecting a part of the above wires down to the gates of the arbiter. Vias may congest routes on intervening layers, but because they are much less than the metal tracks[2], their overhead is also small. (An orthogonal observation is that routing point-to-point links in this floorplan is similar to routing router IO links in multi-dimensional topologies.)

3.3 Experimentation

We experimented with layouts of the above block architecture using a 90nm CMOS process with 9 layers of interconnect [5], and some popular EDA tools for simulation, synthesis, placement, and routing.

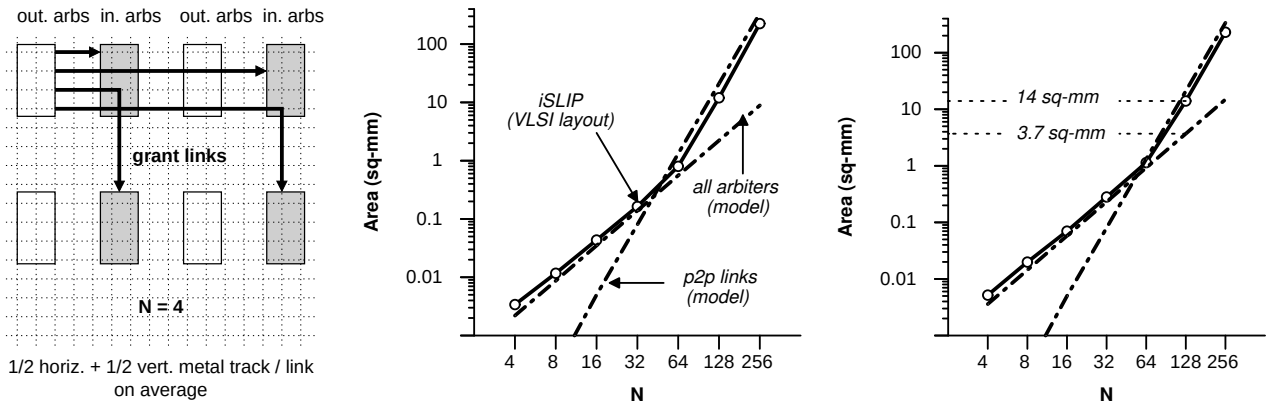
First, we described, synthesized, placed, optimized, and routed an area- and a speed-optimized arbiter for N in [4:256]. We empirically found that, for all N, the lower three metal layers are sufficient for the layouts to be routable with near 100% area utilization and no speed degradation. Next, for each N, we described an area- and a speed-optimized *i*SLIP scheduler using the corresponding arbiter layout as a *hard block*. We placed the arbiter blocks using the floorplan of figure 3(a). Finally, through experimentation loops, we specified their minimum spacing for the layouts to be routable.

Figure 3(b) plots the area of the area-optimized layouts; also plotted is: *(i)* the modeled gate area of the scheduler, i.e. the modeled gate area of one arbiter (section 3.1) multiplied by $2N$ –we use $4\mu m^2$ per two-input gate and $20\mu m^2$ per flip flop (with load enable and clear); and *(ii)* the modeled area of the point-to-point links (section 3.2) –we use $R = 0.56\mu$m and $L = 3$. On the other hand, the wiring area of the arbiter is much smaller than the gate area thereof, thus we omit it without affecting our results. We observe that the scheduler is gate limited for radices below 64 and wire limited for higher radices[3].

Figure 3(c) plots the area of the speed-optimized layouts. Compared to the area-optimized layouts, gate area doubles, mainly due to the stronger gates used. On the other hand, the area of the point-to-point links is almost the same. Thus, for $N = 128$, the gate area of *i*SLIP is $3.7mm^2$ and the total area $14mm^2$. Notice that the overhead of the point-to-point links is above 70%.

[2]The area of a via is few square routing pitches.

[3]A minimal spacing was needed between the arbiter blocks to avoid design rule violations. This explains the small discrepancy between the *i*SLIP area and the total arbiter area for small Ns.



(a) floorplan & global routing (b) area-optimized layouts (c) speed-optimized layouts

Figure 3: *i*SLIP area as a function of N; iSLIP is wire limited for N above 64.

4. CROSS MICROARCHITECTURE

In this section, we describe the proposed cross microarchitecture. First (4.1, 4.2), we slice and interleave orthogonally the input with the output arbiters, thus removing the overhead of both the grant and the accept links. Then (4.3, 4.4), we show how to remove the overhead of the request links by maintaining state at the crosspoints of the arbiters.

4.1 Arbiter Slicing

The first step is the slicing of the arbiter. An area-optimized arbiter can be sliced into N bits as in figure 4; each bit implements: *(i)* one of the request inputs, *(ii)* one of the update inputs, and *(iii)* one of the grant outputs; and contains the corresponding: *(i)* one of the pointer-mask gates, *(ii)* one of the pointer flip flops, *(iii)* one of the bits of each of the two priority-encoder blocks, *(iv)* one of the bits of each of the two multiplexors, and *(v)* one of the bits of the OR of updates. Thus, five metal tracks are needed vertically (independent of N): One track for each of the three OR chains, plus one track for each of the load enable and the *any* signals.

The slicing of a speed-optimized arbiter is analogous. However, the chain inside each of the priority encoders is replaced by a Brent-Kung tree. Thus, each bit of the priority encoders contains one of the nodes of the Brent-Kung tree, i.e. two instead of one OR gate. Moreover, the OR of updates is a tree instead of a chain. Finally, each arbiter bit additionally contains three buffers, implementing one of the nodes of each of the three fanout-amplification trees (one tree for the load-enable signal, plus two trees for the *any* signal). Thus, $8logN$ metal tracks are needed vertically, to route eight trees[4].

Because the area-optimized design is impractical, in the rest of this paper, we will be considering only the speed-optimized design, for brevity.

4.2 Crossed Arbiters

Now, we can invert the locality of wires by orthogonally interleaving the input with the output arbiters, as shown in figure 5.

[4]We consider all trees are binary, thus each tree contains $N-1$ nodes, while the arbiter contains N bits. This asymmetry can be resolved by using N nodes per tree, where one node is spare.

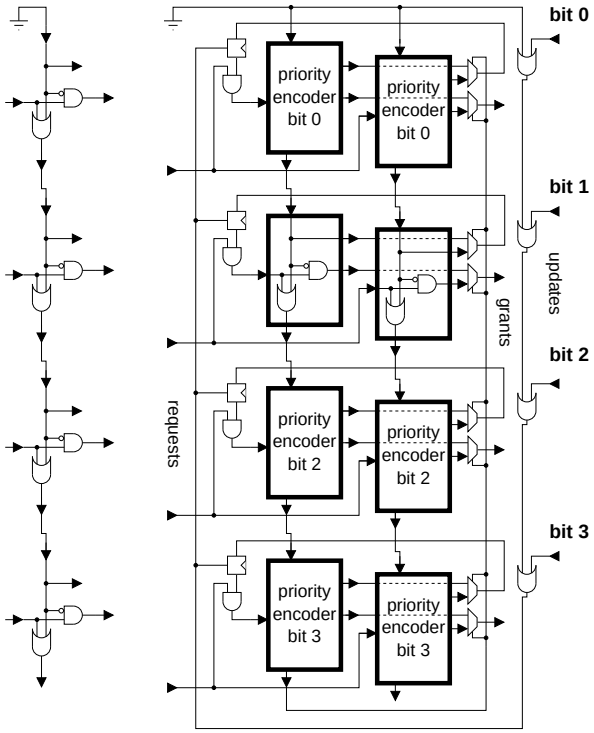


Figure 4: Arbiter slicing, illustrated for an example N = 4.

Let us index the arbiters and the bits of an arbiter with the integers i, j; i, j in [0:N-1].

As shown in figure 5(a), the grant output of bit i of output arbiter j connects via a grant link to the request input of bit j of input arbiter i, for all i, j in [0:N-1]. Moreover, the grant output of bit i of input arbiter j connects via an accept link to the update input of bit j of output arbiter i, for all i, j in [0:N-1].

As shown in figure 5(b), by orthogonally interleaving the input with the output arbiters, we manage to place the bit i of output (input) arbiter j in the proximity of the bit j of input (output) arbiter i, for all i, j in [0:N-1]. Notice that

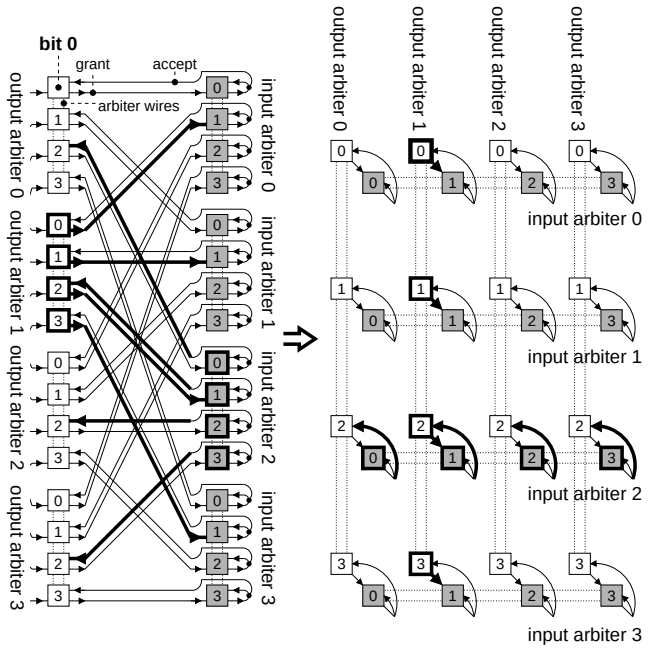


Figure 5: Orthogonal interleaving of the input with the output arbiters to localize the grant and accept links, illustrated for an example N = 4.

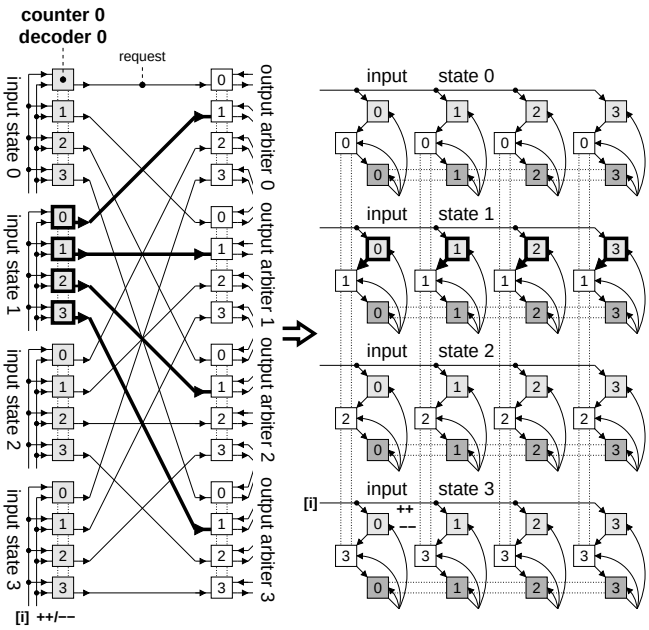


Figure 6: Orthogonal interleaving of the input state with the output arbiters to localize the request links, illustrated for an example N = 4.

the gates of the input and the output arbiters are arranged as *obeli*, their wires *cross*.

In this way, we need to provide metal tracks only for the arbiter wires. Since, there are eight global trees per output arbiter vertically and eight global trees per input arbiter horizontally, the corresponding metal-track area is $N(8logN)R/L \times N(8logN)R/L$.

Finally, crossed arbiters were also used in the *Wavefront Scheduler* [14]. However, in this scheduler, which pair of input and output arbiter bits has the top priority is determined by a single, global pointer, and priority shifts diagonally, in a wavefront. In contrast, *i*SLIP uses one separate pointer per arbiter, thus priority shifts in two phases: First vertically, in parallel across the output arbiters, and then horizontally, in parallel across the input arbiters. The stronger fairness properties of *i*SLIP should be attributed to this feature.

4.3 Encoded Requests

Extending N request links from the scheduler to each input (see sections 1.1, 1.2) introduces too much wiring overhead. Instead, we consider that each input sends only $1+logN$ request signals to the scheduler, encoding the destination of each newly injected packet.

For each input i, the scheduler maintains one counter for each output j, counting the packets queued for output j at input i. Additionally, for each input i, it maintains a decoder j next to the counter j, incrementing the counter whenever the newly injected packet at input i is destined to output j. Similarly, the decoder decodes the grant output of input arbiter i, and decrements the counter whenever this output is j. Thus, the counters have a consistent view of backlogs at the input queues.

Note that the aggregate throughput of the encoded request links is $O(NlogN)$, as $O(N)$ packets arrive at the network in this interval. On the other hand, the aggregate throughput of the point-to-point request links is $O(N^2)$, as all request links may be hot during periods of traffic stress.

Finally, instead of counting whole queue backlogs, the counters suffices to cover the round-trip time between the inputs and the scheduler, in analogy to *flow-controlled buffers* –in this paper, we consider 3-bit counters.

4.4 Crosspoint State

We call a *state bit* the decoder and counter pair corresponding to an input-output pair.

We can shorten the request links by orthogonally interleaving the state bits with the output arbiters, as shown in figure 6. Notice that the decrement of the counters is enabled by the point-to-point accept links, which are already local.

In conclusion, the requests add only a small overhead of $N(1+logN)$ horizontal wires, in total. Adding this overhead to the wiring area of the arbiters, the total wiring area of the scheduler is $N(9logN)R/L \times N(8logN)R/L$.

5. CMOS LAYOUT

We laid out a radix-128 cross *i*SLIP scheduler using some popular EDA tools and a 90nm CMOS technology with 9 layers of interconnect [5].

First, we synthesized one arbiter and one state bit. Then, by replicating and interconnecting *bit cells*, we synthesized the scheduler. Next, we placed the scheduler by generating the placement of one bit cell and shifting the coordinates of the rest[5]. Figure 7 shows the floorplan of the scheduler. For

[5]Notably, implementing the bit cells as hard blocks was inefficient because the blocks are rather small, thus introducing significant peripheral overhead.

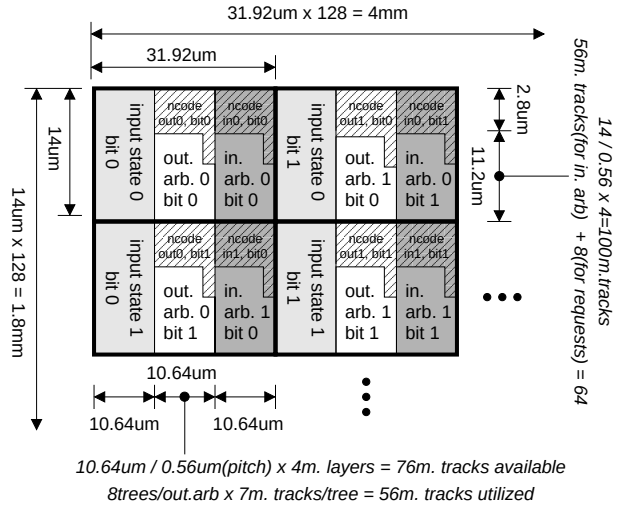


Figure 7: Floorplan of a radix-128 cross *i*SLIP scheduler in 90nm CMOS.

completeness, we also designed one encoder for each arbiter as seven trees of OR gates, distributed at the grant outputs of the arbiter.

Total area is 7.2mm^2 and area utilization nears 100%. This suggests a savings of 50% compared to the block architecture (see figure 3(c)). The area breakdown is as follows: 52% is due to arbiters (30% priority encoders, 22% multiplexors, 21% pointer flip flops, 12% buffers, and 15% others), 34% due to state keeping (50% counters and 50% decoders), and 14% due to encoders –see table 1.

In section 3, for simplicity, we assumed the block architecture is stateless. While the block architecture needs state keeping as well, at high radices, where wiring dominates, the state bits can be placed at the empty space "below the wires". In other words, in the above comparison, there is no state overhead in the block architecture. Thus, subtracting the state overhead from the cross architecture, the gate area of the cross and the block architecture are equal –both architectures use the same netlist, the cross architecture changes only the topology of the layout.

Moreover, in figure 7, observe that there are plenty of metal tracks on top of the standard cells, both vertically and horizontally. Thus, the layout is routable. Indeed, we were able to route it automatically with zero design-rule violations.

The critical path is 8ns long, lying in the "three-way handshake" between the leftmost output arbiter and the bottom input arbiter. First, there is a 2.8ns delay at the output arbiter, for the traversal of two Brent-Kung trees; second, there is a symmetrical delay of 3.2ns at the input arbiter; third, there is a 1.7ns delay at the output arbiter, for the OR of updates and for the traversal of the buffer tree. Thus, the scheduler is fast enough to allow a decent 30Byte minimum packet at 24Gb/s line rate.

Finally, the cross layout remains gate limited even for radices above 128. For example, one can scale the layout to a radix of 256 by replicating one bit cell 256 times horizontally $\times$ vertically. The wiring over each bit cell grows logarithmically, thus the metal tracks needed are 64 vertically and 73 horizontally, and they are already available. Thus, area is defined by gates, and it is 29mm^2. This is more than 7$\times$ smaller compared to the block architecture (see figure 3(c)).

Module	mm^2	% Total	Sub-mod.	mm^2	% Mod.
Arbiters	3.7	52	pencdrs	1.1	30
			muxes	0.8	22
			ptr FFs	0.8	21
			buffers	0.4	12
			others	0.6	15
State	2.4	34	cntrs	1.2	50
			decdrs	1.2	50
Encoders	1.0	14	ORs	1.0	100

Table 1: Area breakdown of a radix-128 cross *i*SLIP layout in 90nm CMOS. Total area is 7.2mm^2.

6. FIFO & VC SCHEDULERS

In this section, we adapt the block and the cross microarchitecture to schedulers for FIFO and VC crossbars, and we give area estimates.

6.1 FIFO Scheduler

When there is a single FIFO per input, each input may request at most one output at a time, i.e. the destination output of its head packet. As a consequence, every input receives at most one grant at a time from the outputs. In other words, when an input receives any grant, it may start forwarding to the destination output of its head packet.

Thus, a FIFO scheduler (architecture and layout) emerges from *i*SLIP by replacing each input arbiter with an N-input OR. Moreover, the accept point-to-point links, as well as the counters of the state bits, are needless.

Following the above discussion, the area of a block FIFO scheduler can be extrapolated from a block *i*SLIP scheduler (figure 3(c)) by halving the gate area (the overhead of the extra OR gates is negligible) and by multiplying the wiring area by $(2/3)^2$ (a FIFO scheduler has 2/3 the links of *i*SLIP). Then, for $N = 128$, a block FIFO scheduler is wire limited and occupies 6.2mm^2.

On the other hand, the cross architecture interleaves orthogonally the output arbiters with the input ORs and with the state bits. Thus, a cross FIFO scheduler occupies half the area of a cross *i*SLIP scheduler: For $N = 128$, a cross FIFO scheduler is gate limited and fits in 3.6mm^2, which suggests a 40% improvement compared to the block architecture.

Finally, the critical path of a cross FIFO scheduler contains the delay at an output arbiter, plus the delay of a distributed OR gate. For $N = 128$, we estimate these delays to 2.8ns and 0.9ns respectively. Thus, the delay of a radix-128 cross FIFO scheduler should be less than $4ns$.

6.2 VC Scheduler

Dally [3] proposed a basic design for VC schedulers. This design comprises a separate *VC allocator*, allocating (output) VCs to packets, and a *switch allocator*, allocating the crossbar to the flits of the packets.

In this paper, we consider packets do not change VCs. This is a basic configuration, still providing a mechanism to resolve protocol deadlock by separating the request from the

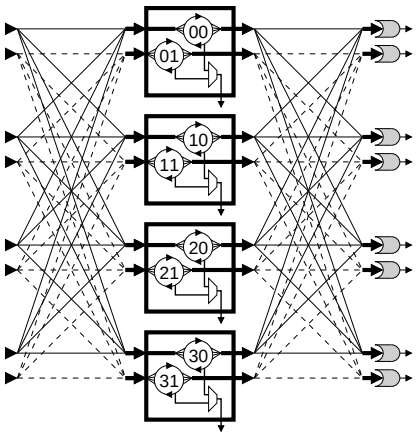


Figure 8: 2-VC crossbar scheduler, illustrated for an example N = 4. Not shown is a second-level arbiter per output and per input, prioritizing the VCs.

reply messages. Moreover, we consider VCs are scheduled at the flit level, thus we merge the VC and the switch allocator into one VC scheduler.

In particular, we consider the VC scheduler of figure 8. In this scheduler, each input may request up to V outputs, where V the number of VCs. Thus, each input has one request link per output per VC. In turn, each output has one arbiter (identical to an *i*SLIP arbiter) per VC and a second-level arbiter (not shown in the figure) to prioritize the VCs. Each input ORs together the output grants per VC, and a second-level arbiter (not shown in the figure) prioritizes the VCs.

To ensure fairness, this scheduler needs feedback from the second-level input arbiters to the output arbiters, similarly to *i*SLIP. However, we omit it here for simplicity. Moreover, when the number of VCs is small, the extra multiplexors and second-level arbiters introduce small area overhead.

Thus, we have a direct comparison between a VC and a FIFO scheduler. In particular, a block VC scheduler uses $V\times$ the gate area and $V^2\times$ the wiring area of a block FIFO scheduler. Hence, for N=128 and V=2, we estimate the area of a block VC scheduler to 25mm^2.

On the other hand, a cross VC scheduler emerges by replicating each bit of a cross FIFO scheduler V times. Thus, for N=128 and V=2, a cross VC scheduler takes up 7.2mm^2, which is a 70% improvement compared to the block architecture.

Finally, compared to a cross FIFO scheduler, delay increases only by the delay of the second-level arbiters. For small V, we can ignore this overhead. Then, for N=128 and V=2, delay should be well below 10ns.

7. CONCLUSION

High gate densities, abundant wiring resources, and studious layout techniques all make it possible to build high-radix crossbar NoC, using less than 16mm^2 of silicon even in 90nm CMOS. Before replacing the moderate-scale multi-hop NoC topologies with crossbar NoC, there are several more issues to consider, such as energy efficiency and fault tolerance. However, the realization of how small a crossbar NoC is, is the first and perhaps the most important step in this direction.

Acknowledgments

This work was supported by the European Commission in the context of the HiPEAC Network of Excellence (FP7 #217068). We thank the reviewers for their helpful comments.

References

[1] T. Anderson, S. Owicki, J. Saxe, and C. Thacker. High-speed switch scheduling for local-area networks. *ACM Transaction on Computer Systems*, 11, 1993.

[2] R. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, C-31, 1982.

[3] W. Dally. Virtual-channel flow control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2), 1992.

[4] W. Dally and B. Towles. Route packets, not wires: on-chip inteconnection networks. In *Proceedings of the 38th annual ACM Design Automation Conference (DAC)*, June 2001.

[5] FARADAY Technology Corporation. UMC's 90nm Logic SP-RVT Standard Cell Process. www.faraday-tech.com.

[6] P. Gupta and N. McKeown. Designing and implementing a fast crossbar scheduler. *IEEE Micro*, 19, 1999.

[7] D. Harris. A taxonomy of parallel prefix networks. In *Proceedings of the Asilomar Conference on Signals, Systems, and Computers*, November 2003.

[8] M. Katevenis, G. Passas, D. Simos, I. Papaefstathiou, and N. Chrysos. Variable packet size buffered crossbar (CICQ) switches. In *Proceedings of the IEEE Int. Conference on Communications (ICC)*, June 2004.

[9] J. Kim, W. Dally, B. Towles, and A. Gupta. Microarchitecture of a high-radix router. In *Proceedings of the 32nd annual ACM/IEEE Int. Symposium on Computer Architecture (ISCA)*, June 2005.

[10] N. McKeown. The iSLIP scheduling algorithm for input-queued switches. *IEEE/ACM Transactions on Networking*, 7, 1999.

[11] G. Mora, J. Flich, J. Duato, P. López, E. Baydal, and O. Lysne. Towards an efficient switch architecture for high-radix switches. In *Proceedings of the 2006 ACM/IEEE Int. Symposium on Architectures for Networking and Communications Systems (ANCS)*, December 2006.

[12] G. Passas, M. Katevenis, and D. Pnevmatikatos. A 128x128x24Gb/s crossbar, interconnecting 128 tiles in a single hop, and occupying 6% of their area. In *Proceedings of the 4th ACM/IEEE Int. Symposium on Networks-on-Chip (NOCS)*, May 2010.

[13] A. Pullini, F. Angiolini, S. Murali, D. Atienza, G. De Micheli, and L. Benini. Bringing NoCs to 65 nm. *IEEE Micro*, 27, September 2007.

[14] Y. Tamir and H. Chi. Symmetric crossbar arbiters for VLSI communication switches. *IEEE Transactions on Parallel & Distributed Systems*, 4, 1993.