Æthereal Network on Chip: Concepts, Architectures, and Implementations

Kees Goossens, John Dielissen, and Andrei Rădulescu
Philips Research Laboratories

Editor's note:
Many SoC applications require guaranteed levels of service and performance. Can networks on chips (NoCs) enable such guarantees? Here, the authors demonstrate that the Æthereal network can. This particular NoC, developed at Philips Research Laboratories, encompasses hardware, a programming model, and a design flow. Read on to find out about the details.
—André Ivanov, University of British Columbia

CONTINUING ADVANCES in semiconductor technology enable the integration of increasing numbers of IP blocks in a single SoC. Interconnect infrastructures, such as buses, switches, and networks on chips (NoCs), combine the IPs into a working SoC. Moreover, the industry expects platform-based SoC design to evolve to communication-centric design, with NoCs as a central enabling technology.[1]

In this article, we introduce the Æthereal NoC.[2-4] The tenet of the Æthereal NoC is that guaranteed services (GSs)—such as uncorrupted, lossless, ordered data delivery; guaranteed throughput; and bounded latency—are essential for the efficient construction of robust SoCs. One reason is that many IPs have inherent performance requirements, such as a minimum throughput (for real-time streaming data) or bounded latency (for interrupts). Furthermore, because the traffic of different IPs does not interfere with each other, the IPs' behaviors are decoupled; thus, the IPs can be designed and tested independently of each other and the NoC. This aids in the compositional design and programming of SoCs.

GSs require resource reservations for the worst case. To exploit the NoC capacity unused by GS traffic, we also provide best-effort services (BESs). GSs serve critical (for example, real-time) traffic, and BESs serve noncritical communication.

Many architectures that implement BESs already exist, but our concept of contention-free routing is one of the first to offer guaranteed services—throughput and latency, in particular—in addition to BESs. GSs require resource reservation; the Æthereal NoC thus requires configuration and programming. We offer alternative programming models and router architectures to facilitate design space exploration: A system architect can optimize a NoC with either a distributed programming model (for scalability) or a centralized programming model (for low cost). In the latter case, he can choose between a NoC without BESs or one with normal or improved BES performance. Of course, better services cost more. All alternative NoCs are based on contention-free routing, and, as a result, the Æthereal design flow can generate, program, and simulate them.[2]

Performance guarantees in networks

Researchers have paid much attention to the problem of building networks (both on- and off-chip) with predictable performance.[5-8] Fundamentally, there are two reasons for unpredictable network behavior: First, the network can drop packets as a result of buffer overflows, misrouting, router failure, and so forth. A given drop rate provides only a statistical reasoning about packet arrival, not a hard, 100% guarantee. Second, even if the network does not drop any packets, packets share resources (such as wires and buffers) with other packets. When two packets attempt to use the same resource at the same time, contention occurs and the network must either delay or drop one of the packets. Delayed packets often delay the packets following them, causing network congestion.

To offer guaranteed performance in NoCs, we observe that their characteristics differ from those of off-chip networks. First, NoCs can avoid dropping data, assuming that a SoC operates reliably (that is, its routers do not fail, misrouting does not occur, and so forth). Buffer overflow is avoidable by implementing flow control, as we describe later. This is much harder in off-chip networks, where wires are deeply pipelined and relatively longer, than in NoCs.

Secondly, contention exists in both NoCs and off-chip networks. There are several ways to address this problem in a NoC. First, contention and congestion are acceptable, as long as an upper bound is statically determinable. Rate-controlled and deadline-based arbitration schemes[7,8] are two ways to determine this upper bound. However, both schemes require large buffers, which make routers unacceptably expensive. Alternatively, contention is avoidable by ensuring that two packets are never at the same place at the same time. Circuit switching can enforce distinct places giving each communication its own wire; time multiplexing can enforce distinct times. Of course, combinations are also possible. In NoCs, wires are relatively short, and routers can synchronize relatively easily (for time-division multiplexing).

Multimedia systems contain many real-time data streams, but with different requirements. As a result, priority-based schemes do not work well, because all streams are equally important. Within the single real-time priority, all streams would behave like BESs between themselves, which is insufficient.

Given these insights, the Æthereal NoC uses contention-free routing, or pipelined time-division-multiplexed circuit switching, to implement its guaranteed performance services. Although all data streams have the same priority, they can obtain different bandwidth reservations. However, with higher average latency, time-division multiple access is not ideal for high-priority control traffic. Contention-free routing uses fewer wires than circuit switching and has minimal buffering in the routers.

Æthereal concepts

A NoC contains two components: routers and network interfaces. Network interfaces convert the IP view on communication (protocols, such as AXI—the Advanced eXtensible Interface—and OCP—the Open Core Protocol) to the router view on communication (packets). Here, we focus on routers and, later, briefly describe network interfaces.

Contention-free routing

Guaranteeing a certain level of performance (in terms of throughput and latency, for example) for a communication requires resource reservation (of wires and buffers) in the NoC. This is accomplished with connections, which are opened (reserving resources), used for some time, and then closed (releasing resources). In contention-free routing, or pipelined time-division-multiplexed circuit switching, a connection reserves wires and buffers for certain points in time.

A router with arity N (that is, N inputs and N outputs) uses a slot table to

- avoid contention on a link,
- divide up bandwidth per link between connections, and
- switch data to the correct output.

Every slot table T has S time slots (rows) and N router outputs (columns). There is a logical notion of *synchronicity*: All routers in the network occupy the same fixed-duration slot. In a slot, s, a network node (that is, a router or network interface) can read and write at most one block of data per input and output ports, respectively. In the next slot, $(s + 1)$ modulo S, the network node writes the read blocks to their appropriate output ports. Blocks thus propagate in a store-and-forward fashion and cannot deadlock. The latency that a block incurs per router equals the duration of a slot, and the slot reservations guarantee bandwidth in multiples of block size per S slots.

The slot table entries map outputs to inputs for every slot: $T(s, o) = i$, which means that blocks from input i (if present) proceed to output o at each $s + kS$ slot; $k \in N$. An entry is empty when there is no reservation for that output in that slot. There is no contention, by construction, because there is at most one input per output for each slot. Slot tables optimize away the header that specifies the path to the destination; as a result, GS blocks contain only data, and NoC efficiency improves.

Figure 1 illustrates contention-free routing with a snapshot of a router network and its corresponding slot tables. The network contains three routers, R_1, R_2, and R_3, at slot $s = 2$, which the pointer to the third entry in each table indicates (we number slots starting from zero). The size of the slot tables is $S = 4$, and the figure depicts only the relevant columns. The three gray arrows labeled a, b, and c represent connections; the three circles labeled a, b, and c represent blocks on the corresponding connections. Router R_1 switches block

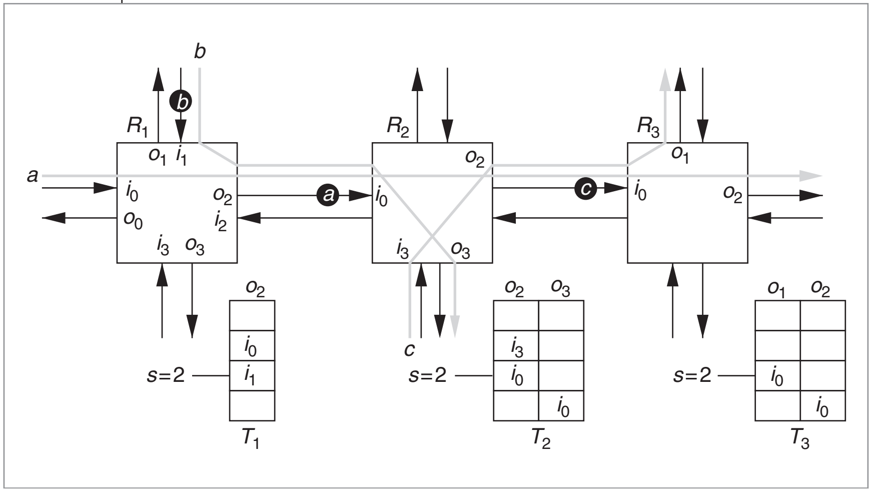

Figure 1. Contention-free routing: network of three routers (R_1, R_2, and R_3) at slot $s = 2$, with corresponding slot tables (T_1, T_2, and T_3).

b from input i_1 to output o_2, as slot table $T_1(2, o_2) = i_1$ indicates. Similarly, R_2 switches block a to output o_2, and R_3 switches block c to output o_1.

Contention-free routing depends on a logical notion of global synchronicity: All routers in the network must occupy the same fixed-duration slot. Obviously, this synchronicity is implementable with a single, centralized synchronous clock in combination with techniques like waterfall clock distribution and synchronous latency-insensitive design.[9] However, the notion of global synchronicity is also implementable in a distributed manner. For every slot synchronization, each router produces a token on every output before consuming a token on every input,[10] like a synchronous-data-flow (SDF) actor.[11] In other words, each router synchronizes every slot with all of its neighbors. Thus, all routers always remain in the same slot, and the NoC will run as fast as its slowest router. (Many extensions of this basic model are possible, including multicast, multirate SDF, and nonunit delays in routers.) The slot values for the slots reserved for a block along its source-to-destination path increase by one (modulo S). Assigning slots to connections in the network is an optimization problem. When designing for specific applications (connection requirements), designers can use sophisticated off-line global slot allocation algorithms. The resulting slot assignments are then programmable at runtime, as we show later. However, if connection requirements are only known at runtime, the design can use either relatively simple distributed algorithms, such as randomly picking slots, or simple centralized algorithms (assuming limited runtime computation resources).

Accordingly, the type of algorithm used to compute the slot allocation (design time or runtime, distributed or centralized) depends on how designers program the slot allocation into the NoC.

Contention-free GS architecture

The router architecture to implement contention-free routing is quite simple. Every input requires a queue for a single block, which is the minimum size. The queues connect to a switch, which is configured at every slot s by reading the $T(s, o) = i$ entries from the slot table. Configuration is simpler (that is, faster) than arbitration because contention does not occur by construction. GS blocks never wait, and link-level flow control between the routers is unnecessary. As a result, the switch is configurable without considering either the inputs or link-level flow control.

Best-effort architecture

The best-effort router is a conventional wormhole-routing, input-queued router. Round-robin arbitration of the switch occurs at the granularity of three words (a *flit*, or flow-control unit). The capacity of the input queues is a router parameter. We use link-level flow control between the routers to avoid queue overflow. BES packets use *source routing*: The packet header contains the path from source to destination. Each router removes as many bits ($\log_2 N$, the base 2 logarithm of the router arity) from the path as necessary to determine to which output the packet must go. Because of the absence of multiple buffer classes, BES packets can deadlock. We avoid deadlock with appropriate routing strategies.

Combined GS-BE architecture

The guaranteed performance of GS connections results from wire and buffer reservations in the NoC. To give 100% guarantees, these reservations must be for the worst case, wasting any unused bandwidth. To increase resource usage, we introduce BES connections that use all unused bandwidth (unreserved, as well as reserved but unused, slots). Our combined GS-BE router model consists of a GS router and a BES router placed in parallel. The BES router has a lower priority: A BES flit can use a link only when there is no GS block on the link.

Æthereal programming models

The Æthereal NoC, combining GS-BES routers, is programmable with slot allocations. Typically, however, a slot allocation occurs on a mode change, which occurs relatively infrequently. Our Æthereal design flow generates slot allocations for applications specified at design time.[2] We introduce two programming models: distributed and centralized. Although each has different advantages, both programming models use identical slot allocations, and both use the NoC to program themselves; this way, they avoid introducing an additional communication infrastructure just to program the network.

Distributed programming model

Our scalable distributed programming model[3] does not require a global view or centralized resources. It uses BES system packets to set up and tear down GS connections, much like asynchronous transfer mode (ATM).[12]

Initially, the slot table of every router is empty. Three BES system packets are used for configuration: SetUp, TearDown, and AckSetUp. These packets program the slot table of every router along their path. The SetUp packet creates a connection from a source to a destination and travels downstream (that is, toward the same destination as the data) along the same path as the data. When a SetUp packet successfully arrives at the destination, it indicates it's a successful connection by returning an AckSetUp upstream (that is, toward the data's source) along any path. When the connection creation fails, the SetUp packet is dropped and a TearDown packet is used to remove partial connections. TearDown packets can travel in either direction along the data path.

SetUp packets contain the data's source, the path to the destination, and slot number s. In every router along its path, the SetUp packet checks if the output to the next router in the path is free in the slot indicated by the packet. If it is free, the router reserves the output in that slot $[T(s, o) = i]$, and the SetUp packet is forwarded with an incremented (modulo S) slot. Otherwise, the SetUp packet is discarded and an upstream TearDown packet returns to the source. The TearDown packets must use the reverse path (which has already been assembled by the downstream SetUp packet). Thus, every path must be reversible; this is the only assumption we make about the network topology. The TearDown packet frees the slot and continues with a decremented slot number. Downstream TearDown packets work similarly and remove existing connections, starting from the source. A source has successfully opened a connection when it receives an AckSetUp; otherwise, it receives a TearDown. (Many extensions are possible, including multicast and GS system packets.)

Our distributed programming model uses slot tables to avoid contention, distributing them over the routers for scalable and consistent programming. For efficiency, the programming model is pipelined and concurrent (multiple system packets can be active in the network simultaneously, and they can come from the same source) and distributed (active in multiple routers). The outcome of programming can depend on the execution order of system packets, but it is always consistent. The time required to program the NoC depends on the load because system packets are BES.

Note that SetUp packets of different connections do not fail if the connections are set up with conflict-free slots. All execution orders of SetUp packets then give the same result, and design-time slot allocations are deterministically programmable at runtime. This applies when designers know the applications at design time. If applications are only known at runtime, then the distributed programming model is scalable, but implementing an efficient distributed runtime slot allocation algorithm might not be easy.

Centralized programming model

The distributed programming model is scalable, but we expect NoCs to be small in the near future. We do not expect a centralized programming scheme to be a bottleneck unless reconfiguration rates are high. Moreover, most current SoCs contain a central root or configuration process (or processor) that configures the system. For this reason, the Æthereal NoC also offers a centralized programming model.[4] Recall that the slot allocations are identical for both models; they differ only in how the NoC is programmed.

In the distributed programming model, network interfaces send SetUp packets to determine which slots they can use for a connection. However, a (central) third party, such as a root process, could directly program the network interfaces with the correct slots. As noted earlier, each router contains a slot table to

- allow for distributed and consistent programming using SetUp packets, and
- optimize away headers from GS blocks.

We can remove slot tables from all routers by omitting optimization. However, the network interfaces still require slot tables to determine when GS data can enter

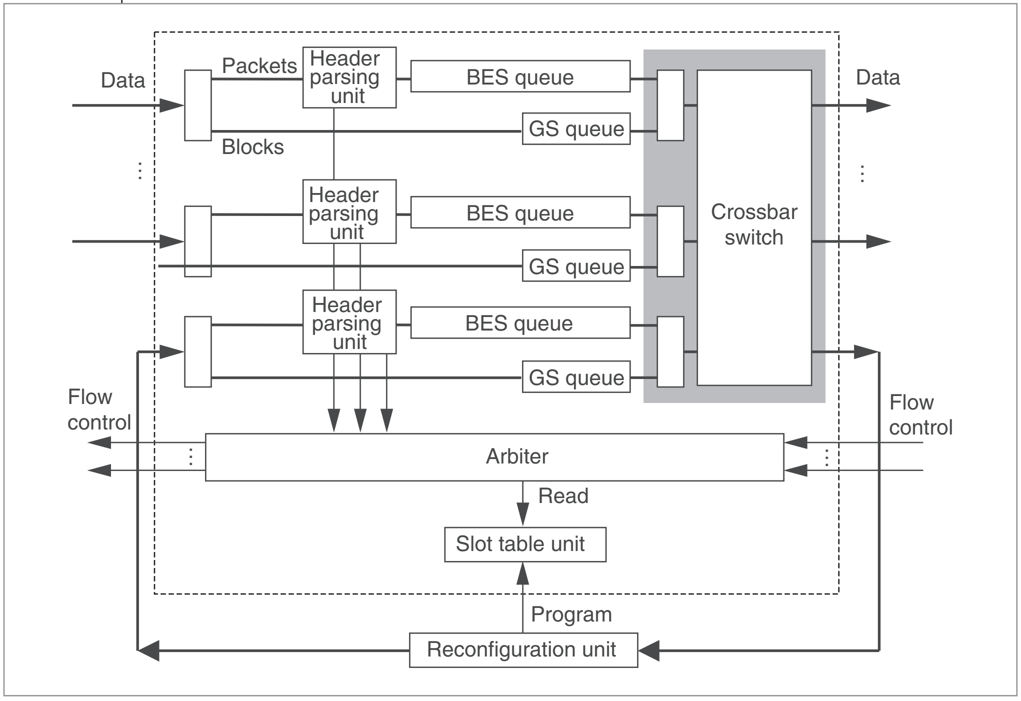

Figure 2. GS-BE distributed programming architecture.

the router network. Therefore, we convert GS blocks without a header to GS packets with a header. GS and BES packets differ only in their priority in the router. Note that GS packets never collide because they behave identically to GS blocks: Their propagation speeds in the NoC are fixed (one hop per flit delay), and their departure times in the network interface are equal.

The root process can program a connection from one network interface (A) to another (B) using abstract GS or BES ReserveSlot and FreeSlot packets that the network interfaces interpret. To program a network interface, the root sets up a connection to B, programs it, and removes the connection. The root then similarly programs A.

An alternative implementation uses memory-mapped I/Os (MMIOs) and read/write transactions instead of ReserveSlot and FreeSlot packets.[4] The NoC's registers are visible in the global memory map(s), just like other IPs. This style of programming (distributed shared memory) is conventional, but less abstract than message passing. (This holds even more strongly in the distributed programming model, which can also use the MMIO variant.)

These variations make the programming model less scalable and flexible. They are, however, closer to current practice and, as we shall show, significantly cheaper to implement.

Router architectures and implementations

We now describe four router architectures and implementations that originate from the contention-free routing concept. They vary in the programming model they support and hence also vary in cost. The Æthereal design flow currently generates only one type of router, but it is easy to add the rest.

From the conceptual point of view, network interfaces behave like routers. GS blocks or packets are injected into the NoC according to the network interface's slot table; BES packets can use the link otherwise. (Rădulescu et al. provide more details on network interfaces.[4])

All routers are implemented in a 0.13-μm technology and contain a 6×6 switch. A mesh requires five I/O pairs and uses one pair internally. The GS queues are one flit (the minimum), and BES queues are eight flits (fitting the largest packet). Given the prevalence of 32-bit address and data sizes, the data path is 34 bits wide, including two control bits. Except where indicated, area numbers include scan chains and are after layout with back-annotated timing, assuming worst-case military conditions.

GS-BE distributed programming router

Earlier, we defined a basic GS-BE router. In a GS-BE router, the GS and BES routers share the switch and links between them, as Figure 2 illustrates. The GS controller and BES arbiter must therefore be in lock step, and a GS block must be a multiple of the BES flit; we choose them equal in size for low cost and low latency. GS blocks bypass the header parsing unit (HPU) because they do not contain a header. The reconfiguration unit (RCU) is logically a separate module from the router, to which system packets are routed, just like any other output. An $(N+1) \times (N+1)$ router with an RCU effectively becomes an $N \times N$ router. Normal link-level flow control between the router and RCU ensures that RCU queues (of one flit) do not overflow, and the RCU programs the slot table unit (STU) via an MMIO port. Thus, the distributed programming model uses the BES system packets, which are translated into read/write transactions. Alternatively, one or more configuration modules can program the slot table directly using MMIO using any system-programming interconnect, such as a bus, token ring, or NoC.

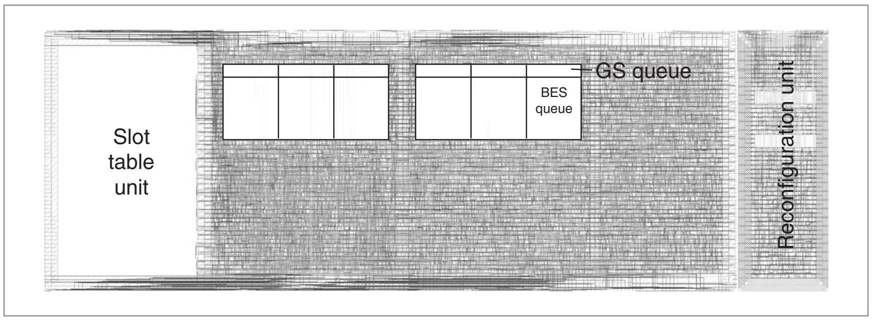

Figure 3. GS-BE router (left) and reconfiguration unit (right).

For an area-efficient GS-BE router, as we show in Figure 2, we developed dedicated hardware FIFOs to implement the GS and BES input queues. Figure 3 clearly shows the FIFOs (GS queue = one GS block; BES queue = eight BES flits), the SRAM for the slot table on the left ($S = 256$ slots), and the RCU with two one-flit queues (on the right). The total area is 0.24 mm^2. The data path operates at 500 MHz, giving 2 Gbytes/s raw bandwidth for each input and each output.

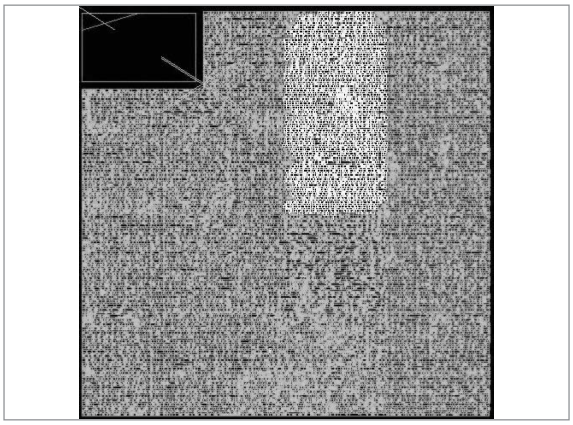

Figure 4. GS-BE distributed programming implementation (without RCU).

To highlight the importance of optimized memory architectures, we show a naive implementation of the router (without the RCU) in Figure 4. The same SRAM for the slot table is visible in the top-left corner, but we now use registers to implement the GS and BES queues. One of the six BES queues is shaded in gray to show that the router area is now dominated (80%) by the queues. The speed remains unchanged. This confirms that contention-free GS with minimal input queues and input queuing for BES is the right choice.

GS-BE centralized programming router

Figure 5 shows the combined GS-BE router that supports the centralized programming model. As before, the GS and BES routers are placed in parallel, and the GS controller and BES arbiter are in lock step. This time, however, GS blocks are packets with a header (because the router does not contain a slot table for routing); hence, they pass through the header parsing unit (HPU).

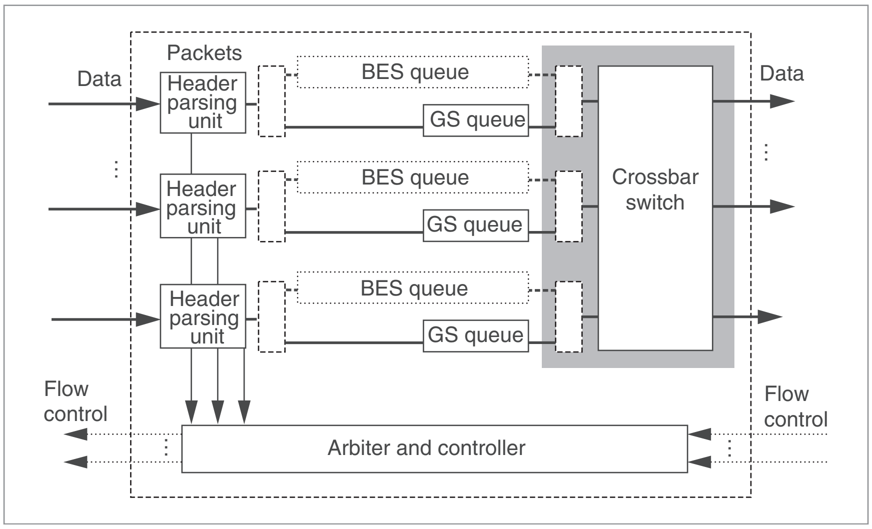

Figure 5. GS-BE centralized programming architecture.

Figure 6 shows the 0.13-mm^2, 500-MHz implementation of the GS-BE centralized programming architecture. To exemplify a cost-performance trade-off, we can combine the switch and the multiplexers in front of it into a $2N \times N$ switch. This reduces BES congestion but increases the router area to 0.175 mm^2.

GS centralized programming architecture

The architecture of a GS-only router with centralized programming equals that of a GS-BE router with centralized programming, without the dashed boxes and lines

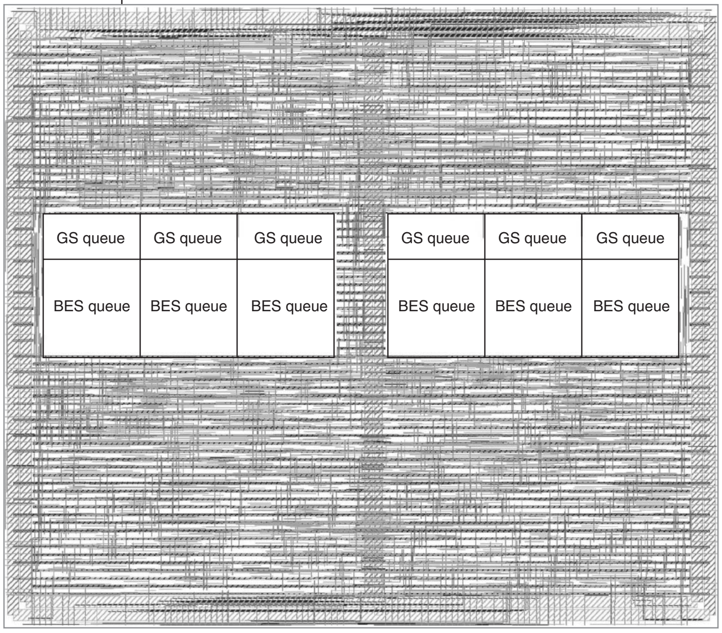

Figure 6. Implementation of GS-BE centralized programming architecture.

of Figure 5. The GS-only router architecture omits BES queues, (de)multiplexers, and link-level flow control. The arbiter becomes nothing more than a simple controller, and the flit size reduces to one word. As a result, the router is very small (0.033 mm^2) and fast (1 GHz), giving 4 Gbytes/s raw bandwidth for each input and each output (after synthesis). We have omitted the layout because it uses only standard cells and gives no architectural insights.

Router implementation overview

Table 1 gives an overview of the different router implementations. For each implementation, the table shows the service class (GS, BES, or both), supported programming model, number of effective inputs and outputs, and switch size (an architectural parameter). It also includes area (in square millimeters for a 0.13-μm technology) and frequency (megahertz).

Comparing the data, we can conclude that the GS-BE combination is relatively expensive. The GS-only NoC, however, is attractive: It provides twice the performance for a quarter of the area. But if we use the GS-only NoC even for BES traffic, we might need some additional GS routers. The trade-off then is the number of global inter-router wires (larger with GS only; smaller with combined GS-BE) versus area (smaller with GS only; larger with combined GS-BE). Thus, Table 1 shows that a system architect can trade off the desired programming model, performance, and cost to achieve a balanced solution for the SoC as a whole.

OF THE FOUR ROUTER ARCHITECTURES and implementations, the two ends of the spectrum are the GS-BE router with distributed programming and system packets, and the GS router with centralized programming and distributed shared memory. The former is scalable and future proof, whereas the latter is faster, cheaper to implement, and closer to current practice. ■

References

1. L. Benini and G. De Micheli, “Networks on Chips: A New SoC Paradigm,” *IEEE Computer*, vol. 35, no. 1, Jan. 2002, pp. 70-80.
2. K. Goossens et al., “A Design Flow for Application-Specific Networks on Chip with Guaranteed Performance to Accelerate SOC Design and Verification,” *Proc. Design, Automation and Test in Europe* (DATE 05), IEEE CS Press, 2005, pp. 1182-1187.
3. E. Rijpkema et al., “Trade-offs in the Design of a Router with Both Guaranteed and Best-Effort Services for Networks on Chip,” *Proc. IEE Computers and Digital Techniques*, IEEE Press, 2003, vol. 150, no. 5, pp. 294-302.
4. A. Rădulescu et al., “An Efficient On-Chip Network Inter-

Table 1. Overview of router architectures and implementations.

Router implementation	Service class	Programming model	No. of effective inputs and outputs	Switch size	Area (mm^2)	Frequency (MHz)
GS-BE distributed programming	GS and BES	Distributed	5	$N \times N$	0.24	500
GS-BE centralized programming	GS and BES	Centralized	6	$2N \times N$	0.175	500
GS-BE centralized programming	GS and BES	Centralized	6	$N \times N$	0.13	500
GS centralized programming	GS	Centralized	6	$N \times N$	0.033	1,000

face Offering Guaranteed Services, Shared-Memory Abstraction, and Flexible Network Programming," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 1, Jan. 2005, pp. 4-17.
5. T. Bjerregaard and J. Sparso, "A Router Architecture for Connection-Oriented Service Guarantees in the MANGO Clockless Network-on-Chip," *Proc. Design, Automation and Test in Europe* (DATE 05), IEEE CS Press, 2005, pp. 1226-1231.
6. M. Millberg et al., "Guaranteed Bandwidth Using Looped Containers in Temporally Disjoint Networks within the Nostrum Network on Chip," *Proc. Design, Automation and Test in Europe* (DATE 04), IEEE CS Press, 2004, pp. 20890-20895.
7. J. Rexford, "Tailoring Router Architectures to Performance Requirements in Cut-Through Networks," PhD thesis, Dept. Computer Science and Eng., Univ. of Michigan, 1999.
8. H. Zhang, "Service Disciplines for Guaranteed Performance Service in Packet-Switching Networks," *Proc. IEEE*, IEEE Press, 1995, pp. 1374-1396.
9. A. Edman and C. Svensson, "Timing Closure through a Globally Synchronous, Timing Partitioned Design Methodology," *Proc. 41st Design Automation Conf.* (DAC 04), IEEE CS Press, 2004, pp. 71-74.
10. E. Rijpkema et al., "A Router Architecture for Networks on Silicon," *Proc. Workshop on Embedded Systems*, 2001; http://www.homepages.inf.ed.ac.uk/kgoossen/2001-progress.pdf.
11. E. Lee and D. Messerschmitt, "Synchronous Data Flow," *Proc. IEEE*, IEEE Press, 1987, pp. 1235-1245.
12. ATM Forum, *ATM User-Network Interface Specification*, Prentice Hall, 1994.

Kees Goossens is a principal research scientist at Philips Research Laboratories in Eindhoven, The Netherlands. His research interests include networks on chips for consumer electronics systems. Goossens has a BSc in computer science from the University of Wales and a PhD in computer science from the University of Edinburgh. He is a member of the IEEE.

John Dielissen is a research scientist at Philips Research Laboratories in Eindhoven, The Netherlands. His research interests include algorithms and ASIC implementations of IP in the communications domain, including NoCs and channel coding. Dielissen has an MSc in electrical engineering from Eindhoven University of Technology.

Andrei Rădulescu is a senior research scientist at Philips Research Laboratories in Eindhoven, The Netherlands. His research interests include on- and off-chip networks, quality of service, protocols, resource mapping and scheduling, and distributed systems. Rădulescu has an MSc in computer science from Polytechnica University of Bucharest, Romania, and a PhD in computer engineering from Delft University of Technology, The Netherlands.

■ Direct questions and comments about this article to Kees Goossens, Philips Research Laboratories, Room 3.16, Building WDC, Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands; kees.goossens@philips.com.