System-Level Design of NoC-Based Dependable Embedded Systems

Mihkel Tagel, Peeter Ellervee, Gert Jervan
Department of Computer Engineering, Tallinn University of Technology, Estonia

ABSTRACT

Technology scaling into subnanometer range will have impact on the manufacturing yield and quality. At the same time, complexity and communication requirements of systems-on-chip (SoC) are increasing, thus making a SoC designer goal to design a fault-free system a very difficult task. Network-on-chip (NoC) has been proposed as one of the alternatives to solve some of the on-chip communication problems and to address dependability at various levels of abstraction. This chapter concentrates on system-level design issues of NoC-based systems. It describes various methods proposed for NoC architecture analysis and optimization, and gives an overview of different system-level fault tolerance methods. Finally, the chapter presents a system-level design framework for performing design space exploration for dependable NoC-based systems.

Introduction

As technologies advance and semiconductor process dimensions shrink into the nanometer and subnanometer range, the high degree of sensitivity to defects begins to impact the overall yield and quality. The International Technology Roadmap for Semiconductors (2007) states that relaxing the requirement of 100% correctness for devices and interconnects may dramatically reduce costs of manufacturing, verification, and test. Such a paradigm shift is likely forced by the technology scaling that leads to more transient and permanent failures of signals, logic values, devices, and interconnects. In consumer electronics, where the reliability has not been a major concern so far, the design process has to be changed. Otherwise, there is a high loss in terms of faulty devices due to problems stemming from the nanometer and subnanometer manufacturing process.

There has been a lot of research made on system reliability in different computing domains by employing data encoding, duplicating system components or software-based fault tolerance techniques. This research has mostly had either focus on low level hardware reliability or covered the distributed systems. Due to future design complexities and technology scaling, it is infeasible to concentrate only onto low level reliability analysis and improvement. We should fill the gap by looking at the application level. We have to assume that the manufactured devices might contain faults and an application, running on the system, must be aware that the underlying hardware is not perfect.

The advances in design methods and tools have enabled integration of increasing number of components on a chip. Design space exploration of such many-core systems-on-chip (SoC) has been extensively studied, whereas the main focus has been so far on the computational aspect. With the increasing number of on-chip components and further advances in semiconductor technologies, the communication complexity increases and there is a need for an alternative to the traditional bus-based or point-to-point communication architectures.

Network-on-chip (NoC) is one of the possibilities to overcome some of the on-chip communication problems. In such NoC-based systems, the communication is achieved by routing packets through the network infrastructure rather than routing global wires. However, communication parameters (inter-task communication volume, link latency and bandwidth, buffer size) might have major impact to the performance of applications implemented on NoCs. Therefore, in order to guarantee predictable behaviour and to satisfy performance constraints, a careful selection of application partitioning, mapping and synthesis algorithms is required. NoC platform provides also additional flexibility to tolerate faults and guarantees system reliability. Many authors have addressed these problems but most of the emphasis has been on the systems based on bus-based or point-to-point communication (Marculescu, Ogras, Li-Shiuan Peh Jerger, & Hoskote, 2009). However, a complete system-level design flow, taking into account the NoC network modelling and dependability issues, is still missing.

This chapter first analyzes the problems related to the development of dependable systems-on-chip. It outlines challenges, specifies problems and examines the work that has been done in different NoC research areas relevant to this chapter. We will give an overview of the state-of-the-art in system-level design of traditional and NoC-based systems and describe briefly various methods proposed for system-level architecture analysis and optimization, such as application mapping, scheduling, communication analysis and synthesis. The chapter gives also an overview of different fault-tolerance techniques that have been successfully applied to bus-based systems. It analyzes their shortcomings and applicability to the network-based systems.

The second part of the chapter describes our system-level design framework for performing design space exploration for NoC-based systems. It concentrates mainly on the specifics of the NoC-based systems, such as network modelling and communication synthesis. Finally, the chapter addresses the dependability issues and provides methods for developing fault-tolerant NoC-based embedded systems.

BACKGROUND AND RELATED WORK

In this section we first describe the design challenges that have emerged together with the technology scaling and due to increase of the design complexity. We give an overview of the key concepts and NoC terminology. The second part of this section is devoted to system-level design and dependability issues.

Design Challenges of Systems-on-Chip

The advances in design methods and tools have enabled integration of increasing number of components on the chip. Design space exploration of such many-core SoCs has been extensively studied, whereas the main focus has been so far on the computational aspect. With the increasing number of on-chip components and further advances in semiconductor technologies, the communication complexity increases and there is a need for an alternative to the traditional bus-based or point-to-point communication architecture. The main challenges in the current SoC design methodologies are:

- Deep submicron effects and variability – the scaling of feature sizes in semiconductor industry have given the ability to increase performance while lowering the power consumption. However, with feature sizes reducing below 40 nm it is getting hard to achieve favourable cost versus performance/power trade-offs in future CMOS technologies (International Technology Roadmap for Semiconductors, 2007;

Konstadinidis, 2009). The emergence of deep submicron noise in the form of cross-talk, leakage, supply noise, as well as process variations is making it increasingly hard to achieve the desired level of noise-immunity while maintaining the historic improvement trends in performance and energy-efficiency (Shanbhag, Soumyanath, & Martin, 2000; Kahng, 2007). Interconnects also add a new dimension to design complexity. As interconnects also shrink and come closer together, previously negligible physical effects like crosstalk become significant (Hamilton, 1999; Ho, Mai, & Horowitz, 2001).

- Global synchrony – SoCs are traditionally based on a bus architecture where system modules exchange data via a synchronous central bus. When number of components increase rapidly, we have a situation where the clock signal cannot be distributed over the entire SoC during one clock cycle. Ho et al. (2001) describe that while local wires scale in performance, global and fixed-length wires do not. The technology scaling is more rapid for gates than for wires. It affects the design productivity and reliability of the devices. Optimization techniques, such as optimal wire sizing, buffer insertion, and simultaneous device and buffer sizing are solving only some of the problems. As feature size continues to shrink, the interconnect itself becomes complex circuitry in its own (Hamilton, 1999). Consequently, increased SoC complexity and feature size scaling below 40 nm requires alternative means for providing scalable and efficient interconnects. Globally asynchronous locally synchronous (GALS) design approach has been proposed as a feasible solution for communication intensive complex SoCs. In 2000, Agarwal, Hrishikesh, Keckler, & Burger have examined the effects of technology scaling on wire delays and clock speeds, and measured the expected performance of a modern microprocessor core in CMOS technologies down to 35 nm. Their estimation shows that even under the best conditions the latency across the chip in a top-level metal wire will be 12-32 cycles (depending on the clock rate). Jason Cong's simulations at the 70 nm level suggest that delays on local interconnect will decrease by more than 50 percent, whereas delays on non-optimized global interconnect will increase by 150 percent (from 2 ns to 3.5 ns) (Hamilton, 1999). GALS systems contain several independent synchronous blocks that operate using their own local clocks and communicate asynchronously with each other. The main feature of these systems is the absence of a global timing reference and the use of several distinct local clocks (or clock domains), possibly running at different frequencies (Iyer & Marculescu, 2002).
- Productivity gap – chip design has become so complex that designers need more education, experience, and exposure to a broad range of fields (device physics, wafer processing, analogue effects, digital systems) to understand how all these aspects come together. For the same reasons, designers need smarter tools that comprehend distributed effects like crosstalk (Hamilton, 1999). The complexity and cost of design and verification of multi-core products has rapidly increased to the point where developers devote thousands of engineer-years to a single design, yet processors reach market with hundreds of bugs (Allan, Edenfeld, Joyner, Kahng, Rodgers, & Zorian, 2002). The primary focus of consumer-products in CMOS process development is the integration density. By allowing to pack a greater functionality onto a smaller area of silicon, the higher integration density and lower cost can be achieved. For consumer applications, Moore's law may continue for as long as the cost per function decreases from node to node (Claasen, 2006). To bridge the technology and productivity gap, the computation need to be decoupled from the communication. The communication platform should be

scalable and predictable in terms of performance and electrical properties. It should enable high intellectual property (IP) core reuse by using standard interfaces to connect IP-s to the interconnect.

- Power and thermal management – interconnect wires account for a significant fraction (up to 50%) of the energy consumed in an integrated circuit and is expected to grow in the future (Raghunathan, Srivastava, & Gupta, 2003). Feature size scaling increases power density on the chip die that in turn can produce an increase in the chip temperature. The rapidly increasing proportion of the consumer electronics market represented by handheld, battery-powered, equipment also means that low power consumption has become a critical design requirement that must be addressed (Claasen, 2006).
- Verification and design for test – the increasing complexity of SoCs and the different set of tests required by deep submicron process technologies (for example tests for delay faults) has increased test data volume and test time to the extent that many SoCs no longer fit comfortably within the capabilities of automated test equipment (ATE) (Claasen, 2006). As a result, the cost of test has been rapidly increasing. Due to process variability, the reliability of the devices is not anymore a concern of only safety-critical applications but also a concern in consumer electronics. The products need to be designed to tolerate certain number of manufacturing (permanent) or transient faults.

To overcome some of the above challenges the network-on-chip paradigm has been proposed. While computer networking techniques are well known already from the 80’s, the paradigm shift reached to the chips in the beginning of this millennium. There were several independent research groups (Benini & De Micheli, 2002; Dally & Towles, 2001; Guerrier & Greiner, 2000; Hemani et al., 2000; Rijpkema, Goossens, & Wielage, 2001; Sgroi et al., 2001) introducing networking ideas to embedded systems.

Network-on-chip as a new design paradigm

In 2000, Guerrier and Greiner proposed a scalable, programmable, integrated network (SPIN) for packet-switched system-on-chip interconnections. They were using fat-tree topology and wormhole switching with two one-way 32-bit data paths having credit-based flow control. They proposed a router design with dedicated input buffers and shared output buffers, estimated the router cost and network performance. The term “network-on-chip” was first used by Hemani et al. in November 2000. The authors introduced the concept of reconfigurable network of resources and its associated methodology as solution to the design productivity problem. In June 2001, Dally and Towles proposed NoC as general-purpose on-chip interconnection network to connect IP cores replacing design-specific global on-chip wiring. It was demonstrated that using a network to replace global wiring has advantages in structure, performance, and modularity. The GigaScale Research Center suggested a layered approach similar to that defined for communication networks to address the problem of connecting a large number of IP cores. Additionally the need for a set of new generation methodologies and tools were described (Sgroi et al., 2001). In October 2001, researchers from Philips Research presented a quality of service (QoS) router architecture supporting both best-effort and guaranteed-throughput (Rijpkema et al., 2001). In January 2002, Benini and De Micheli formulated NoC as a new SoC design paradigm.

During the years many, NoC research platforms have been developed such as Aethereal (Goossens, Dielissen, & Radulescu, 2005), MANGO (Bjerregaard & Sparso, 2005), Nostrum

(Kumar et al., 2002), SPIN (Guerrier & Greiner, 2000), Xpipes (Bertozzi & Benini, 2004), CHAIN (Felicijan, Bainbridge, & Furber, 2003). Commercial NoC platforms include Arteris (Arteris, 2009), STNoC (STMicroelectronics, 2009), Silistix (Silistix, 2009) and Sonics (Sonics, 2009).

Current and future directions of on-chip networks include 3D NoCs (Feero & Pande, 2007; Pavlidis & Friedman, 2007; Murali, Seiculescu, Benini, & De Micheli, 2009) and optical interconnects (Haurylau et al., 2006). Both emerged in the middle of 90’s in various forms. 3D NoCs are having its roots in 2001 (Banerjee, Souri, Kapur, & Saraswat, 2001).

Comparison with bus based systems and macro networks

Point-to-point connections (circuit switching), common to SoC, are replaced in NoC by dividing the messages into packets (packet switching). Each component stores its state and exchanges data autonomously with others. Such systems are by their nature GALS systems, containing several independent synchronous blocks that operate with their own local clocks and communicate asynchronously with each other (Iyer & Marculescu, 2002). Having multiple different network routes available for the data transmission makes NoCs to be adaptive – to balance the network load, for instance.

The communication platform limitations, data throughput, reliability and QoS are more difficult to address in NoC architectures than in computer networks. The NoC components (memory, resources) are relatively more expensive, whereas the number of point-to-point links is larger on-chip than off-chip. On-chip wires are also relatively shorter than the off-chip ones, thus allowing a much tighter synchronization than off-chip. On one hand, only a minimum design overhead is allowed that is needed to guarantee the reliable data transfer. On the other hand, the on-chip network must handle the data ordering and flow control issues (Radulescu & Goossens, 2002). The packets might appear at the destination resource out of order – they need to be buffered and put into the correct order.

Principles of Networks-on-Chip

In this section we provide an overview of the key concepts and terminology of NoCs. The NoC design paradigm has two good properties to handle the SoC design complexity – predictability and reusability. The throughput, electrical properties, design and verification time are easier to predict due to the regular structure of the NoC. We can connect to the network any IP component that has the appropriate network interface. The NoC paradigm does not set any limits to the number of components. The components and also the communication platform are reusable – the designer needs to design, optimise and verify them once. The layered network architecture provides the needed communication and network services enabling the functionality reuse (Jantsch & Tenhunen, 2003).

NoC decouples communication from computation and provides a flexible and reusable communication platform. The interconnection network is a shared resource that the designer can utilize. To design an on-chip communication infrastructure and to meet the performance requirements of an application, the designer has certain design alternatives that are governed by topology, switching, routing and flow control of the network. NoC provides the communication infrastructure for resources. Resources can be heterogeneous. A resource can be memory, processor core, DSP, reconfigurable block or any IP block that conforms to the network interface

(NI). Every resource is connected to switch via resource network interface (RNI). Instead of dedicated point-to-point channels between two IP cores, the interconnection network is implemented as set of shared routers and communication links between the routers. The way the routers are connected with each other defines the network topology. Data to be transferred between communicating nodes is called a message. As messages can have varying sizes it is infeasible to design routers to handle unbounded amounts of data. Instead, messages are divided into smaller bounded flow control units. The way a message is split and transferred through the routers is called switching. Usually there are alternative paths to deliver a message from source to destination. An algorithm to choose between such paths is called routing. A good routing algorithm finds usually minimal paths while avoiding deadlocks. Another alternative would be to balance the network load. Flow control handles network resource accesses. If a network is not able to handle the current communication load the flow control might forward more critical messages while dropping or re-routing the non-critical ones. An effective network design maximises the throughput and decreases network latency and communication conflicts (Dally & Towles, 2004).

Topology

Topology refers to the physical structure of the network (how resources and switches are connected to each other). It defines connectivity and routing possibilities between the nodes affecting therefore performance of the network and design of the router. Topologies can be divided into two classes by their regularity – regular and application specific. The regular topologies can be described in terms of *k*-ary *n*-cube, where *k* is the degree of each dimension and *n* is the number of dimensions (Dally, 1990). Regular topology is not the most efficient in terms of manufacturing but allows easier routing algorithms and better predictability. The regularity aims for design reuse and scalability while application specific topologies target performance and power consumption. Most NoCs implement regular forms of network topology that can be laid out on a chip surface, for example *k*-ary *2*-cube meshes (Kumar et al., 2002) and torus (Dally & Towles, 2001). The *k*-ary tree and *k*-ary *n*-dimensional fat tree (Adriahantenaina, Charlery, Greiner, Mortiez, & Zeferino, 2003) are two alternative regular NoC topologies. Recent research in this area is devoted to 3-dimensional NoCs. Each router in a 2D NoC is connected to a neighbouring router in one of four directions. Consequently, each router has five ports. Alternatively, in a 3D NoC, the router typically connects to two additional neighbouring routers located on the adjacent physical planes (Pavlidis & Friedman, 2007). Figure 1 shows examples of various regular and application specific topologies, including 3D.

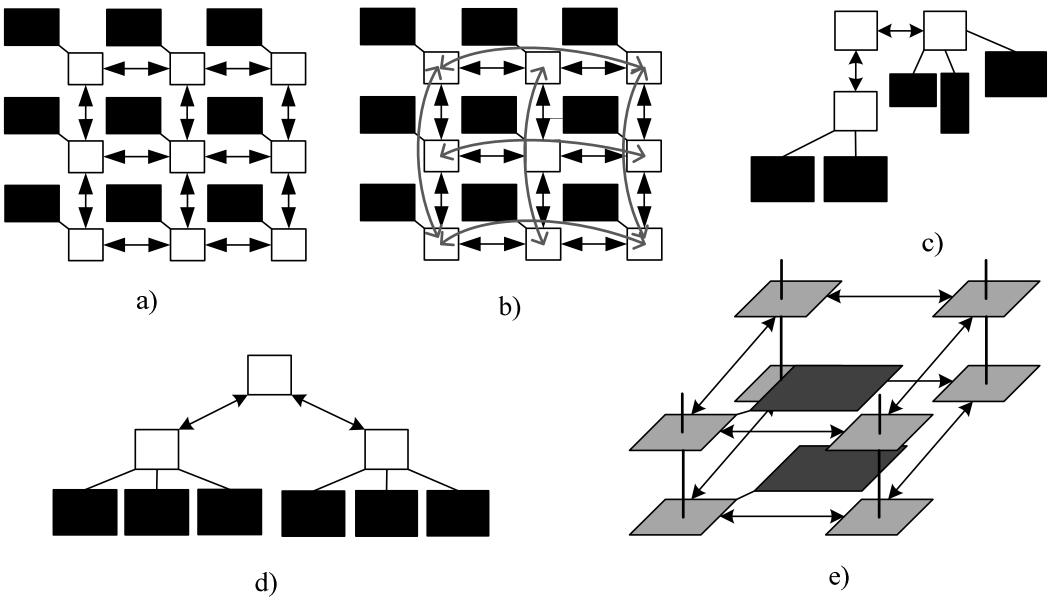


Figure 1. Regular topologies. Examples are (a) 4-ary 2-cube mesh, (b) 4-ary 2-cube torus, (c) application specific, (d) binary 2-ary tree and (e) 3D mesh

Switching method

Switching method determines how a message traverses its route. There are two main switching methods – circuit switching and packet switching. Circuit switching is a form of bufferless flow control that operates by first allocating channels to form a circuit from source to destination and then sending messages along this circuit. After the data transmission, the circuit can be deallocated and released for other communication. Circuit switching is connection-oriented, meaning that there is an explicit connection establishment (Lu, 2007). In packet switching the messages are split into packets. Depending of switching methods, a packet can be further divided into smaller flow control units (flits). A packet consists usually of a header, a payload and a tail. The packet header contains routing information, while the payload carries the actual data. The tail indicates the end of a packet and can contain also error-checking code. Packet switching can be either connection-oriented or connection-less. In contrast to the connection-oriented switching, in the connection-less the packets are routed in a non-guaranteed manner. There is no dedicated circuit built between the source and destination nodes.

Most common packet switching techniques include store-and-forward, virtual cut-through and wormhole switching.

- Store-and-forward – when a packet reaches an intermediate node, the entire packet is stored in a packet buffer. The packet is forwarded to the next selected neighbour router after the neighbouring router has an available buffer. Store-and-forward is simple to implement but it has major drawbacks. First, it has to buffer the entire packet before forwarding it to the downstream router. This has a negative effect on router area overhead. Second, the network latency is proportional to the distance between the source and the destination nodes. The network latency of store-and-forward can be calculated (Ni & McKinley, 1993) as

$$Latency_{store\text{-}and\text{-}forward} = (L/B)D \qquad (1)$$

where *L* is message size, *B* is channel bandwidth and *D* is distance in hops. The smallest flow control unit is a packet.

- Virtual cut-through – to decrease the amount of time spent transmitting data Kermani and Kleinrock (1979) introduced the virtual cut-through switching method. In the virtual cut-through a packet is stored at an intermediate node only if the next required channel is busy. The network latency of the virtual cut-through can be calculated as
$$Latency_{virtual\ cut\text{-}through} = (L_h/B)D + L/B \qquad (2)$$
where L_h is size of the header field. Usually the message size is times bigger than header field and therefore the distance *D* will produce a negligible effect on the network latency. The smallest flow control unit is a packet.
- Wormhole – operates like virtual cut-through but with channel and buffers allocated to flits rather than packets (Dally & Towles, 2004). A packet is divided into smaller flow control units called flits. There are three types of flits – body, header, and tail. The header flit governs the route. As the header advances along its specified route, the rest of the flits follow in a pipeline fashion. If a channel is busy, the header flit gets blocked and waits the channel to become available. Rather than collecting and buffering the remaining flits in the current blocked router, the flits stay in flit buffers along the established route. Body flits carry the data. The tail flit is handled like a body flit but its main purpose is to release the acquired flit buffers and channels. The network latency of wormhole switching can be calculated according to Ni & McKinley (1993) as
$$Latency_{wormhole} = (L_f/B)D + L/B \qquad (3)$$
where L_f is size of the flit. In similar way to virtual cut-through distance *D* has not significant effect on the network latency unless it is very large. Wormhole switching is more efficient than virtual cut-through in terms of the buffer space. However, this comes at the expense of some throughput since wormhole flow control may block a channel mid-packet (Dally & Towles, 2004).
- Virtual channels – associates several virtual channels (channel state and flit buffers) with a single physical channel. Virtual channels overcome the blocking problems of the wormhole switching by allowing other packets to use the channel bandwidth that would otherwise be left idle when a packet blocks (Dally & Towles, 2004). It requires an effective method to allocate the optimal number of virtual channels. Allocating the virtual channels uniformly results in a waste of area and significant leakage power, especially at nanoscale (Huang, Ogras, & Marculescu, 2007).

Routing

Routing algorithm determines the routing paths the packets may follow through the network. Routing algorithms can be divided in terms of path diversity and adaptivity into deterministic, oblivious and adaptive routing. Deterministic routing chooses always the same path given the same source and destination node. An example is source ordered XY routing. In XY routing the processing cores are numbered by their geographical coordinates. Packets are routed first via X and then via Y-axis by comparing the source and destination coordinate. Deterministic routing has small implementation overhead but it can cause load imbalance on network links. Deterministic routing cannot also tolerate permanent faults in NoC and re-route the packets. Oblivious routing considers all possible multiple paths from the source node to destination but does not take the network state into account. Adaptive routing distributes the load dynamically in response to the network load. For example, it re-routes packets in order to avoid congested area or failed links. Adaptive routing has been favourable providing high fault tolerance. The drawbacks include higher modelling and implementation complexity. Deterministic routing algorithms guarantee in-order delivery while in adaptive routing buffering might be needed at the receiver side to re-order the packets.

There are two important terms when talking about routing – deadlock and livelock. Deadlock occurs in an interconnection network when group of packets are unable to progress because they are waiting on one another to release resources, usually buffers or channels (Dally & Towles, 2004). Deadlocks have fatal effects on a network. Therefore deadlock avoidance or deadlock recovery should be considered for routing algorithms that tend to deadlock. Another problematic network phenomenon is livelock. In livelock, packets continue to move through the network, but they do not make progress toward their destinations (Dally & Towles, 2004). It can happen for example when packets are allowed to take not the shortest routes. Usually it is being handled by allowing a certain number of misroutes after which the packet is discarded and need to be re-submitted.

Flow control

Flow control deals with network load monitoring and congestion resolution. Due to the limited buffers and throughput, the packets may be blocked and flow control decides how to resolve this situation. The flow control techniques can be divided into two – bufferless and buffered flow controls. The bufferless flow control is the simplest in its implementation. In bufferless flow control there are no buffers in the switches. The link bandwidth is the resource to be acquired and allocated. There is need for an arbitration to choose between the competing communications. Unavailable bandwidth means that a message needs to be misrouted or dropped. Dropped message has to be resent by the source. Misrouting and message dropping both increase latency and decrease efficiency (throughput) of the network. Deflection routing is an example of the bufferless flow control. In deflection routing, an arbitrary routing algorithm chooses a routing path, while deflection policy is handling the resource contentions. In the case of network contention, the deflection policy grants link bandwidth to the higher priority messages and misroutes the lower priority messages. Deflection routing allows low overhead switch design while at the same time provides adaptivity for network load and resilience for permanent link faults.

In the buffered flow control, a switch has buffers to store the flow control unit(s) until bandwidth can be allocated to the communication on outgoing link. The granularity of the flow control unit can be different. In store-and-forward and virtual cut-through both the link bandwidth and buffers are allocated in terms of packets but in wormhole switching in flits. In buffered flow control, it is crucial to distribute the buffer availability information between the neighbouring routers. If buffers of the upstream routers are full, the downstream routers must stop transmitting any further flow control units. The flow control accounting is done at link level. The most common flow control accounting techniques are credit-based, on/off and ack/nack (Dally & Towles, 2004).

Quality of Service

Quality of Service (QoS) gives guarantees on packet delivery. The guarantees include correctness of the result, completion of the transmission, and bounds on the performance (Lu, 2007). The network traffic is divided usually into two service classes – best-effort and guaranteed. A best-effort service is connectionless. Packets are delivered when possible depending on the current network condition. A guaranteed service is typically connection-oriented. The guaranteed service class packets are prioritized over the best-effort traffic. In addition, guaranteed service avoids network congestions by establishing a virtual circuit and reserving the resources. It can be implemented for example by using multiple timeslots (Time Division Multiple Access, TDMA) or virtual channels.

Further reading

There is comprehensive survey of research and practices of network-on-chip (Bjerregaard & Mahadevan, 2006), survey of different NoC implementations (Salminen, Kulmala, & Hämäläinen, 2008) and overview of outstanding research problems in NoC design (Marculescu et al., 2009).

System-level design

System-level design starts with the specification of a system to be designed and concludes with integration of the created hardware and software. Of course, considering the complexity of systems, a systematic approach is needed and the system-level design methodologies try to take into account important implementation issues already at higher abstraction levels.

Traditional system-level design flow

Having its roots in the end of the 80's, system-level design is a hierarchical process that begins with a high-level description of the complete system and works down to fine grained descriptions of individual systems modules (Stressing, 1989). Initially, the descriptions of a system are independent from the implementation technology. There are even no details whether some component of the system should be implemented in hardware or in software. Therefore, early system descriptions are more behavioural than structural, focusing on system functionality and performance specification rather than interconnects and modules. In addition to the system specification, it is important to have possibility to verify the performance and functional specification. A specification at the system-level should be created in such a way that its correctness can be validated by simulation. Such a model is often referred to as simulatable specification. In addition, a model at the system-level should be expressed in a form that enables verification that further refinements correctly implement the model (Ashenden & Wilsey, 1998). Possible approaches include behavioural synthesis (correct by construction), and formal verification using model checking and equivalence checking (Ashenden & Wilsey, 1998). A third essential element of system-level design is the exploration of various design alternatives. For example, whether to implement a function in hardware or in software, whether to solve it with sequential or parallel algorithm. The analysis of trade-offs between design alternatives is a key element of system-level design and shows the quality of the particular system-level design flow. It is important that a system-level design flow is supported by system-level tools – simulators/verifiers, estimators and partitioners. The first system-level design tools were introduced in 1980 by Endot, a company formed out of the staff at the Case Western Reserve University (Stressing, 1989). The need for the system-level design tools was the complexity of the aerospace and defence systems that were then being developed, but it soon became apparent that these tools were applicable to design complex digital hardware/software systems of any type (Stressing, 1989).

At the system-level, a system can be modelled as a collection of active objects that react to events, including communication of data between objects and stimuli from the surrounding environment. Abstractions are needed in a number of areas to make the system-level behavioural modelling tractable in the following views:

- abstraction of data,
- abstraction of concurrency, and
- abstraction of communication and timing (Ashenden & Wilsey, 1998).

Of course, different views can stress on different abstractions, e.g., concurrency is replaced by calculation, and communication and timing are looked at separately (Jantsch, 2003).

The classical system-level design flow consists of several consecutive design tasks with loopbacks to previous steps (Lagnese & Thomas, 1989). An input to the system-level design flow is a system specification that is represented in a formal way, e.g., dataflow or task graph. In the dataflow graph, the nodes represent operators and the arcs between them represent data and control dependencies like in task graphs. The operators are scheduled into time slots called control steps. Scheduling determines the execution order of the operators. The scheduling can be either static or dynamic. In the dynamic scheduling, the start times are obtained during execution (online) based on priorities assigned to processes. In the static scheduling, the start times of the processes are determined at the design time (off-line) and stored in the form of schedule tables. Scheduling sets lower limits on the hardware because operators scheduled into the same control step cannot share the hardware. Thus, scheduling has a great impact on the allocation of the hardware. After scheduling the data-flow operators and values are mapped to allocated hardware. If the hardware platform is given with the system specification then designer can also start first with the mapping and then perform the scheduling. Since both, mapping and scheduling, are NP-hard, the parallel execution of those design phases is extremely difficult. When the results of the system-level design flow do not satisfy the initial requirements, either the mapping or the scheduling of application’s components can be changed. If no feasible solution is found, changes are needed in the system specification or in the architecture. After an acceptable schedule is found, lower abstraction-levels of hardware/software co-design will follow.

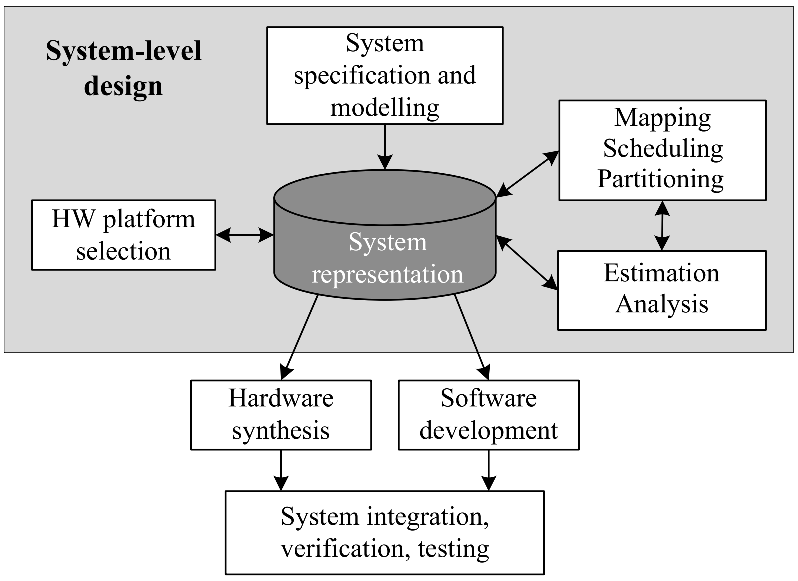


Figure 2. Classical system-level design flow

Refinement to a software implementation is facilitated by a system-level modelling language that is closely related to programming languages. In principle, both the hardware and software implementations could be expressed in the same language as the system-level model, thus avoiding semantic mismatches between different languages in the design flow (Ashenden & Wilsey, 1998). Some of the most common system-level design languages are StateCharts (Harel, 1987), Estelle (Budkowski & Dembinski, 1987), SDL (Færgemand & Olsen, 1994), CSP (Hoare, 1978) and SystemC (SystemC, 2009). Most recent and prominent of those is SystemC. SystemC is a C++ class library that can be used to create a cycle-accurate model for software algorithms, hardware architectures, and interfaces, related to system-level designs (SystemC, 2009).

Most of modern embedded systems have both the hardware and software components. When designing such a system, it is important that both sides are developed not in an isolated but in an integrated manner. The generic hardware/software co-design methodology, as a part of the overall system design flow, supports concurrent development of software and hardware. Important tasks in such a development are co-simulation and co-verification. It should be noted that in many cases, systems have also analogue parts that should be designed concurrently with rest of the system (Gerstlauer, Haubelt, Pimentel, Stefanov, Gajski, & Teich, 2009).

System-level design issues of NoC-based systems

In principle, the system-level design issues for NoC-based systems follow the same principles as described above. That is, the initial specification is modelled to estimate performance and resource requirements when using different architectural solutions. This includes platform selection, task mapping and task scheduling. In addition, because of the rather complex communication behaviour between resources, communication mapping and scheduling between tasks should be addressed with care. The reason for that is rather simple – communication latencies may be unpredictable, especially when trying to apply dynamic task organisation. Therefore, the traditional scheduling techniques that are applicable to the hard real-time and distributed systems are not suitable as they address only the bus-based or point-to-point communication. Also, system-level design for NoCs has one major difference when compared to the traditional system-level design – hardware platform is either fixed or has limited modification possibilities (Keutzer, Newton, Rabaey, & Sangiovanni-Vincentelli, 2000). Therefore the main focus is on the application design and distribution between resources.

NoC communication latency depends on various parameters such as topology, routing, switching algorithms, etc., and need to be calculated after task mapping and before the task graph scheduling (Marculescu et al., 2009). In several research papers, the average or the worst case communication delay has been considered (Lei & Kumar, 2003; Marcon, Kreutz, Susin, & Calazans, 2005; Hu & Marculescu, 2005; Shin & Kim, 2004; Stuijk, Basten, Geilen, & Ghamarian, 2006; Shim & Burns, 2008; Shin & Kim, 2008). In many cases, it is an approximation that can be either too pessimistic (giving the upper bound) or too optimistic (by not scheduling explicitly the communication or not considering the communication conflicts). Therefore, an efficient system-level NoC design framework requires an approach for the communication modelling and synthesis to calculate communication hard deadlines that are represented by communication delay and guide the system-level synthesis process by taking into account possible network conflicts.

Dependable Systems-on-Chip

System dependability is a QoS having attributes reliability, availability, maintainability, testability, integrity and safety (Wattanapongsakorn & Levitan, 2000). Achieving a dependable system requires combination of a set of methods that can be classified into:

- fault-avoidance – how to prevent (by construction) fault occurrence,
- fault-tolerance – how to provide (by redundancy) service in spite of faults occurred or occurring,
- error-removal – how to minimize (by verification) the presence of latent errors,
- error-forecasting – how to estimate (by evaluation) the presence, the creation and the consequences of errors (Laprie, 1985).

In 1997, Kiang has depicted dependability requirements over past several decades showing shift in the dependability demands from the product reliability into customer demands for total solutions. The percentage of hardware failures noted in the field is claimed to be minimal, thus allowing to focus on system architecture design and software integrity through the design process management and concurrent engineering. Technology scaling, however, brings process variations and increasing number of transient faults (Constantinescu, 2003) that requires focus together with system design also on fault-tolerance design. According to Wattanapongsakorn and Levitan (2000) a design framework that integrates dependability analysis into the system design process must be implemented. To date, there are very few such system design frameworks, and none of them provide support at all design levels in the system design process, including evaluations of system redundancy, and dependency.

Classification of faults

Different sources classify the terms fault, error, failure differently. However, in everyday life we tend to use them interchangeably. According to IEEE standard 1044-2009 (2009) of software anomalies, an error is an action which produces an incorrect result. A fault is a manifestation of the error in software. A failure is a termination of the ability of a component to perform a required action. A failure may be produced when a fault is encountered. In Koren and Krishna (2007) view a fault (or a failure) can be either a hardware defect or a software mistake. An error is a manifestation of the fault or the failure.

Software faults are in general all programming mistakes (bugs). Hardware faults can be divided into three groups: permanent, intermittent and transient faults according to their duration and occurrence.

- Permanent faults – the irreversible physical defects in hardware caused by manufacturing process variations or wearout mechanism. Once a permanent fault occurs it does not disappear. Manufacturing tests are used to detect permanent faults caused by the manufacturing process. Fault tolerance techniques can be used to achieve higher yield by accepting chips with some permanent faults that are then masked by the fault tolerance methods.
- Intermittent faults – occur because of unstable or marginal hardware. They can be activated by environmental changes, like higher temperature or voltage. Usually intermittent faults precede the occurrence of permanent faults (Constantinescu, 2003).
- Transient faults – cause a component to malfunction for some time. Transient faults are malfunctions caused by some temporary environmental conditions such as neutrons and alpha particles, power supply and interconnect noise, electromagnetic interference and electrostatic discharge (Constantinescu, 2003). Transient faults cause no permanent damage and therefore they are called soft errors. The soft errors are measured by Soft Error Rate (SER) that is probability of error occurrence.

Fault tolerance

Fault tolerance is an exercise to exploit and manage redundancy. Redundancy is the property of having more of a resource than is minimally necessary to provide the service. As failures happen, redundancy is exploited to mask or work around these failures, thus maintaining the desired level of functionality (Koren & Krishna, 2007).

Usually we speak of four forms of redundancy:

- Hardware – provided by incorporating extra hardware into the design to either detect or override the effects of a failed component. We can have
    - static hardware redundancy – objective to immediately mask a failure;
    - dynamic hardware redundancy – spare components are activated upon a failure of a currently active component;
    - hybrid hardware redundancy – combination of the two above.
- Software – protects against software faults. Two or more versions of the software can be run in the hope that that the different versions will not fail on the same input.
- Information – extra bits are added to the original data bits so that an error in the bits can be detected and/or corrected. The best-known forms of information redundancy are error detection and correction coding. Error codes require extra hardware to process the redundant data (the check bits).
- Time – deals with hardware redundancy, re-transmissions, re-execution of the same program on the same hardware. Time redundancy is effective mainly against transient faults (Koren & Krishna, 2007).

Metrics are used to measure the quality and reliability of devices. There are two general classes of metrics that can be computed with reliability models:

- the expected time to some event, and
- the probability that a system is operating in a given mode by time *t*.

The expected time to some event is characterized by mean time to failure (MTTF) – the expected time that a system will operate before a failure occurs. Mean Time To Repair (MTTR) is an expected time to repair the system. Mean Time Between Failures (MTBF) combines the two latter measures and is the expected time that a system will operate between two failures:

$$MTBF = MTTF + MTTR \qquad (4)$$

The second class is represented by reliability measure. Reliability, denoted by $R(t)$, is the probability (as a function of the time t) that the system has been up continuously in the time interval $[t_0, t]$, given that the system was performing correctly at time t_0 (Smith, DeLong, Johnson, & Giras, 2000).

While general system measures are useful at system-level, these metrics may overlook important properties of fault-tolerant NoCs (Grecu, Anghel, Pande, Ivanov, & Saleh, 2007). For example, even when the failure rate is high (causing undesirable MTBF) recovery can be performed quickly on packet or even on flit level. Another drawback is related to the fact that generic metrics represent average values. In a system with hard real-time requirements the NoC interconnect must provide QoS and meet the performance constraints (latency, throughput). Therefore specialized measures focusing on network interconnects should be considered when designing fault-tolerant NoC-based Systems-on-Chip. For example, one has to consider node connectivity that is defined as the minimum number of nodes and links that have to fail before the network becomes disconnected or average node-pair distance and the network diameter (the maximum node-pair distance), both calculated given the probability of node and/or link failure (Koren & Krishna, 2007). In 2007 Ejlali, Al-Hashimi, Rosinger, and Miremadi proposed performability metric to measure the performance and reliability of communication in joint view. Performability $P(L, T)$ of an on-chip interconnect is defined as the probability to transmit L useful bits during the time T in the presence of noise. In presence of erroneous communication re-transmission of messages is needed which reduces probability to finish the transmission in a given time period. Lowering the bit-rate increases time to transmit the messages but also increases probability to finish the transmission during the time interval. According to authors the performability of an interconnect which is used for a safety-critical application must be greater than $1\text{-}10^{-1}$.

Fault tolerance techniques

Fault tolerance has been extensively studied in the field of distributed systems and bus-based SoCs. In (Miremadi & Torin, 1995) the impact of transient faults in a microprocessor system is described. They use three different error detection mechanisms – signature, watchdog timer, and error capturing instruction (ECI) mechanism. Signature is a technique where each operation or a set of operations are assigned with a pre-computed checksum that indicates whether a fault has occurred during those operations. Watchdog Timer is a technique where the program flow is periodically checked for presence of faults. Watchdog Timer can monitor, for example, execution time of the processes or to calculate periodically checksums (signatures). In the case of ECI mechanism, redundant machine-instructions are inserted into the main memory to detect control flow errors. Once a fault is detected with one of the techniques above, it can be handled by a system-level fault tolerance mechanism. In 2006, Izosimov described the following software based fault tolerance mechanisms: re-execution, rollback recovery with checkpointing and active/passive replication. Re-execution restores the initial inputs of the task and executes it again. Time penalty depends on the task length. Rollback recovery with checkpointing mechanism reduces the time overhead – the last non-faulty state (so called checkpoint) of a task has to be saved in advance and will be restored if the task fails. It requires checkpoints to be designed into the application that is not a deterministic task. Active and passive replications utilize spare capacity of other computational nodes. In 2007, Koren and Krishna described fault tolerant routing schemes in macro-distributed networks.

Similarly to distributed systems, NoC is based on a layered approach. The fault tolerance techniques can be classified by the layer onto which they are placed in the communication stack. We are, however, dividing the fault tolerance techniques into two bigger classes – system-level and network-level techniques. At the network level, the fault tolerance techniques are based, for example, on hardware redundancy, error detection / correction and fault tolerant routing. By system-level fault tolerance we mean techniques that take into account application specifics and can tolerate even unreliable hardware.

One of the most popular generic fault tolerance techniques is n-modular redundancy (NMR) that consists of *n* identical components and a voter to detect and mask failures. This structure is capable of masking *(n - 1)/2* errors having *n* identical components. The most common values for *n* are three (triple modular redundancy, TMR), five and seven capable of masking one, two and three errors, respectively. Because a system with an even number of components may produce an inconclusive result, the number of components used must be odd (Pan & Cheng, 2007). NMR can be used to increase both hardware and system-level reliability by either duplicating routers, physical links or running multiple copies of software components on different NoC processing cores.

Pande, Ganguly, Feero, Belzer, and Grecu (2006) propose a joint crosstalk avoidance and error correction code to minimize power consumption and increase reliability of communication in NoCs. The proposed schemes, Duplicate Add Parity (DAP) and Modified Dual Rail (MDR), use duplication to reduce crosstalk. Boundary Shift Code (BSC) coding scheme attempts to reduce crosstalk-induced delay by avoiding shared boundary between successive codewords. BSC scheme is different from DAP that at each clock cycle, the parity bit is placed on the opposite side of the encoded flow control unit. Data coding techniques can be used in both inter-router and end-to-end communication. Dumitras and Marculescu (2003) propose a fast and computationally lightweight fault tolerant scheme for the on-chip communication, based on an error-detection and multiple-transmissions scheme. The key observation behind the strategy is that, at the chip level, the bandwidth is less expensive than in traditional networks because of the existing high-speed buses and interconnection fabrics that can be used for the implementation of a NoC. Therefore we can afford to have more packet transmissions than in the previous protocols in order to simplify the communication scheme and to guarantee low latencies. Dumitras and Marculescu call this strategy where IPs communicate using probabilistic broadcast scheme – on-chip stochastic communication. Data is forwarded from a source to destination cores via multiple paths selected by probability. Similar approach is proposed in (Pirretti, Link, Brooks, Vijaykrishnan, Kandemir, & Irwin, 2004) and (Murali, Atienza, Benini, & De Micheli, 2006). Lehtonen, Liljeberg and Plosila (2009) describe turn models for routing to avoid deadlocks and increase network resilience for permanent faults. Kariniemi and Nurmi (2005) presented a fault tolerant eXtended Generalized Fat Tree (XGFT) NoC implemented with a fault-diagnosis-and-repair (FDAR) system. The FDAR system is able to locate faults and reconfigure routing nodes in such a way that the network can route packets correctly despite the faults. The fault diagnosis and repair is very important as there is only one routing path available in the XGFTs for routing the packets downwards from nearest common ancestor to its destination. Frazzetta, Dimartino, Palesi, Kumar and Catania (2008) describe an interesting approach where partially faulty links are also used for communication. For example, data can be transmitted via “healthy wires” on a 24-bit wide channel although the channel is before degrading 32-bit wide. Special method is used to split and resemble the flow control units. Zhang, Han, Xu, Li and Li (2009) introduce virtual topology that allows to use spare NoC cores to replace faulty ones and re-configure the NoC to maintain the logical topology. A virtual topology is isomorphic with the topology of the target design but is a degraded version. From the viewpoint of programmers and application, they always see a unified virtual topology regardless of the various underlying physical topologies. Another approach is to have a fixed topology but remap the tasks on a failed core. Ababei and Katti (2009) propose a dynamic remapping algorithm to address single and multiple processing core failures. Remapping is done by a general manager, located on a selected tile of the network.

In Valtonen, Nurmi, Isoaho and Tenhunen (2001) view, reliability problems can be avoided with physical autonomy, i.e., by constructing the system from simple physically autonomous cells. The electrical properties and logical correctness of each cell should be subject to verification by other autonomous cells that could isolate the cell if deemed erroneous (self-diagnosis is insufficient, because the entire cell, including the diagnostic unit, may be defect). In 2007, Rantala, Isoaho and Tenhunen motivate the shift from low level testing and testability design into system-level fault tolerance design. They propose an agent-based design methodology that helps bridging the gap between applications and reconfigurable architectures in order to address the fault tolerance issues. They add a new functional agent/control layer to the traditional NoC architecture. The control flow of the agent-based architecture is divided hierarchically to different levels. The granularity of functional units on the lowest level is small and grows gradually when raised on the levels of abstraction. For example the platform agent at the highest level controls the whole NoC platform while a cell agent monitors and reports status of a processing unit to higher level agents. Rusu, Grecu and Anghel (2008) propose a coordinated checkpointing and rollback protocol that is aimed towards fast recovery from system or application level failures. The fault tolerance protocol uses a global synchronization coordinator Recovery Management Unit (RMU) which is a dedicated task. Any task can initiate a checkpoint or a rollback but the coordination is done each time by the RMU. The advantages of such an approach are simple protocol, no synchronization is needed between multiple RMUs, less hardware overhead and power consumption. The drawback is the single point of failure – the dedicated RMU itself.

As a conclusion, there are various techniques to increase NoC fault tolerance but most of the research has been so far dedicated to NoC interconnects or fault tolerant routing. With the increase of variability the transient faults play more important role. The application running on a NoC must be aware of the transient faults and be able to detect and recover efficiently from transient faults. Therefore, a system-level synthesis framework with communication modelling is needed.

SCHEDULING FRAMEWORK OF NETWORK-ON-CHIP BASED SYSTEMS

In this section we propose an approach for communication modelling and synthesis to calculate communication hard deadlines that are represented by communication delay and guide the scheduling process to take into account possible network conflicts.

Design flow

We are employing a traditional system-level design flow (Figure 3) that we have extended to include NoC communication modelling and dependability issues. Input to the system-level design flow is an application *A*, NoC architecture *N* and application mapping *M*. Application is specified by a directed acyclic graph $A = (T, C)$, where $T = \{t_i \mid i = 1,\ldots,T\}$ is set of vertices representing non-preemptable tasks and $C = \{c_{i,j} \mid (i,j) \in \{1,\ldots,V\} \times \{1,\ldots,V\}\}$ is a set of edges representing communication between tasks. Each task t_i is characterized by the Worst Case Execution Time (WCET) $Wcet_i$ and mobility Mob_i that are described in more details in the section "Scheduling of extended task graph". NoC platform introduces communication latency that depends not only on message size but also on resource mapping and needs to be taken into account. An edge $c_{i,j}$ that connects two tasks t_i and t_j represents control flow dependency in case edge parameter message size $Msize_{i,j} = 0$ and communication in case $Msize_{i,j} > 0$. In addition to the message size, the edge is characterized by the Communication Delay (CD) $Cd_{i,j}$ that is described in more details in section "Communication synthesis". We assume that application has dummy start and end vertices. Both these vertices have $Wcet = 0$.

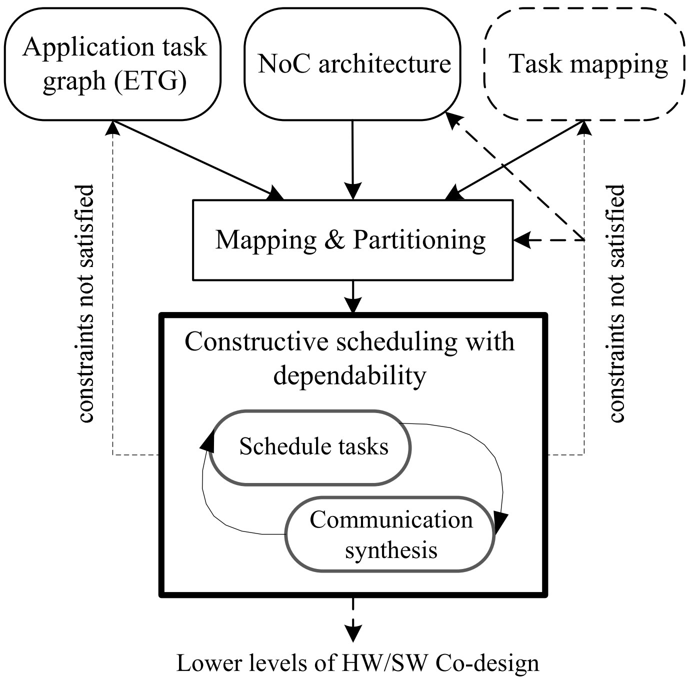

Application task graph (ETG)
NoC architecture
Task mapping
Mapping & Partitioning
Constructive scheduling with dependability
Schedule tasks
Communication synthesis
constraints not satisfied
constraints not satisfied
Lower levels of HW/SW Co-design

Figure 3. System-Level design flow

NoC architecture is a directed graph $N = (R, L)$ where $R = \{r_k \mid k = 1,\ldots,R\}$ is a set of resources and $L = \{l_{k,l} \mid (k,l) \in \{1,\ldots,R\} \times \{1,\ldots,R\}\}$ is a set of links connecting a pair of resources (k,l). The resources can be routers and computational cores. The architecture is characterized by operating frequency, topology, routing algorithm, switching method and link bit-width. The mapping *M* of an application *A* is represented by a function $M (T \rightarrow R)$. According to Marculescu et al. (2009) the application mapping has a major impact on the schedule length, NoC performance and power consumption. However, in our work we assume that the application is already mapped and finding an optimal application mapping is out of the scope of this work.

Once the tasks have been mapped to the architecture, constructive task scheduling starts. It consists of communication synthesis and task scheduling that are described in more detail in section “Communication synthesis”. The application and architecture can also contain information about dependability which is explained in section “Task graph scheduling with dependability requirements”. If dependability and other design requirements are met the lower levels of HW/SW co-design processes continue. Otherwise changes are needed in the architecture or in the mapping.

Communication synthesis

Importance of communication synthesis

One of the key components of the scheduling framework, described in this work, is the communication synthesis, which main purpose is to calculate communication hard deadlines that are represented by Communication Delay (CD) and guide the scheduling process to take into account possible network conflicts. In hard real-time dependable systems the predictable communication delays are crucial. Once a fault occurs, the system will apply a recovery method that might finally require re-scheduling of the application. To analyze the fault impact on the system we need to have information how a fault affects the task execution and communication delays. In our proposed approach the communication is embedded into extended task graph (ETG) that allows us to use the fine-grained model during the scheduling and avoid over dimensioning of the system. Detailed information about communication is also needed for accurate power model (Marculescu et al., 2009). Another design aspect is the ratio of modelling speed and accuracy. A communication schedule could be extracted by simulating the application on a NoC simulator, but the simulation speed will be the limiting factor.

In Figure 4, an example task graph (Figure 4a) and its mapping onto five processing units (Figure 4b) is presented. Task t_0 is mapped onto PU_1, t_1 onto PU_2 etc. It can be seen that communication c_1 (from t_0 to t_2) takes three links ($link_1$, $link_2$, $link_3$) while c_2 (from t_1 to t_2) takes two links ($link_2$, $link_3$). We can calculate the communication delays without conflicts for different switching methods based on formulas below (Ni & McKinley, 1993):

$$Cd_{i,j}^{store\text{-}and\text{-}forward} = (S/B)D \qquad (5)$$

where S is the packet size, B is the channel bandwidth and D is the length of the path in hops between source and destination task.

$$Cd_{i,j}^{virtual\ cut\text{-}through} = (L_h/B)D + S/B \qquad (6)$$

where L_h is the size of the header field.

$$Cd_{i,j}^{wormhole} = (L_f/B)D + S/B \qquad (7)$$

where L_f is the maximum size of the flit.

The physical links, which the communication traverses, are shared resources. It means that in addition to calculating the latencies we need to avoid or have a method to take into account the network conflicts as well. It should be noted that the actual routes will depend on how tasks are mapped and which routing approach is being used.

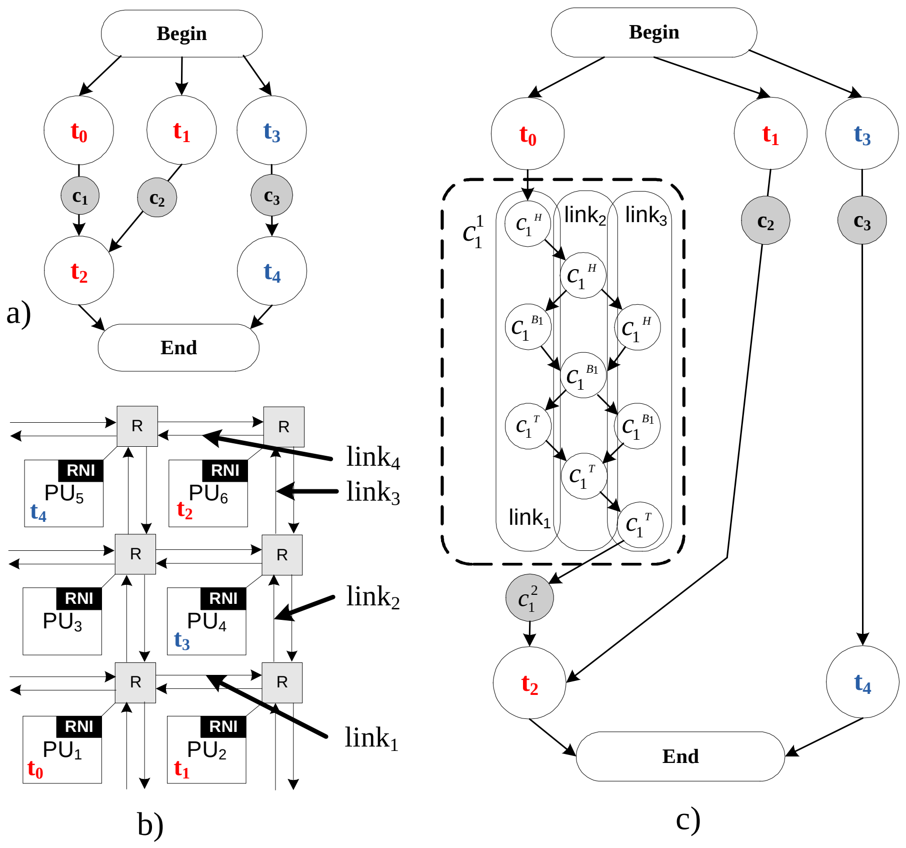


Figure 4. Extended task graph, mapping and partially transformed ETG

Manolache, Eles and Peng (2007) proposed a task graph extension with detailed communication dependencies employing virtual cut-through switching with deterministic source-ordered XY routing. The basic idea is to cover with the task graph not only the tasks but also the flow control units (e.g., packages, flits). That is, all communication edges between tasks are transformed into sequences of nodes representing flow control units. Edges represent dependencies between tasks and/or flow control units. Such an approach assumes that both tasks and communication are already mapped, i.e., it is known which tasks are mapped onto which resources and which data-transfers are mapped onto which links. Of course, different routing strategies will give different communication mapping but all information needed for the scheduling is captured in the task graph. We have generalized the proposed approach and made it compatible with different switching methods such as store-and-forward, virtual cut-through and wormhole switching.

Assumptions on architecture

We assume that each computational core is controlled by a scheduler that takes care of task execution on the core and schedules the message transfer between the tasks. The schedule is calculated offline and stored in the scheduler memory. Such scheduler acts also as a synchronizer for data communication. Otherwise a task, which completes earlier of its calculated WCET and starts message transfer, could lead to an unexpected network congestion and have a fatal effect on the execution schedule. We assume that the size of an input butter is one packet in the case of virtual cut-through or store-and-forward and one flit in the case of wormhole switching method. Input buffer of a flow control unit allows it to be coupled with the incoming link and to look at them as one shared resource. Multiple input buffers would require extension of the graph model and the scheduling process. The proposed approach can be extended to be used in wormhole switching with virtual channels – each virtual channel could be modelled as a separate physical channel having a separate input buffer of one flow control unit. We assume deterministic routing. In our experiments we are using dimension ordered XY routing. Our NoC topology is $m \times n$ (2D) mesh with bidirectional links between the switches (Figure 4b).

Communication synthesis for different switching methods

Input for the scheduling is an extended task graph where tasks are mapped onto resources. Once a communication task is ready to be scheduled, we start the communication synthesis sub-process. Depending on the selected switching method, some of the flow control units must be scheduled strictly to the subsequent time slots. In wormhole switching, the header flit contains the routing information and builds up the communication path, meaning when the header flit goes through a communication link, the body flits must follow the same path. Also, when the header flit is temporarily halted, e.g., because of the traffic congestion, the following flits in downstream routers must be halted too. This sets additional constraints for the communication synthesis. The constraints – fixed order and delay between some of the nodes – are similar to the restrictions used in pipe-lined scheduling (De Micheli, 1994).

Figure 4c depicts the communication synthesis sub-process for communication task c_1 between tasks t_0 and t_2 in case of wormhole switching. The variable size message c_1 (Figure 4a) is divided into bounded size packets c_1^1 and c_1^2. A packet is further divided into three types of data-units (flits) – header (*H*), body (*B*) and tail (*T*). Typically there is only one *H* and one *T* flit, while many *B* flits. The flit pipeline is built for all links the communication traverses. The edges represent dependencies between two flits. As a result, the body flit c_1^{B1} on $link_1$ depends on the header flit c_1^H on the $link_2$. Therefore the body flit c_1^{B1} cannot be sent before the header flit c_1^H has been scheduled (acquired a flit buffer in the next router). Combined with traditional priority scheduling to handle network resource conflicts (e.g., list scheduling), the body flit will be scheduled after the header flit has been sent.

Scheduling of extended task graph

Our proposed approach can be used with arbitrary scheduling algorithm, although the schedules in this paper are produced by using list scheduling. Our goal is to find a schedule *S* which minimizes the worst-case end-to-end delay *D* (application execution time), schedules messages on communication links and produces information about contentions. First, we will calculate the priorities of the tasks represented by mobility Mob_i. Mobility is defined as difference between task ASAP (As-Soon-As-Possible) and ALAP (As-Late-As-Possible) schedule. We will schedule a ready task. Next, we will start the communication synthesis and scheduling for messages initiated by this task. Figure 5a shows a scheduling state where tasks t_0, t_3 and communication c_1, c_3 (Figure 4a) have been scheduled. The respective extended task graph is depicted in Figure 5d. As a next step we are going to schedule communication c_2 in between tasks t_1 and t_2. Without any conflict the schedule looks like depicted in Figure 5b. The respective extended task graph is shown in Figure 5e. Combining schedules depicted in Figure 5a and Figure 5b show that there is a communication conflict on $link_2$ and $link_3$. Based on calculated priority we need to delay the communication c_2 and schedule it after c_1^T. Figure 5c shows that even if we will delay the c_2 start time there will be a conflict between the c_2^H and c_3^H flit on $link_3$. Therefore the flit c_2^H needs to be buffered in downstream router and wait for available input buffer in next router. This is done by finding the maximum schedule time on $link_3$ and scheduling the flit $c_2^{Hstart} = max(link_3^{time})$. After the flit c_2^H has been scheduled on $link_3$ the schedule end time of the same flit on $link_2$ need to be updated. Figure 5c shows the schedule for communication c_2 after the conflicts have been resolved. The resulting schedule is depicted in Figure 6.

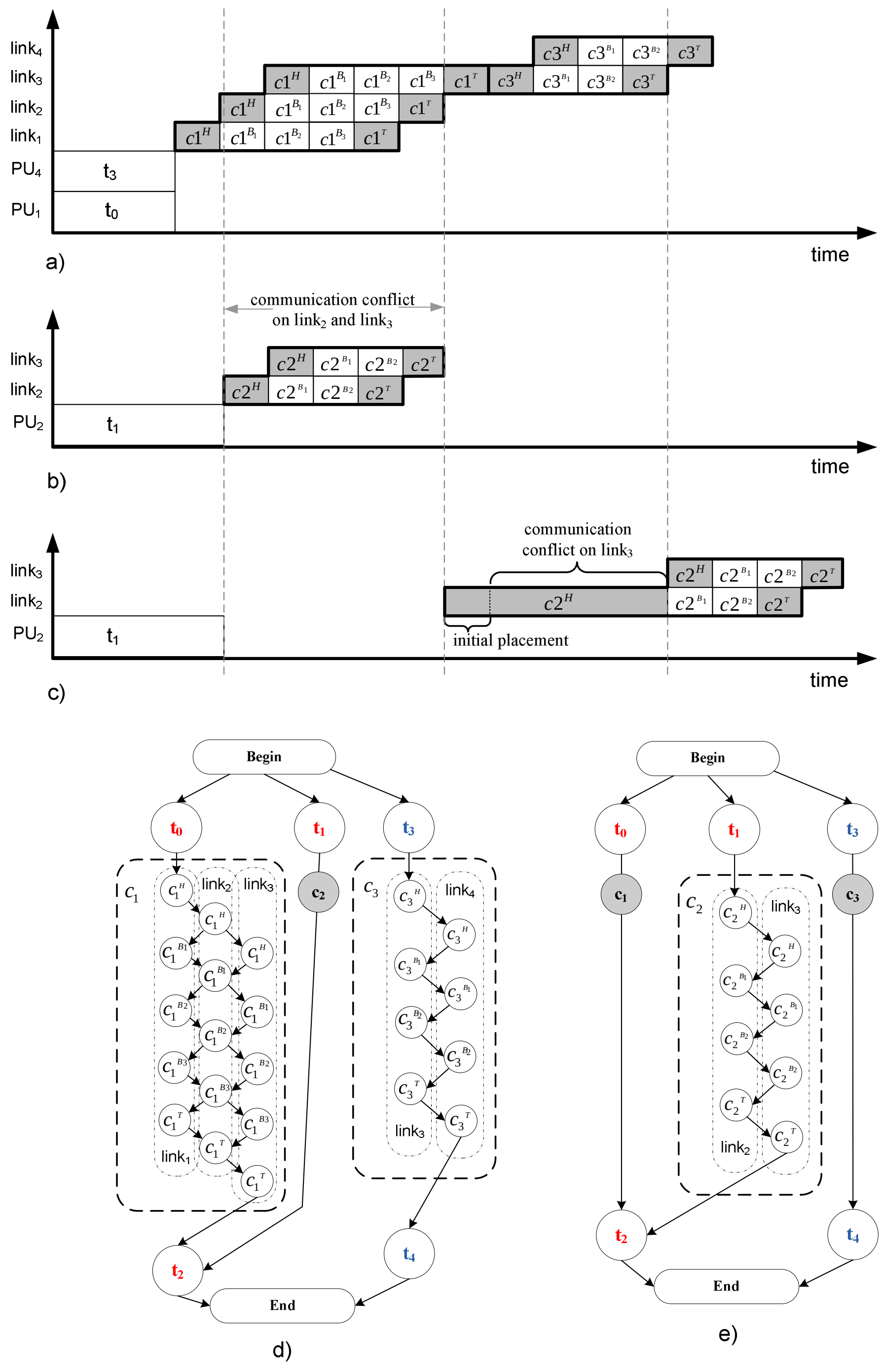
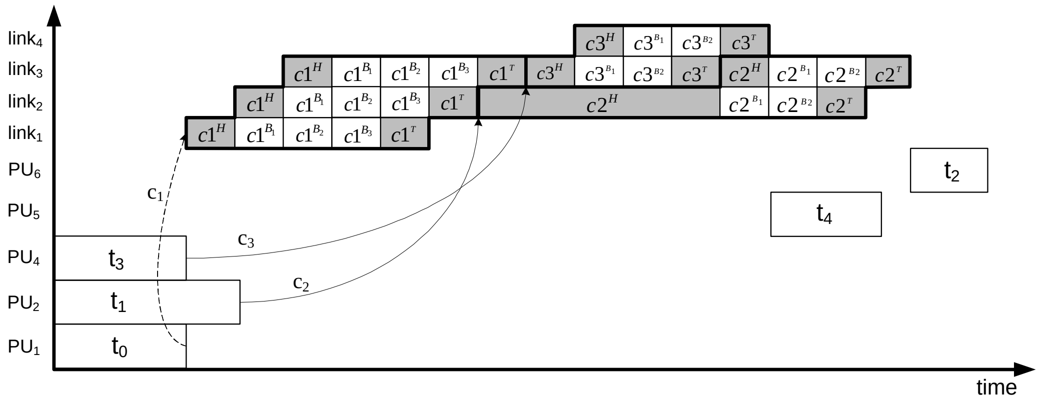


Figure 5. Communication synthesis and scheduling



link4
link3
link2
link1
PU6
PU5
PU4
PU2
PU1
$c3^H$ $c3^{B_1}$ $c3^{B_2}$ $c3^T$
$c1^H$ $c1^{B_1}$ $c1^{B_2}$ $c1^{B_3}$ $c1^T$ $c3^H$ $c3^{B_1}$ $c3^{B_2}$ $c3^T$ $c2^H$ $c2^{B_1}$ $c2^{B_2}$ $c2^T$
$c1^H$ $c1^{B_1}$ $c1^{B_2}$ $c1^{B_3}$ $c1^T$ $c2^H$ $c2^{B_1}$ $c2^{B_2}$ $c2^T$
$c1^H$ $c1^{B_1}$ $c1^{B_2}$ $c1^{B_3}$ $c1^T$
c_1
c_3
c_2
t_0
t_1
t_2
t_3
t_4
time

Figure 6. Final schedule of the application

```
ScheduleCommunication(c_i)
first vertex of sub-graph = transform communication edge c_i into sub-graph
add into ReadyToSchedule list the first vertex of sub-graph
while ReadyToSchedule ≠ ∅, i = 0 do
    if current flit being scheded is a head flit from new packet then
        //ScheduleTimePrev – schedule time from predecessor flit or task
        if predecessor of current flit is a task then
            ScheduleTimePrev = store the task schedule end time
        else
            //predecessor of current flit was also a flit
            ScheduleTimePrev = maximum link schedule time where the
            predecessor flit was mapped
        end if
    else
        //we are scheduling flits from the same packet
        if flit type of current flit == HEAD then
            ScheduleTimePrev = schedule end time of predecessor
        else
            ScheduleTimePrev = maximum link schedule time where the
            predecessor flit was mapped
        end if
    end if

    LinkTime = get max schedule time from mapped link of current flit
    //choose the maximum schedule time from predecessor task or a flit on a link
    if ScheduleTimePrev < LinkTime then
        TaskStartTime = LinkTime
    else
        TaskStartTime = ScheduleTimePrev
    end if

    TaskEndTime = TaskStartTime + Communication Delay of current flit
    Back annotate previous head flit schedule end time if applicable
    Add successor vertexes and remove scheduled flit from ReadyToSchedule
end while
end ScheduleCommunication
```

Figure 7. Communication scheduling algorithm for wormhole switching

For each flow control unit we will calculate its communication delay on corresponding link that is represented by the formula:

$$c_i^{CD} = Sf / Bl \qquad (8)$$

where *Sf* is the size of the flow control unit (flit or packet) and *Bl* bandwidth of the corresponding link. $\sum(c_i^{Endtime} - c_i^{Starttime})$ gives us the total communication delay of c_i. Currently we take into account only the transmission time between the network links. The start-up latency (time required for packetization, copying data between buffers) and inter-router delay are static components and are considered here having 0 delay. Figure 7 depicts the communication scheduling algorithm for wormhole switching. The approach can be used in similar way also for virtual cut-through and store-and-forward switching methods.

The benefits of the proposed approach are fine-grained scheduling of control flow data units, handling network conflicts and the generalization of the communication modelling – the communication is explicitly embedded in a natural way into the task graph. The flit level schedules can be used for debug purposes or for power estimation. The proposed approach can be used for different topologies (including 3D NoC) and different switching methods in relation with deterministic routing algorithms. The network conflicts can be extracted from the schedule and the information used for re-mapping and re-scheduling the application. Our approach does not suffer also from the destination contention problem, thus eliminating the need for buffering at the destination. The graph complexity depends on number of tasks, NoC size, mapping and flow control unit size CFU_{size}. We can represent this by a function $G_{complexity} = (A, N, M, CFUsize)$. Experimental results show that the approach scales well for store-and-forward and virtual cut-through. Wormhole switching contains fine-grained flit level communication schedule and therefore the scaling curve is more sharp than for aforementioned. In the next subsection we will describe a message-level communication synthesis approach that addresses the scaling problem.

Message-level communication synthesis

If the flow control unit level schedule need to be abstracted then the complexity of the communication synthesis can be reduced by transforming the communication edge $c_{i,j}$ into a message sub-graph of traversed links instead of flow control units. In this way we can reduce the graph complexity into $G_{complexity} = (A, N, M)$. Figure 8 shows on the left flit level and on the right message level communication synthesis for c_1. When compared to each other it can be seen that for given example the complexity has been reduced almost by 7 times. The lines 4 - 20 in the wormhole scheduling algorithm in Figure 7 will be replaced in the message-level scheduling by getting communication c_i start time on first link from predecessor task end time. Communication c_i start time on next link is c_i start time on previous link added by head flit communication delay. Similar approach can be applied to virtual-cut-through and store-and-forward switching methods. Experimental results show equal scaling for all of the three switching methods as communication synthesis does not depend anymore on the flow control units. In the following section we will demonstrate the applicability of our approach for scheduling with additional requirements, such as dependability.

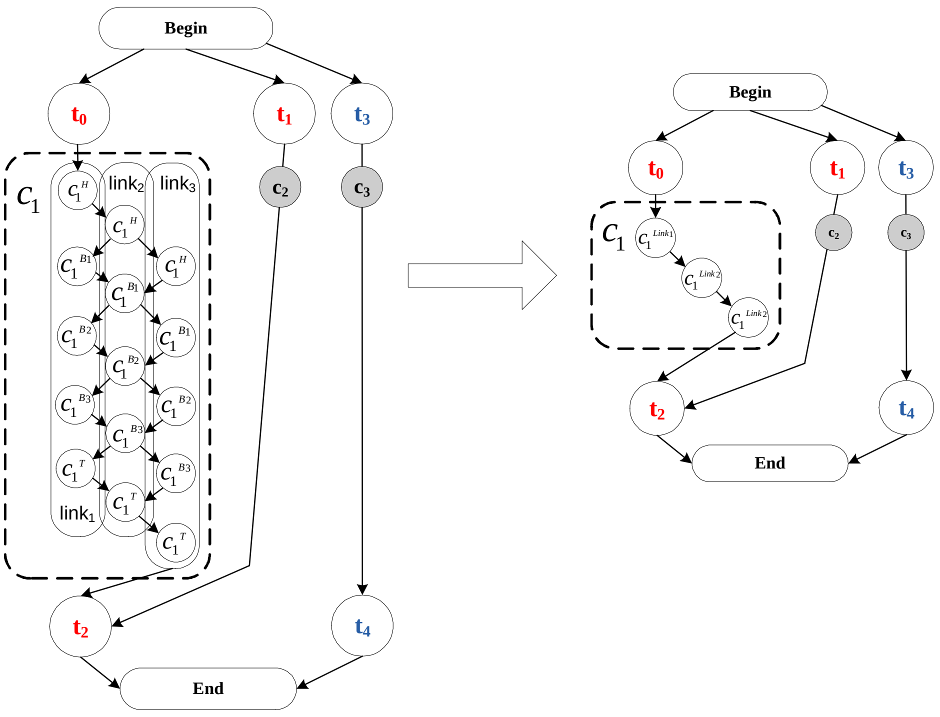


Figure 8. Communication c_1 detailed and message-level

Task graph scheduling with dependability requirements

Our objective is to extend those aforementioned techniques to the system-level to provide design support at early stages of the design flow. The application should be able to tolerate transient or intermittent faults. We are not currently considering permanent faults that need a bit different approach and can be handled by re-scheduling and re-mapping the application on a NoC. The work of Izosimov (2006) describes system-level scheduling and optimizations of fault-tolerant embedded systems in bus based systems. The work considers faults only in computational tasks. The communication fault tolerance is not taken into account. According to Murali et al. (2005) shrinking feature sizes towards nanometer scale cause power supply and treshold voltage to decrease, consequently wires are becoming unreliable because they are increasingly suspectible to noise sources such as crosstalk, coupling noise, soft errors and process variations. Additionally, in bus based systems the task mapping does not have such influence on communication delays as in NoCs. Therefore, we need a method to detect and tolerate transient faults and take possible fault scenarios into account during scheduling.

In our approach we assume that each NoC processing and communication node is capable of detecting faults and executing a corrective action. Transient fault in processing node can be detected with special techniques such as watchdogs or signatures that are easy to implement and have a low overhead. Once a fault is detected inputs of the process will be restored and the task will be re-executed. Murali et al. (2005) proposes two error detection and correction schemes, end-to-end flow control (network level) and switch-to-switch flow control (link level), that can be used to protect NoC communication links from transient faults. We are using a simple switch-to-switch re-transmission scheme where the sender adds error detection code (parity, cyclic redundancy check code (CRC)) to the original message and the receiver checks the received data for correctness. If a fault is detected the sender is requested to re-transmit the data. Depending of switching method the error detection code is added either to flits or to packages.

We are assigning the recovery slacks and scheduling the application using shifting-based scheduling (SBS) (Izosimov, 2006). Shifting-based scheduling is an extension of the transparent recovery against single faults. A fault occurring on one computation node is masked to other computation nodes. It has impact only on the same computation node. According to Izosimov (2006) providing fault containment, transparency can potentially improve testability, debugability and increase determinism in fault-tolerant applications. In shifting-based scheduling the start time of communication is fixed (frozen). It means that we do not need a global real-time scheduler or to synchronize a local recovery event with other cores in the case of fault occurrence. Fixed communication start time allows shifting-based scheduling to be used with our communication synthesis and scheduling approach. We can use the contention information from communication scheduling to be taken into account when trying to find a compromise between the level of dependability and meeting the deadlines of tasks. A downside is that SBS cannot trade-off transparency for performance – communication in a schedule is preserved to start at predefined time.

The scheduling problem we are solving with SBS can be formulated as follows. Given an application mapped on a network-on-chip we are interested to find a schedule table such that the worst-case end-to-end delay is minimized and the transparency requirements with frozen communication are satisfied. In 2006, Izosimov proposed a Fault-Tolerant Conditional Process Graph (FT-CPG) to represent an application with dependability requirements. FT-CPG captures alternative schedules in the case of different fault scenarios. Graphically FT-CPG is a directed acyclic fork-and-join graph where each branch corresponds to a change of condition. In similar way to Izosimov (2006) we are not explicitly generating a FT-CPG for SBS. Instead, all possible execution scenarios are considered during scheduling.

The shifting-based scheduling algorithm is depicted in Figure 9. Input for the SBS is application A, architecture N with mapping M, the number of transient faults *k* to be tolerated in any processing core and the number of transient faults *r* that can appear during data transmission on communication links. First, priorities of tasks are calculated based on mobility and the first task is put into the ready list. Scheduling loop is processed until all tasks have been scheduled. The first task is chosen from the ready list and the work list of ready tasks that are mapped to the same processor as the selected task is created. The work list is sorted based on mobilities and task with smallest mobility is chosen to be scheduled. The task start time is maximum time from mapped processor or predecessor tasks. Next, recovery slack will be calculated for the chosen task in following three steps:

1. The idle time b between chosen task t_{chosen} and the last scheduled task t_{last} on the same processor is calculated
$$b = t_{chosen} - t_{last} \qquad (8)$$
2. Initial recovery slack sl_0 of chosen task t_{chosen} is
$$sl_0 = k * (WCET_{tchosen} + RecoveryOverHead) \qquad (9)$$
where k is number of required recovery events, $WCET_{tchosen}$ worst-case execution time of chosen task and *RecoveryOverHead* time needed to restore the initial inputs. *RecoveryOverHead* has a constant value.
3. The recovery slack sl of chosen task t_{chosen} is changed if recovery slack of previous task t_{last} subtracted with the idle time b is larger than the initial slack sl_0. Otherwise initial recovery slack is preserved.

SBS is adjusting recovery slack to accommodate recovery events of tasks mapped to the same processing core and will schedule communication to the end of the recovery slack. Communication synthesis and scheduling has been explained in previous sections “Communication synthesis for different switching methods” and “Scheduling of extended task graph”. In case of virtual-cut-through and store-and-forward switching methods each packet contains CRC error detection code and we are re-submitting *r* packets from a message. In wormhole switching each flit has CRC error detection code and we are re-submitting *r* flits from a package. CRC code increases router complexity and increases slightly amount of transmitted data but allows to decrease communication latency compared to end-to-end scheme. Re-submission slack is taken into account when reserving buffers and link bandwidth for communication. After a task has been scheduled its predecessor tasks, that are ready, are inserted into ready list and scheduled task removed from ready list.

```
Shifting-based-scheduling(A, N, k, r)
1   Calculate mobility of tasks
2   Put BEGIN task into ready list
3   while ReadyList ≠ ∅ do
4       FirstTask = ReadyList[0]
5       WorkList = Get all ready tasks assigned to same core as FirstTask
6       Sort WorkList based on mobility
7       ChosenTask = WorkList[0]
8
9       TaskStartTime = Get max time from mapped processor of ChosenTask or from predeccessor tasks
10      RecoverySlack = Calculate recovery slack of ChosenTask(k)
11      Schedule ChosenTask(ChosenTask, TaskStartTime, RecoverySlack)
12      Schedule Communication with recovery(r)
13      Add ready successor tasks of ChosenTask into ReadyList
14      Delete ChosenTask from ReadyList
15
16  end while
end Shifting-based-scheduling
```

Figure 9. Shifting-based-scheduling algorithm

At the run-time of an application, local schedulers have a partial schedule table that includes start time and dependability information of tasks and start time of communication. In the case of a fault occurrence, corresponding local scheduler will switch to contingency schedule by looking up how many time a task can be re-executed on given processing core before reserved recovery slack will be passed and the deadline missed. The event of exceeding number of re-submission of flits or packages can be catched by local scheduler at the late or missing arrival of incoming data.

Figure 10 depicts an extract of an example of SBS schedule where task t_1 can be re-executed and packet c_3 re-transmitted one time in the case of fault occurrence. We can see that communication c_2 has been to the end of the recovery slack of task t_1. The schedule produced by SBS is longer than schedule without dependability but will tolerate a specified amount of transient faults and the calculated deadline is satisfied. The advantage of our approach is that we can take into account communication induced latencies and fault effects already at very early stages of the design flow. Possible solutions to decrease the schedule length due to transparency would be to introduce check-pointing and replication.

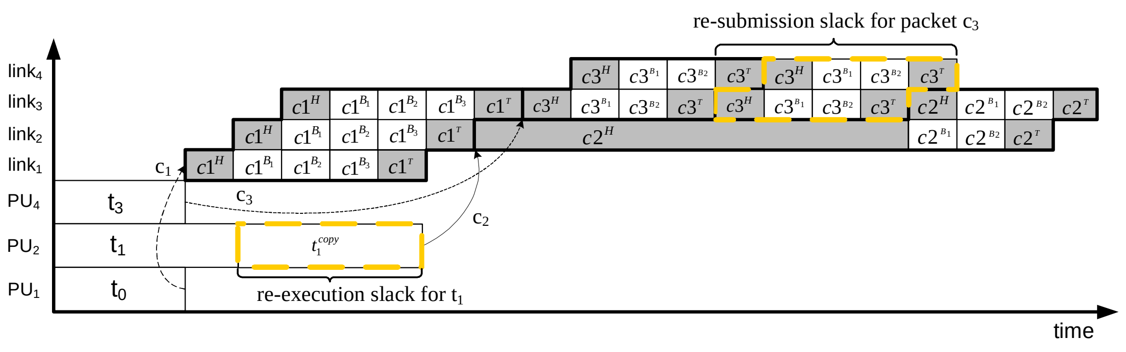


Figure 10. Shifting-based-scheduling example

Experimental results

We have built a design environment that supports our system-level design flow and scheduling framework described in previous sections. To evaluate different aspects of our approach we have ran tests with synthetic task graphs containing 500, 1000, 5000 and 10000 tasks mapped on different NoC architectures. The mapping was generated in all cases randomly. The architecture parameters were varied in together with the application size to show the scaling of the approach. The NoC architecture parameters, if not written differently under experiments, were specified as in Table 1. The tests were performed on computer with Intel L2400 CPU (1,66 GHz), 1 GB of available physical RAM and operating system Microsoft Windows XP.

Table 1 . NoC Architecture Parameters

Parameter name	**Value**
NoC operating frequency	500 MHz
Link bit-width	32 bit
Flit size, packet size	32 bit, 512 bit
Packet header size	20 bit
Link bandwidth	16 Gbit/s
Topology and routing algorithm	2D Mesh, XY routing
Mapping	Random

Our first experiment shows how NoC size impacts the schedule calculation time and length. From one hand, the more computational units we have available the shorter schedule we are able to produce. On the other hand, it takes more processor time to model and synthesise the communication on a bigger NoC. The input task graph of this experiment consists of 1000 tasks and 9691 edges. Virtual cut-through switching is used. The schedules are calculated with both communication synthesis methods – detailed and message-level. The results in Figure 11 show that when NoC size increases the schedule length decreases and schedule calculation time increases. Figure 12 shows scaling of communication synthesis methods from graph size point of view – when the NoC size increases the communication ratio also increases. This can be seen from the number of communication vertices in the extended task graph. However, schedule calculation time increase for both communication synthesis methods is linear. Therefore, it is feasible to use our proposed approach in addition to application scheduling also for performance estimation and design exploration.

The second experiment shows how detailed and message-level communication synthesis methods, based on wormhole switching, are scaling. Detailed communication synthesis is performed for wormhole switching at flit level while in message-level synthesis the smallest unit of communication is a message. Task graphs with different size were mapped and scheduled on a 10 x 10 NoC. To have comparable results the same mapping and NoC size was used for both communication synthesis methods. The results are depicted in Figure 13. When detailed flit-level synthesis is not required then reduction in schedule calculation time and graph complexity can be achieved. However, when detailed flit-level communication schedules are needed, e.g., for power estimation, the detailed communication synthesis approach should be used.

The third experiment shows results of communication modelling and scheduling when a relatively big application is mapped on a NoC with different sizes. Input application contains 5000 tasks and 25279 edges. The results are depicted in Table 2. As mentioned earlier, the larger amount of computational units enables to shorten the schedule, but consequently, the larger network increases the communication ratio as average number of hops between tasks keeps also increasing. At the same time we can see that conflicts length keeps decreasing. It is because of the fact that source-ordered XY routing does no load balancing on the network links by itself. However, when more communication links are available there is less possibility that two message transfers between tasks will intersect on the same link and in the same timeframe. The amount of communication conflicts in the system can be reduced by developing more efficient scheduling heuristic, taking into account the specifics of on-chip networks. As our modelling approach provides detailed information about the communication then it is also possible to use different deterministic routing algorithms during the communication synthesis, in addition to the XY-routing algorithm, used in this paper.

The last experiment shows performance and dependability trade-off when using shifting-based scheduling. We are using an application with 1000 tasks mapped to a 10x10 NoC. We are changing the dependability parameters *k* and *r* of SBS. Results are depicted in Table 3. As explained in previous section SBS cannot trade-off transparency for performance and this can be seen also in the results. Increasing the processing node fault tolerance parameter *k* the schedule length increases roughly *k+1* times for given application. Communication fault-tolerance overhead is marginal compared to computation fault-tolerance. This is due to switch-to-switch error detection and re-submission scheme which reduces communication and recovery latency compared to end-to-end scheme. Additionally, we are attaching error detection code either to each packet or to each flit and re-submit only the faulty flow control unit instead of the whole message. Checkpointing and task replication could be used to decrease schedule length caused by computation delay.

Table 2. Results of communication synthesis

NoC size	**Schedule length (µs)**	**Communication ratio %**	**Communication conflicts length (µs)**	**Calculation time (s)**
25	51235	5%	8981	9
100	36449	10%	8238	17
225	32001	14%	6620	21
400	30556	19%	6252	27
625	20446	24%	5553	36
900	28546	29%	4844	50

Table 3. Shifting-based scheduling – performance / dependability trade-off

Level of dependability		**Schedule length (µs)**	**Increase of initial schedule length (x times)**
k – task re-execution	**r – data re-submission**		
Initial schedule length without dependability and no CRC in communication 34 176			
0	**0**	**34 177**	**1.00**
	1	34 196	
	2	34 215	

	3	34 234	
1	**0**	**66 899**	**1.96**
	1	66 919	
	2	66 939	
	3	66 958	
2	**0**	**99 782**	**2.92**
	1	99 802	
	2	99 822	
	3	99 841	
3	**0**	**135 182**	**3.96**
	1	135 200	
	2	135 217	
	3	135 234	

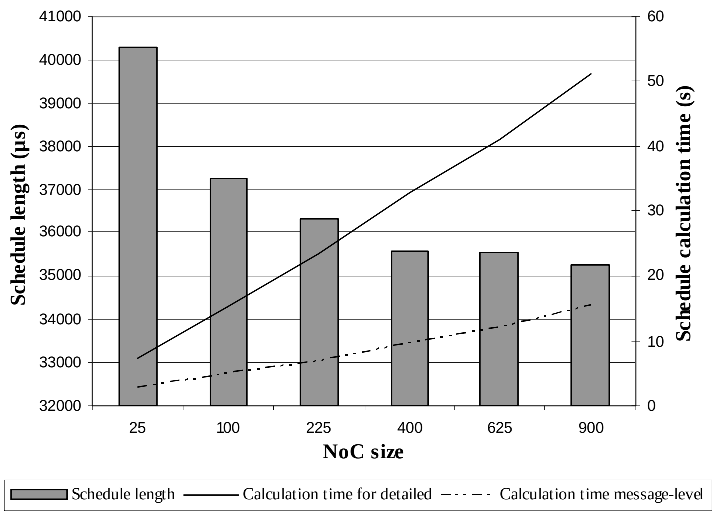


Figure 11 . Schedule length versus calculation time for different NoC sizes

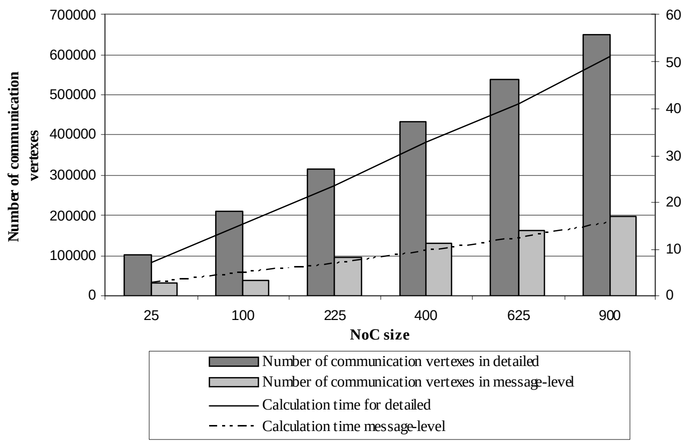


Figure 12 . NoC size impact on Extended Task Graph complexity

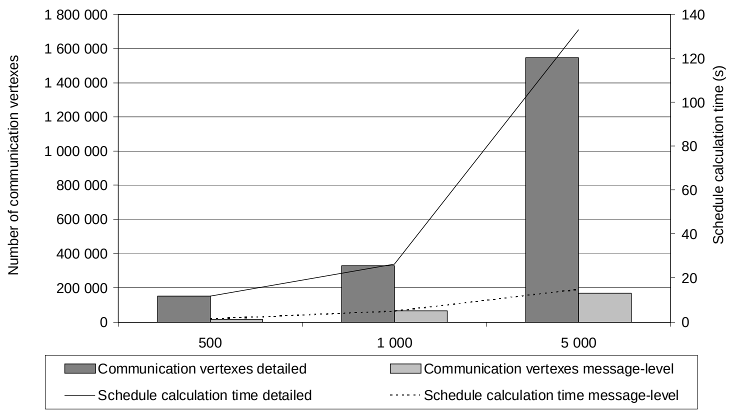


Figure 13 . Reduction on complexity (wormhole switching)

CONCLUSIONS

This chapter described various problems associated with the system-level design of NoC-based systems. The first part of the chapter gave a background and surveyed the related work. The second part described a framework for predictable communication synthesis in NoCs with real-time constraints. The framework models communication at the link level, using traditional task graph based modelling technique and supports various switching techniques. This communication synthesis approach can be used for scheduling of real-time dependable NoC-based systems.

REFERENCES

Ababei, C., & Katti, R. (2009). Achieving network on chip fault tolerance by adaptive remapping. *IEEE International Symposium on Parallel & Distributed Processing (IPDPS '09)* (pp. 1-4).

Adriahantenaina, A., Charlery, H., Greiner, A., Mortiez, L., & Zeferino, C. (2003). SPIN: a scalable, packet switched, on-chip micro-network. *Design, Automation and Test in Europe Conference and Exhibition* (pp. 70-73).

Agarwal, V., Hrishikesh, M., Keckler, S., & Burger, D. (2000). Clock rate versus IPC: the end of the road for conventional microarchitectures. *Proceedings of the 27th International Symposium on Computer Architecture* (pp. 248-259).

Allan, A., Edenfeld, D., Joyner, J. W., Kahng, A. B., Rodgers, M., & Zorian, Y. (2002). 2001 technology roadmap for semiconduc. *IEEE Computer, 35* (1), 42-53.

Arteris. (2009). Retrieved from http://www.arteris.com/

Ashenden, P., & Wilsey, P. (1998). Considerations on system-level behavioural and structural modeling extensions to VHDL. *International Verilog HDL Conference and VHDL International Users Forum* (pp. 42-50).

Banerjee, K., Souri, S., Kapur, P., & Saraswat, K. (2001). 3-D ICs: A Novel Chip Design for Improving Deep-Submicrometer Interconnect Performance and Systems-on-Chip Integration. *Proceedings of the IEEE, 89 (5)*, pp. 602-633.

Benini, L., & De Micheli, G. (2002). Networks on Chips: A New SoC Paradigm. *IEEE Computer, 35* (1), 70-78.

Bertozzi, D., & Benini, L. (2004). Xpipes: a network-on-chip architecture for gigascale systems-on-chip. *IEEE Circuits and Systems Magazine, 4* (2), 18-31.

Bjerregaard, T., & Mahadevan, S. (2006). A survey of research and practices of Network-on-chip. *ACM Computing Surveys, 38* (1).

Bjerregaard, T., & Sparso, J. (2005). A Router Architecture for Connection-Oriented Service Guarantees in the MANGO Clockless Network-on-Chip. *Design, Automation, and Test in Europe, 2,* pp. 1226-1231.

Budkowski, S., & Dembinski, P. (1987). An Introduction to Estelle: A Specification Language for Distributed Systems. *Computer Networks and ISDN Systems, 14* (1), 3-23.

Claasen, T. (2006). An Industry Perspective on Current and Future State of the Art in System-on-Chip (SoC) Technology. *Proceedings of the IEEE, 94 (6)*, pp. 1121-1137.

Constantinescu, C. (2003). Trends and challenges in VLSI circuit reliability. *IEEE Micro, 23* (4), 14-19.

Dally, W. J., & Towles, B. (2004). *Principles and Practices of Interconnection.* Morgan Kaufman Publishers.

Dally, W. J., & Towles, B. (2001). Route packets, not wires: on-chip inteconnection networks. *Design Automation Conference* (pp. 684-689).

Dally, W. (1990). Performance analysis of k-ary n-cubeinterconnection networks. *IEEE Transactions on Computers, 39* (6), 775-785.

De Micheli, G. (1994). *Synthesis and optimization of digital circuits.* McGraw-Hill.

Dumitras, T., & Marculescu, R. (2003). On-chip stochastic communication. *Design, Automation and Test in Europe Conference and Exhibition (DATE ’03)* (pp. 790-795).

Ejlali, A., Al-Hashimi, B., Rosinger, P., & Miremadi, S. (2007). Joint Consideration of Fault-Tolerance, Energy-Efficiency and Performance in On-Chip Networks. *Design, Automation & Test in Europe Conference & Exhibition (DATE '07)* (pp. 1-6).

Færgemand, O., & Olsen, A. (1994). Introduction to SDL-92. *Computer Networks and ISDN Systems*, *26*, 1143-1167.

Feero, B., & Pande, P. (2007). Performance Evaluation for Three-Dimensional Networks-On-Chip. *IEEE Computer Society Annual Symposium on VLSI (ISVLSI '07)* (pp. 305-310).

Felicijan, T., Bainbridge, J., & Furber, S. (2003). An asynchronous low latency arbiter for Quality of Service (QoS) applications. *Proceedings of the 15th International Conference on Microelectronics (ICM 2003)* (pp. 123-126).

Frazzetta, D., Dimartino, G., Palesi, M., Kumar, S., & Catania, V. (2008). Efficient Application Specific Routing Algorithms for NoC Systems utilizing Partially Faulty Links. *11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools (DSD '08)* (pp. 18-25).

Gerstlauer, A., Haubelt, C., Pimentel, A., Stefanov, T., Gajski, D., & Teich, J. (2009). Electronic System-Level Synthesis Methodologies. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 28* (10), 1517-1530.

Goossens, K., Dielissen, J., & Radulescu, A. (2005). Æthereal Network on Chip:Concepts, Architectures, and Implementations. *IEEE Design and Test of Computers, 22* (5), 414-421.

Grecu, C., Anghel, L., Pande, P., Ivanov, A., & Saleh, R. (2007). Essential Fault-Tolerance Metrics for NoC Infrastructures. *13th IEEE International On-Line Testing Symposium (IOLTS 07)* (pp. 37-42).

Grecu, C., Ivanov, A., Pande, R., Jantsch, A., Salminen, E., Ogras, U., et al. (2007). Towards Open Network-on-Chip Benchmarks. *First International Symposium on Networks-on-Chip (NOCS 2007)* (pp. 205-205).

Guerrier, P., & Greiner, A. (2000). A generic architecture for on-chip packet-switched interconnections. *Design, Automation, and Test in Europe* (pp. 250-256).

Hamilton, S. (1999). Taking Moore's law into the next century. *IEEE Computer, 32* (1), 43-48.

Harel, D. (1987). Statecharts: A Visual Formalism for Computer Systems. *Science of Computer, 8* (3), 231-274.

Haurylau, M., Chen, G., Chen, H., Zhang, J., Nelson, N., Albonesi, D., et al. (2006). On-Chip Optical Interconnect Roadmap: Challenges and Critical Directions. *IEEE Journal of Selected Topics in Quantum Electronics, 12* (6), 1699-1705.

Hemani, A., Jantsch, A., Kumar, S., Postula, A., Öberg, J., Millberg, M., et al. (2000). Network on chip: An architecture for billion transistor era. *Proceedings of the IEEE Norchip Conference.*

Ho, R., Mai, K., & Horowitz, M. (2001). The future of wires. *Proceedings of the IEEE, 89 (4),* pp. 490-504.

Hoare, C. A. (1978). Communicating Sequential Processes. *Communications of the ACM, 21* (11), 934-941.

Hu, J., & Marculescu, R. (2005). Communication and task scheduling of application-specific networks-on-chip. *Computers and Digital Techniques, 152* (5), 643-651.

Huang, T.-C., Ogras, U., & Marculescu, R. (2007). Virtual Channels Planning for Networks-on-Chip. *8th International Symposium on Quality Electronic Design (ISQED 2007)* (pp. 879-884).

IEEE Standard Classification for Software Anomalies. IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993). (Jan. 7 2010) (pp. 1-15).

International Technology Roadmap for Semiconductors. (2007). Retrieved from http://www.itrs.net

Izosimov, V. (2006). Scheduling and optimization of fault-tolerant distributed embedded systems. Sweden: Tech. Lic. dissertation, Linköping University.

Iyer, A., & Marculescu, D. (2002). Power and performance evaluation of globally asynchronous locally synchronous processors. *29th Annual International Symposium on Computer Architecture* (pp. 158-168).

Jantsch, A., & Tenhunen, H. (2003). In *Networks on Chip* (pp. 9-15). Kluwer Academic Publishers.

Jantsch, A. (2003). *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation*. Morgan Kaufmann.

Kahng, A.B. (2007). Key directions and a roadmap for electrical design for manufacturability. *37th European Solid State Device Research Conference (ESSDERC 2007)* (pp. 83-88).

Kariniemi, K., & Nurmi, J. (2005). Fault tolerant XGFT network on chip for multi processor system on chip circuits. *International Conference on Field Programmable Logic and Applications* (pp. 203-210).

Kermani, P., & Kleinrock, L. (1979). Virtual Cut-Through: A New Computer Communication Switching Technique. *Computer Networks, 3,* 267-286.

Keutzer, K., Newton, A., Rabaey, J., & Sangiovanni-Vincentelli, A. (2000). System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 19* (12), pp. 1523-1543.

Kiang, D. (1997). Technology impact on dependability requirements. *Third IEEE International Software Engineering Standards Symposium and Forum (ISESS 97)* (pp. 92-98).

Konstadinidis, G. (2009). Challenges in microprocessor physical and power management design. *International Symposium on VLSI Design, Automation and Test, 2009 (VLSI-DAT '09)* (pp. 9-12).

Koren, I., & Krishna, C. (2007). *Fault-Tolerant Systems.* Morgan Kaufmann.

Kumar, S., Jantsch, A., Millberg, M., Öberg, J., Soininen, J. P., Forsell, M., et al. (2002). A Network on Chip Architecture and Design Methodology. *IEEE Computer Society Annual Symposium on VLSI (ISVLSI'02)* (pp. 105-112).

Lagnese, E., & Thomas, D. (1989). Architectural Partitioning for System Level Design. *26th Conference on Design Automation* (pp. 62-67).

Laprie, J.-C. (1985). Dependable Computing and Fault Tolerance: Concepts and Terminology. *Fifteenth International Symposium on Fault-Tolerant Computing (FTCS-15)* (pp. 2-11).

Lehtonen, T., Liljeberg, P., & Plosila, J. (2009). Fault tolerant distributed routing algorithms for mesh Networks-on-Chip. *International Symposium on Signals, Circuits and Systems (ISSCS 2009)* (pp. 1-4).

Lei, T., & Kumar, S. (2003). A two-step genetic algorithm for mapping task graphs to a network on chip architecture. *Proceedings of the Euromicro Symposium on Digital System Design (DSD’03)* (pp. 180-187).

Lu, Z. (2007). Design and Analysis of On-Chip Communication for Network-on-Chip Platforms. Stockholm, Sweden.

Manolache, S., Eles, P., & Peng, Z. (2007). Fault-Aware Communication Mapping for NoCs with Guaranteed Latency. *Intl. Journal of Parallel Programming, 35* (2), 125-156.

Marcon, C., Kreutz, M., Susin, A., & Calazans, N. (2005). Models for embedded application mapping onto NoCs: timing analysis. *Rapid System Prototyping* (pp. 17-23).

Marculescu, R., Ogras, U., Li-Shiuan Peh Jerger, N., & Hoskote, Y. (2009). Outstanding research problems in NoC design: system, microarchitecture, and circuit perspectives. *IEEE Tran. on Computer-Aided Design of Integrated Circuits and Systems, 28* (1), 3-21.

Miremadi, G., & Torin, J. (1995). Evaluating Processor- Behaviour and Three Error-Detection Mechanisms Using Physical Fault-Injection. *IEEE Trans. on Reliability, 44* (3), 441-454.

Murali, S., Atienza, D., Benini, L., & De Micheli, G. (2006). A multi-path routing strategy with guaranteed in-order packet delivery and fault-tolerance for networks on chip. *Design Automation Conference* (pp. 845-848).

Murali, S., Theocharides, T., Vijaykrishnan, N., Irwin, M., Benini, L., & De Micheli, G. (2005). Analysis of error recovery schemes for networks on chips. *IEEE Design & Test of Computers, 22* (5), 434-442.

Murali, S., Seiculescu, C., Benini, L., & De Micheli, G. (2009). Synthesis of networks on chips for 3D systems on chips. *Design Automation Conference* (pp. 242-247).

Ni, L., & McKinley, P. (1993). A survey of wormhole routing techniques in direct networks. *IEEE Computer, 26* (2), 62-76.

Pan, S.-J., & Cheng, K.-T. (2007). A Framework for System Reliability Analysis Considering Both System Error Tolerance and Component Test Quality. *Design, Automation & Test in Europe Conference & Exhibition (DATE '07)* (pp. 1-6).

Pande, P., Ganguly, A., Feero, B., Belzer, B., & Grecu, C. (2006). Design of Low power & Reliable Networks on Chip through joint crosstalk avoidance and forward error correction coding. *21st IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT '06)* (pp. 466-476).

Pavlidis, V., & Friedman, E. (2007). 3-D Topologies for Networks-on-Chip. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 15* (10), 1081-1090.

Pirretti, M., Link, G., Brooks, R., Vijaykrishnan, N., Kandemir, M., & Irwin, M. (2004). Fault tolerant algorithms for network-on-chip interconnect. *IEEE Computer Society Annual Symposium on VLSI: Emerging Trends in VLSI Systems Design (ISVLSI'04)* (pp. 46-51).

Radulescu, A., & Goossens, K. (2002). Communication Services for Networks on Chip. *SAMOS* (pp. 275-299).

Raghunathan, V., Srivastava, M., & Gupta, R. (2003). A survey of techniques for energy efficient on-chip communication. *Design Automation Conference* (pp. 900-905).

Rantala, P., Isoaho, J., & Tenhunen, H. (2007). Novel Agent-Based Management for Fault-Tolerance in Network-on-Chip. *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)* (pp. 551-555).

Rijpkema, E., Goossens, K., & Wielage, P. (2001). A router architecture for networks on silicon. *In Proceedings of Progress 2001, 2nd Workshop on Embedded Systems.*

Rusu, C., Grecu, C., & Anghel, L. (2008). Communication Aware Recovery Configurations for Networks-on-Chip. *14th IEEE International On-Line Testing Symposium (IOLTS '08)* (pp. 201-206).

Salminen, E., Kulmala, A., & Hämäläinen, T. D. (2008). *Survey of Network-on-chip Proposals*. Retrieved from http://www.ocpip.org/uploads/documents/OCP-IP_Survey_of_NoC_Proposals_White_Paper_April_2008.pdf

Sgroi, M., Sheets, M., Mihal, A., Keutzer, K., Malik, S., Rabaey, J., et al. (2001). Addressing the system-on-a-chip interconnect woes through communication-based design. *Proceedings of the Design Automation Conference* (pp. 667-672).

Shanbhag, N., Soumyanath, K., & Martin, S. (2000). Reliable low-power design in the presence of deep submicron noise. *Proceedings of the 2000 International Symposium on Low Power Electronics and Design (ISLPED '00)* (pp. 295-302).

Shim, Z., & Burns, A. (2008). Real-time communication analysis for on-chip networks with wormhole switching networks-on-chip. *The 2nd IEEE International Symposium on Networks-on-Chip (NoCS'08)* (pp. 161-170).

Shin, D., & Kim, J. (2008). Communication power optimization for network-on-chip architectures. *Journal of Low Power Electronics, 2* (2), 165-176.

Shin, D., & Kim, J. (2004). Power-aware communication optimization for networks-on-chips with voltage scalable links. *CODES + ISSS 2004* (pp. 170-175).

Silistix. (2009). Retrieved from http://www.silistix.com/

Smith, D., DeLong, T., Johnson, B., & Giras, T. (2000). Determining the expected time to unsafe failure. *Fifth IEEE International Symposim on High Assurance Systems Engineering (HASE 2000)* (pp. 17-24).

Sonics. (2009). Retrieved from http://www.sonicsinc.com/

STMicroelectronics. (2009). Retrieved from http://www.st.com

Stressing, J. (1989). System-level design tools. *Computer-Aided Engineering Journal, 6* (2), 44-48.

Stuijk, S., Basten, T., Geilen, M., & Ghamarian, A. (2006). Resource-efficient routing and scheduling of time-constrained streaming communication on networks-on-chip. *Proceedings of the 9th Euromicro Conference on Digital System Design (DSD'06)* (pp. 45-52).

SystemC. (2009). Retrieved from http://www.systemc.org

Zhang, L., Han, Y., Xu, Q., Li, X. w., & Li, H. (2009). On Topology Reconfiguration for Defect-Tolerant NoC-Based Homogeneous Manycore Systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 17* (9), 1173-1186.

Valtonen, T., Nurmi, T., Isoaho, J., & Tenhunen, H. (2001). An autonomous error-tolerant cell for scalable network-on-chip architectures. *Proceedings of the 19th IEEE NorChip Conference* (pp. 198-203).

Wattanapongsakorn, N., & Levitan, S. (2000). Integrating dependability analysis into the real-time system design process. *Annual Reliability and Maintainability Symposium* (pp. 327-334).

KEY TERMS

System-level design – a design methodology that starts from higher abstraction levels and refines the system-level model down to a hardware/software implementation.

System-on-chip (SoC) – integrating all system components into a single integrated chip.

Network-on-chip (NoC) – a new communication paradigm for systems-on-chip.

Dependability – system dependability is a quality-of-service having attributes reliability, availability, maintainability, testability, integrity and safety.

Fault-tolerance – is a property that enables a system to provide service even in the case of faults

Communication modelling – explicit modelling of communication in order to enable predictable design

Communication synthesis – communication refinement. Communication edge in the extended task graph is converted into communication sub-graph.

Communication scheduling – a step in the system-level design flow. Schedules flow control units to start at predefined time.