Interfacing Cores with On-chip Packet-Switched Networks

Praveen Bhojwani and Rabi Mahapatra
Department of Computer Science, Texas A&M University, College Station
{praveenb,rabi}@cs.tamu.edu

Abstract

With the emergence of the packet-switched networks as a possible system-on-chip (SoC) communication paradigm, the design of network-on-chips (NoC) has provided a challenge to the designers. Meeting latency requirements of communication among various cores is one of the crucial objectives for system designers. The core interface to the networking logic and the communication network are the key contributors to latency. With the goal of reducing this latency we examine the packetization strategies in the NoC communication. In this paper, three schemes of implementations are analyzed, and the costs in terms of latency, and area are projected through actual synthesis.

1. Introduction

Modern day solutions to design problems in the domain of telecommunications, multimedia and consumer electronics, hinge on the designers' ability to formulate these systems under strong time-to-market conditions. The integration of system-on-chip (SoC) designs to provide these solutions, rely on the utilization of these components in a plug-and-play fashion. Designers face the challenge of designing not only functionally correct systems, but also guaranteeing reliable operation of the interacting components. On-chip physical interconnections will present a limiting factor for performance and possibly energy consumption. The shared bus, which is today's dominant interconnect template, will not meet the performance requirements of tomorrow's systems. The on-chip switching network is a technology that originates in parallel computing, and is well suited for heterogeneous communication among cores in an SoC environment. To exploit task-level parallelism between processing IPs, the aggregated interconnection throughputs to the order of 50 Gbits/s are needed [3]. Bus-based architectures will not meet this requirement because a bus is inherently non-scalable. The bandwidth of a bus is shared by all attached devices and is simply not sufficient. A suitable replacement that has been suggested by most researchers in this field is that of a packet-switched interconnection template. This template would address the performance and the scalability requirements of the SoCs.

Researchers in this field have suggested the usage of regular layouts for the cores in the system [4, 9]. The communication architecture for such systems consists of the basic building block, the tile. The tiles or clients are connected to a network that routes packets between them. Each tile may consist of one or more cores (processor cores, memory cores, etc.). The tile would have routing logic, which would be responsible for routing, forwarding the packets, based on the routing policy of the network. Before such a system can be deployed for on-chip communication we need to address the latency issue. We need to reduce this latency as much as possible, at every stage of the data communication. The communication comprises of three stages, the packet assembly, packet transmission and the packet disassembly and delivery. We examine the latency characteristics in the packet assembly stage of the on-chip communication.

The different packetization strategies that have been investigated in this paper are

- Software library based,
- On-core module based,
- Wrapper based.

The implementations vary depending on the reconfigurability and programmability of the core in question. Our research was to investigate the suitability of these three methods and to determine the subsequent performance differences between them. These results provide crucial information to the system designer at the time of core-network interface design.

The next section discusses the past work done in the domain of networks-on-chip. Section 3 gives an overview of the proposed work considered in our research. Section 4 provides a summary of the results obtained. Section 5 completes this paper with our conclusions.

2. Background Work

The concept of SoC (System-on-Chip) network communication in the form of packet switched communication was first discussed by Guerrier and Greiner[3]. Their paper presented an architectural study of a scalable system-level interconnect template. They proposed a generic interconnection template that addresses the performance and scalability requirements of system-on-chip using integrated switching networks. They accepted the limitation of their proposed architecture to be the complexity of switching network concepts.

Micheli and Benini [1] proposed that on-chip micro-networks, designed with layered methodology, will meet the distinctive challenges of providing functionally correct, reliable operation of interacting system-on-chip components. This idea was also suggested by Sgroi et. al. [5]. They suggested a formal approach to system-on-chip design. Their approach considered communication between components as important as the computations they performed.

Benini and Micheli [2] discussed energy efficient and reliable interconnect design for SoCs. They addressed the distinguishing features of a design methodology that aimed at achieving reliable designs under the limitations of the interconnect technology. They specifically considered energy consumption reduction, under guaranteed quality of service (QoS), as a main objective in system design.

Dally and Towles [4] also discussed the usage of on-chip packet switched interconnection networks, against ad-hoc global wiring structures on a chip. With their approach, system modules communicate by sending packets to one another over the network. The authors claimed that the structured network wiring would give well-controlled electrical parameters that would eliminate timing iterations and enable the use of high-performance circuits to reduce latency and increase bandwidth. The area overhead required to implement the on-chip network logic was estimated to be 6.6%. The authors also suggested the usage of regular layouts for the cores in the SoC. The tiles are connected to the networks through switching logic that are responsible for the routing of the packets over the network. Similar layouts were also suggested by the authors in [9]. Figure 1 illustrates the generic network on chip architecture.

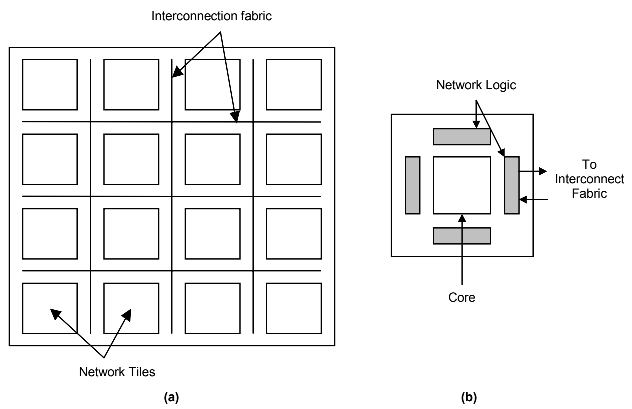

Figure 1 – (a) Generic Network-on-Chip Architecture (b) Tile Structure

The authors in [6] provide a communication architecture synthesis tool-set, shown in Figure 2, that can aid the designer in predicting the various cost parameters and configuring the network-on-chip architecture for optimal performance. They analyzed the issues involved in the synthesis of the on-chip networks and proposed a methodology that will help arrive at an optimal network on chip design. They considered issues, such as the quality of service (QoS) requirements of the communicating cores, in terms of the latency and data rate, utilization of the network resources and implementation cost in terms of area, power and wiring latency. Their tool-set comprised of an IP clustering engine and a simulator to aid in the synthesis. The tools used for the synthesis for the communication architecture, were annotated with the design parameters obtained from gate-level synthesis.

None of the above works address the interfacing issues among the cores and the network interfaces of the tiles in a NoC communication scenario. Our research results here provide the network simulation part of the toolkit with useful information regarding the consequences of the selection between the different schemes of packetization. This additional information will provide more accurate results when we develop systems that use NoC as their interconnect template. Figure 2 demonstrates the relevance of the Core-Network Logic Interface results (dotted block), when integrated with the Synthesis and Verification methodology of On-chip networks discussed in [6].

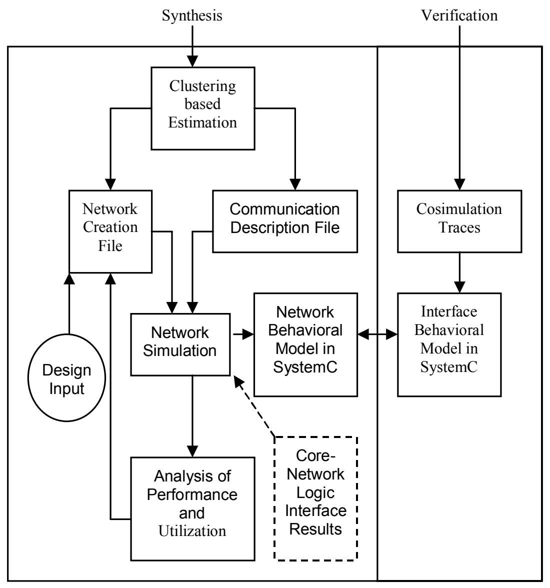
Synthesis
Verification
Clustering based Estimation
Network Creation File
Communication Description File
Cosimulation Traces
Design Input
Network Simulation
Network Behavioral Model in SystemC
Interface Behavioral Model in SystemC
Analysis of Performance and Utilization
Core-Network Logic Interface Results

Figure 2 – Relevance of Core-Network Logic Interface results in the Synthesis and Verification methodology of On-chip networks in [6]

Table 1 – Comparison between packetization schemes

Type of Implementation	Area	Latency (Expected)	Complexity	Flexibility
Software Library (on-core)	Low on HW area, but increases code size (increased instructions to packetize).	High	Increased code size.	Requires programmable cores.
RTL (HW) implementation (on-core)	Additional register and logic to packetize	Low	Additional registers and logic and an increase in instruction set.	Requires programmable cores or development of modified cores.
Wrapper RTL (HW) Implementation (off-core)	Additional control, registers and logic to packetize.	Low	Additional control, registers and logic. Ability to understand core operation	Can use existing cores. Modify wrappers for plug-and-play into different networks.

Since research in the field of packet-switched on-chip interconnection networks is still in its infancy, no published research is as yet available on the complexities and intricacies of this stage of the communication.

3. Proposed Work

In this work, we address the core-network logic interface issues. One component of this issue is the packetization of the core requests. The preparation of the packets is one of the key stages in the communication on the network on-chip. The transportation of the packet over the network is expected to have a large latency, as the packet will have to complete a number of hops (assuming a mesh architecture) to reach its destination. Though the packetization does not depend on the tile-layout, it would be counterproductive to add further latency in the preparation stage of the packet.

Another important issue that arises here is of whether the core should be aware of the network or not. The pros for a network-aware core are:

- Reduced latency, because the core directly provides the packet, once it is informed of the packet format.
- Reduced complexity of the network interface of the core.

The cons of a network-aware core are:

- Specification of packet parameters to the core.
- Core requires a certain degree of programmability.
- Need to modify existing cores to make them network-aware.

The trade-offs here would be of latency, area, complexity and flexibility. Table 1 provides a tabular representation of these features in the three possible implementations.

With the onset of packet-switched networks being a possible mode of communication on SoCs, various aspects of the communication need to be evaluated and optimized to provide the required quality of service (QoS). In order to reduce the packet-switched network on-chip communication latency, several schemes are possible, starting at the compiler-level where the compiler will place instructions - requiring communication medium usage - earlier in the sequence of execution, to have the controllers such as those of memory pre-fetching and transmitting data to the consumers to reduce latency. But these can only be addressed once such a network is deployed.

We address the packet communication process as a target for the reduction of latency. The packet communication process has essentially three stages - packet preparation, packet transmission and packet handling at receiver. Primarily we look at the packet preparation stage of the communication over the network. This is the period from where the processor knows that it has to communicate with an external component (w.r.t. to its tile), to the time it delivers the packet to the network logic of that tile, which eventually delivers it to the destination component. Since this stage could be a possible bottleneck, apart from the latency of the communication channel, we try to reduce the latency exhibited by the system at the beginning of the communication process. These results can also be used for analyzing the system for the final stage of communication too, the packet handling stage. Since these stages are essentially complementary, they will exhibit similar tendencies.

The experimental scenario considered for our research, was of a simple distributed memory environment. The system consists of a core that can access separate memory cores spread through the on-chip network. To the software executing on the processor core, the memory is one contiguous block present at a single location. The processor core is aware of the distributed nature of the memory space. When the software attempts to access a memory location, the destination core has to be identified and accessed. We shall demonstrate how this is implemented in the three different schemes. Before we examine the packet preparation steps and methods, we take a look at the generic structure of the packets. The structure of the packet can be tuned for a particular network, so as to reduce the overhead of packetizing the data. A packet essentially consists of 3 parts, the packet header, the packet data and the packet tail.

Packet Header	Packet Data	Packet Tail

Figure 3 – Packet Structure

The packet header contains the necessary routing and network control information. These will be the destination and source addresses. When source routing is used, the destination address will be ignored. It is replaced with a route field that will specify the route to the destination. A disadvantage of source routing is the added overhead of including the route field in the packet header. But the inclusion of such a field reduces the complexity of the routing logic on the cores on the network. It simplifies their routing decisions and their task will be to just look at the route field and route the packet over the specified output port.

The packet data consists of essentially two types of information. The first is the control information that will indicate to the receiving memory core about the type of memory request being made. The second will be the actual data, i.e. the memory address being accessed. The packet tail contains error-checking code and error-correcting code. But this part of the packet is optional. The inclusion of this information will depend on the error probability of the underlying network.

The packet structure utilized for this research was tuned to the corresponding implementation strategy. The structures are further described in the following sections. With the simple distributed memory environment scenario in mind, we identified the generic operational steps that need to be performed when an address at a memory core needs to be addressed. Figure 4 illustrates these steps.

The set of operations stated above are executed at different locations, depending on the type of packetization strategy. The location will be incumbent on the configurability and programmability characteristic of the core in question.

Step 1: *Translate address by determining which memory core needs to be accessed, and determine the effective address at that memory core.*
Step 2: *Prepare packet header by setting the source address and the route to the destination.*
Step 3: *Examine program instruction requiring memory access, and set control flags in the packet data.*
Step 4: *Set effective address in packet data.*
Step 5: *If using error-checking codes and error-correcting codes, calculate the values and set them in the packet tail.*
Step 6: *Assemble packet and deliver to the network logic of the core.*

Figure 4 – Generic packetizing process for a Simple Distributed Memory Model

As mentioned in Section 1, the main focus of this research has been the analysis of the alternative packet preparation methods available to the system designer. For our research, we used the Xtensa Processor Core from Tensilica [7]. This configurable, extensible and synthesizable processor core was designed specifically to address Embedded System-on-Chip (SoC) applications. This processor can be molded by the system designer to suit the application. The designer can also describe additional data-types, instructions and execution units using the Tensilica Instruction Extension (TIE) language. Using this core, it is possible to develop an application specific core for packetization. In the following sections we shall discuss three implementations of the packetizing modules.

3.1. Software Library for packetization

The software implementation of the packet preparation provides the user with a library of instructions that can be used to access a memory address in a distributed memory space. The library requires three configuration files. These files provide important network associated properties. Figure 5 provides an overview of the configuration file structure.

The *socnet.conf* configuration file specifies the address of the host. It also provides the route information to the network elements. The packet structure used is specified in *packet.conf* and this specifies the fields in the packet and their corresponding size in terms of bits. *mem_alloc.conf* contains the memory allocation information, i.e. the address space of the memory cores in the environment. When the user issues a memory access instruction, the corresponding packet is prepared according to the steps highlighted in the figure 4.

The sample code to test the library was executed on the basic Xtensa Processor Core. The core was configured with a 128-bit processor interface. The cycle count for the execution of the packetization instruction was determined by using the profiling tool - *xt-gprof* - included in the Xtensa toolset. The area results for this strategy are the size of the software library code.

3.2. On-core module for packetizing

In this implementation we utilized the Xtensa Processor Core's configurability and its TIE language to define instructions for preparing the packets. This program. The Xtensa toolset has tools that allow the profiling of the executed instructions. From the profile one can obtain the required results such as the cycle-count for the executed instructions The TIE compiler also generates the required Verilog/VHDL files that are then analyzed using Synopsys Design Analyzer, to obtain the

```
HOST_ADDRESS=<address value>

<num of routes in route table>
<destination> :: <route to destination1>
<destination> :: <route to destination2>
<destination> :: <route to destination3>
…
…
```

socnet.conf
(a)

```
PACKET_HEADR <num of fields>
<header field 1> = <size of field 1>
<header field 2> = <size of field 2>
<header field 3> = <size of field 3>
…
PACKET_DATA <num of fields>
<data field 1> = <size of field 1>
<data field 2> = <size of field 2>
<data field 3> = <size of field 3>
…
PACKET_TAIL <num of fields>
<tail field 1> = <size of field 1>
<tail field 2> = <size of field 2>
<tail field 3> = <size of field 3>
…
```

packet.conf
(b)

```
<num of mem cores>
<start addr1>::<end addr1> <core addr>
<start addr2>::<end addr2> <core addr>
<start addr3>::<end addr3> <core addr>
…
```

mem_alloc.conf
(c)

Figure 5 – Configuration File Structures

timing and area costs.

The TIE definition used for our research, included the specification of the stages listed in Figure 4, in terms of the TIE language. The TIE code was successfully compiled with the TIE compiler and the execution of the custom instruction was tested on the Xtensa processor. The packet structure used in this implementation is equivalent to the one shown in Figure 6. The cycle count for this implementation was obtained using the Instruction Set Simulator (ISS), provided with the Xtensa tool set. The ISS provides detailed information on the contents of the registers in use and the output available at the processor interface.

3.3. Wrapper logic for packetizing

For cores that are neither programmable nor reconfigurable, the only option for interfacing with the networking logic of the tile is to utilize a wrapper, which would have the responsibility of packetizing and de-packetizing the cores requests and responses. The wrappers have the responsibility of (i) receiving the contents from the core interface, preparing the packets and dispatching them to the network logic of the tile and (ii) receiving the packets from the networking logic and presenting the contents to the core interface.

For our experiment, we designed the packetizer module of the wrapper, which was compliant with VSI Alliance's Virtual Component Interface (VCI) Standard Version 2 [8]. This standard defines the basic characteristics of the Virtual Component Interface (VCI). It provides detailed information on the different complexity interfaces, the Peripheral VCI (PVCI), the Basic VCI (BVCI) and the Advanced VCI (AVCI). We developed the wrapper that would be compliant with the BVCI standard. The implementation details are not provided here due to the restrictive nature of the standards document. The VSIA vision is to dramatically improve the productivity of SoC development by specifying open standards and specifications that facilitate the integration of software and hardware VCs from multiple sources. This was the reason we chose to develop a wrapper compliant with the VCI standard because we believe that most future cores will have well-defined interfaces similar to or be VCI standard compliant.

The packetizing module attempts to optimize the packets being generated for the on-chip network. The packet structure in this implementation was dependent on the signals used in the BVCI interface (details cannot be provided due to non-disclosure agreement). The packetizer module maintains the address translation information, i.e. the mapping of memory addresses to destination core addresses. It analyses the content of the core request and tries to optimize on the amount of data being sent over the on-chip network, by filtering the redundant information from the packets. The timing, and area analysis for this implementation was obtained using the Synopsys' Design Analyzer.

4. Results

The results obtained for the analysis carried out evaluates the performance of the packetization schemes in terms of latency, and area. Table 2 provides a summary of the results that were obtained for the latencies experienced. The latency in the case of the software library, was determined using the cycle count obtained from the Instruction Set Simulator (ISS) and the clock frequency. This result will vary with different processors and implementations of the packetization library. In the case of the on-core packetization, the latency was determined in a similar way. However, the clock frequency was obtained through synthesis of the TIE specification. It should be noted here that, these two schemes were implemented on the Xtensa Processor Core. The lower clock frequency is due to the slow-down caused by the TIE logic that was incorporated into the processor core. This is an acceptable trade-off, in light of the performance improvement. This conservative result was obtained by using the TSMC 0.18micron and slow libraries. With a little more effort and better libraries it is possible to have the Xtensa processor operate at its normal clock frequency of 200MHz and will further reduce the latency. The result for the wrapper implementation is obtained using the 0.35micron technology library. With better technology, there will be a further reduction in the latency. The latency result in this case provides the developer with the time taken through the longest path in the wrapper, and will enable him to decide on the clocking rate for the interface.

Table 2 – Latency Results

Packetization Strategy	Cycle Count	Clock Frequency	Latency
Software Library	47	193MHz	243.5ns
On-core packetization	2	185Mhz	10.8ns
Wrapper packetization	-	-	3.02ns

Table 3 – Area Results

Packetization Strategy	Area	Remark
Software Library	118 KB	Code size
On-core packetization	13K	Gate count using 0.18micron technology
Wrapper packetization	4K	Gate count using 0.35micron technology

Table 3, provides the area costs of the three schemes that were implemented. The area cost of the three implementations cannot be compared quantitatively. The results provide a measure of the cost that the system designer would experience using a particular strategy. The area for the software implementation was determined in terms of the code size of the software library. To determine the area of the TIE logic, for the on-core packetization, the TIE specifications were synthesized following the regular steps (i.e. compilation of the TIE specification and synthesis of the compiler output). The 13K gate count is an overly conservative estimate. The silicon area appeared to be under 0.2 square mm. The area for the wrapper was determined using Synopsys' Design Analyzer. It cannot be directly compared to the one obtained for the Xtensa Core, as the technologies used in both are considerably different.

5. Conclusions

In this paper, we proposed three different packetization schemes to analyze their overhead in core-network logic interface in a SoC with a packet-switched network. The results provided by this research are conservative and give us an insight into the complexities and the intricacies that are involved when cores or their corresponding wrappers are developed for SoCs that use packet-switched network on-chip communication. Design decisions will be incumbent upon the latencies and the area overhead factors that have been analyzed in this research. Since communication is a major factor in hardware-software partitioning, an accurate communication cost model will aid in robust system design. This network-aware partitioning is another area of research that needs to be researched into.

6. Acknowledgements

We would like to thank Tensilica Corp. for their Xtensa Toolset and synthesis results for our TIE specifications. We also thank the VSI Alliance for access to their standards and specifications.

7. References

[1] G. De Micheli and L. Benini, "Networks on Chip: A New SOC Paradigm", IEEE Computer, Vol. 35 Issue: 1, Jan 2002, pp. 70 -78

[2] L. Benini and G. De Micheli, "Powering Networks on Chips", System Synthesis, 2001. Proceedings, The 14th International Symposium on, 2001, pp. 33 -38.

[3] P. Guerrier and A. Greiner, "A generic architecture for on-chip packet-switched interconnections", Design, Automation and Test in Europe Conference and Exhibition 2000. Proceedings, 2000, pp. 250 -256.

[4] W. J. Dally and B. Towles, "Route Packets, Not Wires: On-Chip Interconnection Networks", Design Automation Conference, 2001. Proceedings, 2001, pp. 684 -689.

[5] Sgroi, M., et. al., "Addressing the system-on-a-chip interconnect woes through communication-based design", Design Automation Conference, 2001. Proceedings, 2001, pp. 667 - 672.

[6] N. Swaminathan, "Communication Synthesis for On-Chip Networks", Masters Thesis, Texas A&M University, 2002.

[7] Tensilica Xtensa Core, www.tensilica.com

[8] VSI Alliance, Virtual Component Interface Standard Version 2 (OCB 2 2.0), April 2001. www.vsi.org [Document is only available to members]

[9] Kumar S., et. al., " A Network on Chip Architecture and Design Methodology", Proceedings of IEEE Computer Society Annual Symposium on VLSI, April 2002.