Centralized End-to-End Flow Control in a Best-Effort Network-on-Chip[*]

P. Avasare[1], V. Nollet[1], J-Y. Mignolet[1], D. Verkest[1,2], H. Corporaal[3]
[1]IMEC V.Z.W., Kapeldreef 75, 3001 Leuven, Belgium
[2]Vrije Universiteit Brussel and Katholieke Universiteit Leuven, Belgium
[3]Technical University Eindhoven, The Netherlands

ABSTRACT

Run-time communication management in a Network-on-Chip (NoC) is a challenging task. On one hand, the NoC needs to satisfy the communication requirements (e.g. throughput) of running applications competing for NoC resources. On the other hand, the NoC resources should be managed efficiently while keeping additional management functionalities minimal. This paper details a NoC communication management scheme based on a centralized, end-to-end flow control mechanism deployed in a best-effort NoC. This scheme comes at a very low resource (i.e. limited hardware and run-time overhead) cost. We show that by using a flow control mechanism it is possible, even in a best-effort NoC, to provide sufficient communication guarantees with respect to the application requirements. Finally, we illustrate the applicability of our approach for real-life multimedia applications.

Categories and Subject Descriptors

C.3 [**Special Purpose and Application-based Systems**]: Real-time and embedded systems; D.4.4 [**Operating Systems**]: Communications Management

General Terms

Algorithms

Keywords

Network-on-chip, Run-Time Communication Management

1. INTRODUCTION

In order to meet the ever-increasing design complexity, future platforms will require a mixture of (heterogeneous) processing elements integrated on a single chip, also termed as a System-on-Chip (SoC). Interconnecting these processing elements (further denoted as *tiles*) will require a flexible structure like a Network-on-Chip (NoC) [1, 2].

There is a challenge in designing the right NoC communication (also denoted as *traffic*) management schemes. Meaning that one should provide enough Quality-of-Service (QoS) guarantees with respect to the application requirements, while minimizing the hardware/software resources needed to enforce these requirements. In a NoC, providing such guarantees with respect to communication throughput, latency and jitter (variation in message delay) boils down to managing the (potential) congestion in the NoC router and providing a flow control mechanism. Typically congestion is controlled by having a bandwidth reservation mechanism that ensures a deterministic throughput and latency and that simultaneously minimizes the jitter [4]. However, such a scheme requires more complex NoC routers in addition to having a run-time bandwidth allocation mechanism.

Our best-effort routers do not contain any functionality for providing hard guarantees for e.g. message latency. However, as long as the network is not operating near its saturation point, it is possible to provide an upper bound for e.g. message latency [3]. To avoid this network saturation, we use a message injection rate control mechanism. The potential of such an approach was mentioned by Duato [3].

A flow control mechanism should ensure that a message producer task does not produce more messages than the message consumer task can handle. There are several approaches for handling this issue e.g. dropping the messages or using credit-based flow control [4]. In our approach, we show that, it is possible to just (temporarily) store the messages causing network congestion onto the network itself. The resulting potential network congestion is then solved by using a message injection rate control mechanism operated by an algorithm within the central operating system (OS). In short, the central OS monitors communication at every tile-NoC interface. Based on this information and its platform-global view, the OS can limit/shape the traffic a tile is allowed to inject into the NoC. This way, the OS can match the communication rate of the data producer with that of the data consumer (i.e. flow control) in order to handle network congestion.

The main contribution of this paper is the description and evaluation of a centralized NoC traffic management algorithm based on a message injection rate control mechanism. Furthermore, we illustrate the applicability and the performance of our approach using an emulated NoC with real-life multimedia applications.

[*]Work partly funded by the European Commission (IST-AMDREL project IST-2001-34379), by the Flemish Government (GBOU-RESUME project IWT-020174-RESUME) and Xilinx Labs, Xilinx Inc. R&D Group.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EMSOFT'05, September 19–22, 2005, Jersey City, New Jersey, USA.
Copyright 2005 ACM 1-59593-091-4/05/0009 ...$5.00.

The rest of this paper is organized as follows. Section 2 describes the NoC emulation platform. Section 3 details the simulation model we built for our NoC traffic management experiments. Section 4 explains the developed heuristic algorithm that enables efficient NoC traffic management with the simulated experiments. Section 5 presents the results of applying this algorithm on our NoC emulation platform. Finally, Section 6 presents the conclusions.

2. NOC EMULATION PLATFORM

This section details the architecture and capabilities of our NoC emulation platform. A more in-depth description can be found in [5, 6].

Our emulated packet-switched multiprocessor NoC is implemented as a 3x3 bidirectional mesh linking a StrongARM SA-1110 processor (206 MHz), present inside an iPAQ, to an FPGA containing the slave processing elements and the NoC (33 MHz clock). Our packet-switched NoC actually consists of two independent NoCs. The *data NoC* is responsible for delivering data packets, while the *control NoC* is used for transferring OS-control messages (Figure 1). This separation is vital to our traffic management scheme since the control NoC provides a way to control the traffic even in case of data NoC congestion.

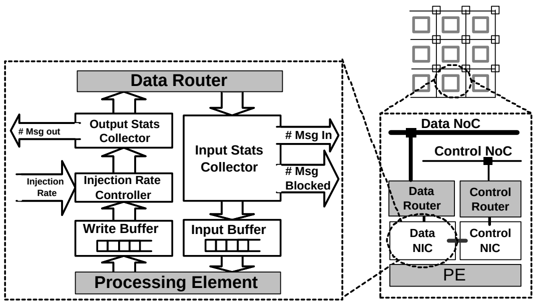

Figure 1: Data and control NIC functionality.

All processing elements are interfaced to the packet-switched data NoC by means of a data Network Interface Component (data NIC). The main role of the data NIC is to buffer input and output messages and to provide a high-level interface to the data router (Figure 1). The data NIC is also responsible for collecting the local processing element message statistics (i.e. number of messages sent, received, blocked, etc.). Especially the number of blocked messages is important: these messages potentially disturb other data traffic passing through the same NoC link. Each tile also contains a control Network Interface Component (control NIC). The control NIC is responsible for providing the central OS with the information (e.g. statistics collected in the data NIC) to base its management decisions on.

The data NIC also implements an *injection rate control mechanism*, allowing the OS to control the amount of messages the attached processing element can inject into the data NoC. The injection rate parameters are passed on to the data NIC from the OS via the control NIC.

This means the OS can limit the time wherein a certain tile is able to send messages onto the network. This time is called the *send window* of a tile. By setting the low (L) and high (H) values for a tile on a NoC, the OS is able to

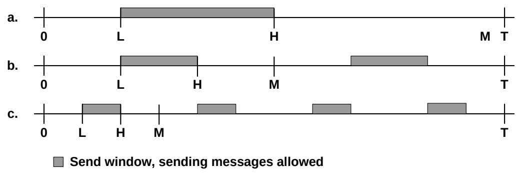

Figure 2: The size and location of 'send window' is specified by a low value (L), a high value (H) and a modulo value (M).

describe a single send window within the whole *send spectrum* (Figure 2a). However, by also using the modulo value (M), this single send window can be spread over the whole send spectrum (Figure 2b, 2c). This second technique, further denoted as *window-spreading technique*, is not intended to limit the amount of messages injected into the NoC, but rather to shape the traffic to achieve maximum throughput. Note that the send windows of different producers that share a link are allowed to overlap [5].

In our previous experiments with the platform, we controlled the injection rate manually for controlling congestion on NoC [5]. We found that, by using the window spreading technique, NoC traffic was more evenly spread over time resulting in sufficient throughput (close to the best-effort network) and considerably less jitter. Moreover, a proper window setting could hide the latency of the receiver side and completely suppress blocking on the network.

Finding these optimum window values manually for a set of simultaneously running applications will be difficult, especially due to the inherent dynamism present in real-life multimedia applications. Hence, it requires an adapted algorithm that monitors the network conditions and that calculates and sets the optimal send window values wherever and whenever necessary.

3. TRAFFIC CONTROL EXPERIMENTS

Due to the long exploration turnaround cycle when directly working on our NoC emulation platform, developing a traffic management scheme using the platform is not a viable option. Hence, we developed a simple simulation model of our platform for the algorithm exploration. This section first describes the simulation model and further details the traffic management exploration experiments.

3.1 Simulation Model

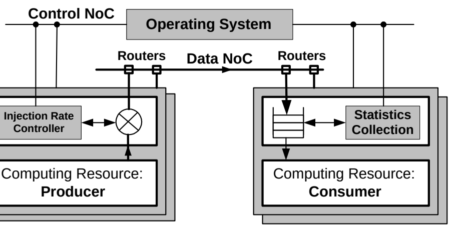
Control NoC
Operating System
Routers
Data NoC
Routers
Injection Rate Controller
Statistics Collection
Computing Resource: Producer
Computing Resource: Consumer

Figure 3: NoC traffic management simulation model

The simulation model is built on top of the OMNET++ [7] network simulation environment. This model (Figure

3) contains four distinct blocks: (i) two producers and two consumers that communicate over the NoC, (ii) a control NIC and a data NIC for every producer and consumer to enable statistics collection and injection rate control, (iii) a set of data NoC routers with round-robin scheduling and, (iv) a central OS to steer the traffic management.

The producer generates data messages which pass through the data NIC before arriving in the data network. The allowed rate of injection for these messages is controlled by the data NIC which gets the injection parameters from the centralized OS through the control NIC. The windowing mechanism (Figure 2) is used for achieving this communication control. The data NoC takes in these messages from the producers in a round-robin fashion and forwards them to their respective consumers. The OS meanwhile periodically collects the message traffic statistics at the different consumer control NICs in order to take traffic management decisions. Note that in reality (i.e. on the emulation platform) the link between a producer and a consumer can span multiple NoC hops. However, since the traffic management works on an end-to-end basis (i.e. retrieving information at the consumer side, while controlling injected traffic at the producer side), multiple hops can be abstracted as a single link (Figure 3).

3.2 Experiments With The Simulation Model

This section details validation and experimentation with our simulation model.

First, the simulation model was validated using the observations (in terms of throughput and blocking) gathered from the experiments performed on the emulation platform [5]. In addition, many simulation model parameters were picked up from the same experiments e.g. for achieving injection rate control mechanism (Figure 2), the whole send spectrum is divided into window-slots, each 100 μs wide. The OS sampling rate for gathering the NoC communication statistics was chosen as 50 ms (more than a period of a frame for a typical multimedia application running at 25 frames per second). As a QoS requirement, the user specifies the required communication bandwidth between a producer-consumer pair for the simulated user application.

The model is used to study two important aspects of NoC communication. First is to study effects of blocking on throughput and jitter on the NoC and the second aspect is about dealing with such a blocking, in order to achieve the user-specified QoS throughput and minimize jitter.

For studying the first aspect, we model one communicating producer-consumer pair. This model is then extended with another producer-consumer pair to estimate the effects on throughput due to sharing common NoC resources. In the second aspect of dealing with this blocking, we use the injection rate control mechanism to control blocking.

For the experiments, the producer is modeled in two modes. In the first mode, the producer generates messages according to a (statistical) normal distribution, while in the second mode the producer generates periodic bursts. The first mode represents a general case of a producer, whereas the second mode resembles the communication of multimedia related applications. For example, a Motion-JPEG decoder at 25 frames per second will generate a burst of data messages at every frame i.e. at every 40 ms.

The initial experiments brought forward two important observations. First, blocking on the network indeed drastically affects throughput and introduces non-deterministic jitter. Second, if the NoC traffic is kept at the level just below where blocking starts, the network resources are utilized at their maximum (i.e. throughput is maximum). This point where the blocking starts depends on various factors such as the difference between consumer-producer input-output rates, input-output buffer sizes in consumer-producer, message buffer spaces in routers and the routing algorithm. Hence, we developed a run-time heuristic algorithm that uses the injection rate control mechanism to minimize the amount of blocking while retaining a maximum throughput.

4. TRAFFIC CONTROL ALGORITHM

Consider one producer-consumer pair. The algorithm, detailed in Algorithm 1, starts out with initial window values (line 1). These initial values are based on the user-specified throughput requirement and the measured throughput of other communicating tasks sharing a known NoC bandwidth over a common link.

Algorithm 1 Finding *send window* values for a tile

```
 1: NewWin = StartWin;
 2: loop
 3:   SetWindow(NewWin); // set only if values change
 4:   CurrWin = NewWin;
 5:   WaitPeriodAndGetStats(CurrStats);
 6:   if $CurrStats.Blocking > THRESHOLD$ then
 7:     if $FoundOptimumWinValues = true$ then
 8:       FoundOptimumWinValues = false;
 9:       NewWin = StartWin;
10:       Reset(BestWin, BlockingWin);
11:     else {Yet to find optimum send window values}
12:       NewWin = Reduce(CurrWin);
13:       BlockingWin = CurrWin;
14:     end if
15:   else if $CurrStats.Thruput < REQUIRED$ then
16:     NewWin = Increase(CurrWin, BlockingWin);
17:   end if
18:   if $WinStable(CurrWin, NewWin) = true$ then
19:     if $CurrStats > BestStats$ then
20:       BestWin = CurrWin; BestStats = CurrStats;
21:     end if
22:     if $CanSpreadWin(CurrWin) = true$ then
23:       NewWin = Spread(CurrWin);
24:     else {Exhausted window spreading}
25:       NewWin = BestWin;
26:       FoundOptimumWinValues = true;
27:     end if
28:   end if
29: end loop
```

Periodically, the OS collects traffic information (line 5). The OS checks if the reported amount of blocked messages exceeds a certain threshold value (line 6). If so, the algorithm will have to (re)calculate the optimal window values. This mainly involves reducing the size of the send window of the producer until blocking drops below the threshold.

If the amount of blocking does not exceed the threshold value but the throughput is lower than required, then the send window size will be increased as long as the new window size remains smaller than the *blocking* window size (line 16). As soon as the window values stabilize (i.e. converge, line 18), the algorithm will spread the send window using the window modulo (M) value. For each modulo value, the send window low (L) and high values (H) at which NoC blocking starts are determined. After maximally spreading the send window over the send spectrum (line 22), the window values that deliver the best communication in terms of throughput and blocking are chosen (line 25).

The efficiency of this algorithm is measured in terms of two key factors: the first regarding the amount of blocking, throughput and jitter on the NoC and the second regarding the amount of run-time resources that the algorithm will use for computing the injection rate control window values.

In our simulations, it takes around 16 iterations for the algorithm to find the optimum send window values for both producers sharing a NoC link. By setting the producer's window values obtained from the algorithm, the total throughput is very close to the one achieved with the best effort service. At the same time our traffic management scheme completely eliminates NoC blocking in contrast to a pure best-effort service, except when (re)calculating the send window values. Furthermore, when the blocking on the NoC is completely eliminated, jitter becomes minimal.

The occasional exceptional blocking is either caused by additional traffic on the NoC generated by other producer(s) or due to a change in burst characteristics of a producer. Typically this will happen when user starts a new application or changes requirements of current application. At such times, temporary QoS violations could be accepted by user.

5. ALGORITHM APPLICATION

In order to test viability of the above algorithm in real-life applications, we inserted the algorithm inside our NoC platform OS running on a StrongARM (206 MHz) based handheld device. We measured on our NoC emulation platform (running at 33 MHz clock-speed) that at every sampling time, the OS takes 60μs to gather communication statistics from a tile. From these message statistics and the current send window values, the algorithm calculates, at every sampling iteration, new send window values in 65μs on an average, with a minimum of 12μs and a maximum of 120μs. In case the algorithm calculates different new window values than the current ones, the OS needs to modify the injection rate at the producer tile. This operation for setting window values on a tile takes 57μs on our platform. Totally, incorporating such a traffic management inside our NoC platform OS takes on an average 182μs per tile at every sampling i.e. every 50ms in our case.

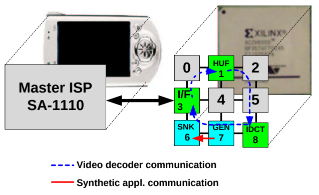

Figure 4: Emulation platform running MJPEG

Figure 4 shows our case study application running the MJPEG video decoder [5]. It is composed of four tasks running concurrently. Two of these tasks, the sender and the receiver, run on StrongARM (connected through tile 3). Two other tasks are hardware blocks: a task that performs the Huffman decoding and dequantization (HUF task on tile 1) and a task that performs a 2D-IDCT and a YUV to RGB conversion (IDCT task on tile 8). Additionally, we have message generator (GEN task on tile 7) and message sink (SNK task on tile 6) tasks that are synthetic applications generating and consuming messages at a constant rate. First the MJPEG application is run on the NoC and then GEN and SNK tasks are inserted on tiles 7 and 6 such that both applications will share the link between tiles 7 and 6.

Initially, after inserting GEN and SNK tasks, the NoC is run without any traffic management. The MJPEG decoder throughput reduces significantly due to heavy blocking on the shared link (between tiles 6 and 7). At this point, we switched on our algorithm inside the OS. We found that the additional CPU load overhead inside the OS due to the algorithm insertion is negligible (less than 1%). Further, the algorithm converges to optimum window values such that the required throughput for MJPEG application is achieved by reducing injection rate of the GEN task. Compared to the simulated results where the algorithm converges within 800 ms (16 iterations), on our NoC platform the algorithm takes on average one second (19-21 iterations) to find optimum window values for IDCT and GEN tasks sharing a link.

The main reasons for this significant difference between the simulation time and the actual measured time is the bursty nature of MJPEG communication. For such bursty traffic, the user-specified throughput requirement proves to be insufficient to calculate a good starting set of window values within the algorithm. Calculating these initial window values needs to take into account the burst characteristics such as periodicity, width and magnitude.

6. CONCLUSIONS

This paper details a centralized algorithm for handling communication management in a Network-on-Chip. In essence it is a centralized, low-cost end-to-end flow control algorithm based on an injection rate control mechanism. Its goal is to maximize communication throughput (with respect to the user's specification), while minimizing jitter. The presented mechanism provides a weak form of QoS, but its main advantage is that it comes at a low cost (i.e. low hardware complexity, low run-time overhead), while it proves to be good enough for the NoC platforms targeting multimedia applications. In our case study with one shared link on a 3x3 tile NoC, less than 1% of the StrongARM execution time was required to implement the management algorithm.

7. REFERENCES

[1] L. Benini, G. DeMicheli, "Networks on Chips: A new SOC paradigm?", IEEE Computer magazine, January 2002.
[2] W. J. Dally, B. Towles, "Route packets, not wires: on-chip interconnection networks", DAC 2001, p684-689.
[3] Jose Duato, S. Yalamanchili, L. Ni, "Interconnection Networks", Morgan Kaufmann Pub., 2002, p428-431.
[4] A. Rădulescu, K. Goossens, "Communication Services for Networks on Chip", Book chapter in "Domain-Specific Processors : Systems, Architectures, Modeling, and Simulation", Marcel Dekker Pub., 2003.
[5] V. Nollet, et al., "Operating System controlled Network-on-Chip", DAC 2004, p256-259.
[6] T. Marescaux, et al, "Runtime Support for Heterogeneous Multitasking on Reconfigurable SoCs", Integration the VLSI journal (Elsevier), Vol 38, Oct. 2004, p107-130.
[7] OMNET++: a discrete event simulator system, http://www.omnetpp.org