SysCon 2008 – IEEE International Systems Conference
Montreal, Canada, April 7–10, 2008

System-Level Modeling of a NoC-Based H.264 Decoder

Ankur Agarwal [1], Cyril-Daniel Iskander [2], Hari Kalva [1], Ravi Shankar [1]
ankur@cse.fau.edu, cyril_iskander@hotmail.com, hari@cse.fau.edu, ravi@cse.fau.edu
[1]Dept of Computer Science and Engineering, Florida Atlantic University, Boca Raton, FL, 33431
[2]Hi-Tek Multisystems, Québec, QC, Canada

***Abstract* - Networks-on-chip (NoC) are expected to play a key role in future embedded systems. A NoC-based system has the potential to support concurrent processing, in both software and hardware. This can however lead to concurrency issues. We present a multiprocessor system modeling and performance evaluation approach that addresses concurrency. We illustrate our methodology by mapping a H.264 decoder onto a 4×3 mesh-based NoC architecture. We show latency, area, and power consumption results for this NoC architecture abstracted from its FPGA implementation.**

***Keywords* - network-on-chip, NoC, MLDesigner, H.264, concurrency.**

I. NOC PLATFORM FOR EMBEDDED SYSTEMS

Network-on-chip (NoC) based systems have the potential to address bus-based system concerns, and improve design productivity by supporting modularity and reuse of complex cores, thus enabling a higher level of abstraction in the architectural modeling of future systems [1]-[4]. The NoC architecture implies a shift in concern from computation and sequential algorithms to the modeling of concurrency, synchronization and communication [5]. In a multiprocessing environment, various processes execute simultaneously. At a given time, every resource may be either producing or consuming data packets. Therefore, several inter-processes communications take place in such a model, and if not modeled properly this may lead to system failure. In this paper, we propose a methodology for modeling a concurrent system architecture, and illustrate it by mapping a Nokia H.264 decoder onto six different cores. The NoC is designed and modeled using the MLDesigner system-level environment, which supports different models of computations, and which allows components to be abstracted as classes.

II. NOC CONCURRENCY MODELING

Designers exploit design reuse to enhance system design productivity. Integration of pre-designed reusable blocks may fail if these blocks execute in parallel, share resources, and/or interact with each other. Such concurrency issues, if not addressed, may be detrimental to the normal functioning of the system. Multiprocessor architectures, recently introduced to extend the applicability of the Moore's law, depend upon concurrency and synchronization in both software and hardware to achieve that goal. Concurrency issues, if ignored, may also lead the system into a deadlock or a livelock state. Traditional system design integration and verification approaches will not be cost-effective in exposing concurrency failures as these are intermittent: such failures can significantly increase time-to-market and field failures. To overcome such failures, one would have to develop abstract concurrency models and do exhaustive analysis on these models to test for concurrency problems.

Fig. 1 shows the concurrency modeling flow chart used in developing our NoC. A concurrency model is developed with a high level platform and data independent specifications. Processes are identified from these specifications, which are later modeled as components, once concurrency concerns are addressed. The concurrent model is developed using Finite State Processes (FSP) and is analyzed for concurrency concerns with LTSA [5], [6]. This concurrency model is then ported to the architect's design phase, for analyzing system performance and for estimating resources needed to map an application onto a system. More details on our concurrency modeling approach can be found in [5].

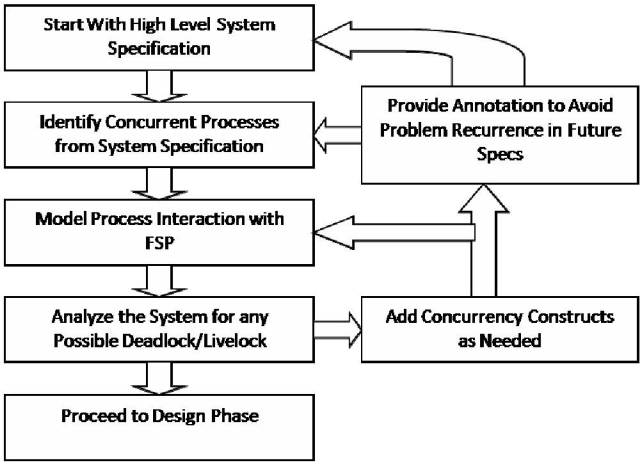


Fig. 1: Concurrency modeling flow chart.

III. SYSTEM-LEVEL MODELING OF NOC

MLDesigner is a system-level design and modeling environment. It allows one to model a system at an abstract level. It supports modeling in different domains such as the Discrete Event (DE), Synchronous Data Flow (SDF), Finite

978-1-4244-2150-3/08/$25.00 ©2008 IEEE

State Machine (FSM), Dynamic Data Flow (DDF), and Synchronous Reactive (SR) domains, among others. Multiple domains can further be combined to represent a system model. Hence, a system model can be represented with any level of details in any part of it. One can do performance analysis on a developed system model. As this performance analysis is done at an abstract level, it is therefore optimized in running the simulation faster as compared to other modeling environments, which are C, C++ or SystemC-based.

Such a model at the design phase will allow one to make key decisions, such as the number of processors, HW/SW partitioning, the estimated performance values for new components, and the use of existing components in software or hardware. Such design decisions have the potential to significantly enhance the productivity of system design. One can further abstract the performance parameters for a given application (e.g. H.264 decoding) on a particular architecture and capture the resource requirements in terms of processor speed, cache size, protocol requirements, power consumption, latency, and silicon area. Fig. 2 shows this system modeling flow.

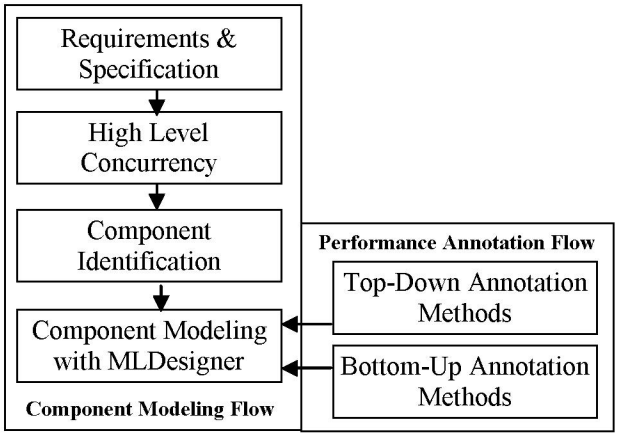


Fig. 2: System modeling flow.

We used MLDesigner to model all the key NoC building blocks, also denoted as *classes*. These classes were identified from a high level system specification. They are: *Producer* (P), *InputBuffer* (IB), *Scheduler* (S), *Router* (R), *OutputBuffer* (OB) and *Consumer* (C). Fig. 3 shows the high-level interaction among these classes in a 3×3 mesh topology [7]. Each node is made up of R, S, and multiple OB and IB instantiations. In the following sections, we discuss the specific implementation of each of the NoC classes.

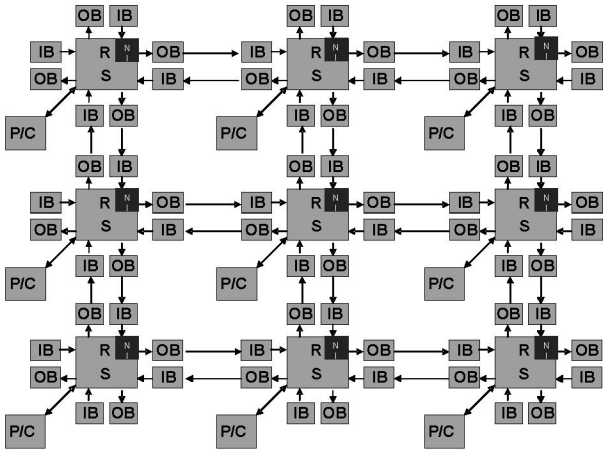


Fig. 3: 3×3 mesh-based NoC architecture.

A. Producer Class

A producer is instantiated from the *Producer* class. It comprises a resource and a resource network interface (NI). A producer generates the required traffic pattern and packetizes the data into flits. A flit is the smallest unit of communication supported in the NoC. The *Producer* class has been implemented with a SDF model of computation as it is responsible for continuous data flow.

A producer outputs a flit which is time-stamped at the time of its generation. The timestamp is used to determine the latency involved in delivering the flit. The source and destination address fields of the flit header are updated at this time. As shown in Fig. 4, the flit header has fields for its priority, timestamp, X-direction of source address, Y-direction of source address, X-direction of destination address, and Y-direction of destination address. The priority of this flit is governed as per a statistical distribution block. For example, in the case of a uniform distribution pattern, every third flit will be a high priority flit. Once the new flit has its timestamp, source and destination addresses and priority fields updated, it is then forwarded to the output.

The customizable parameters for the *Producer* class are: (1) the distribution pattern of the data; (2) the packet injection rate, i.e. the amount of data generated as a function of time; (3) the priorities of the generated data - High, Mid or Low. Each set of parameters is described below.

We used three statistically generated traffic distribution patterns – uniform traffic with linear destination, random, and application-specific patterns.

The packet injection rate represents the number of flits per cycle injected into the network for transmission. By defining it as a customizable parameter, this allows us to test the NoC model for varying load conditions.

We provided three priority levels for data packets in our NoC model: High priority, Mid priority and Low priority. High priority supports control signals such as Read (RD), Write (WR), Acknowledge (ACK), and interrupts. Therefore, high priority data is a short packet (single flit packet). Mid priority supports real-time traffic on the system, while Low

priority supports non-real time block transfers of data packets. We have defined control signals as High priority because the data must respond immediately to a control signal. Therefore, a control signal must reach its destination in time to manage the communication flow of the network. Real-time data must be delivered in real-time bounds. Therefore, we have assigned Mid priority to real-time data. The rest of the data on the network belongs to the Low priority class. The number of priority levels is a customizable parameter.

Priority	Time Stamp	Source Address (X)	Source Address (Y)	Destination Address (X)	Destination Address (Y)

Fig. 4: Flit header.

B. InputBuffer Class

An input buffer is instantiated from the *InputBuffer* class. It contains a buffer, a buffer scheduler, and a virtual channel allocator. An input buffer stores the incoming flits, generates the proper handshaking signals to communicate with the scheduler and forwards the flits to the router. An input buffer has an input block and an output block. These two blocks are controlled by a state machine. Thus, we have implemented *InputBuffer* in the DE and FSM domains. Two concurrent FSM's are responsible for storing the input data at the input terminals of the input buffer and forwarding the data to a router at the output terminal of the input buffer. The DE domain is used for implementing a handshaking protocol. A data forwarding path has been implemented based on a "request-grant" signaling approach (other NoC implementations refer to it as flow control logic). Incoming flits corresponding to all the priority levels (High, Mid and Low) are stored in a common buffer. Let *buffSize* represent all the available space in an input buffer. *buffAvail* indicates whether there is space available in the input buffer. The stored data flits are forwarded to the router based on their scheduling criteria. For example, in case of priority-based scheduling, High priority flits are forwarded before the Mid or Low priority flits, etc… Table 1 shows the classification of flit types.

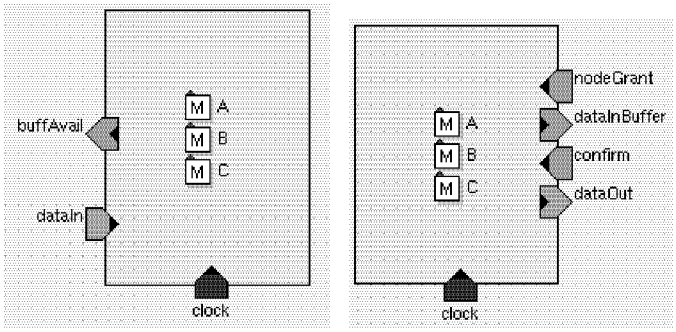
Table 1: Flit priority bit combinations.

Bit combination	Flit type
00	No flit
01	High priority flit
10	Mid priority flit
11	Low priority flit

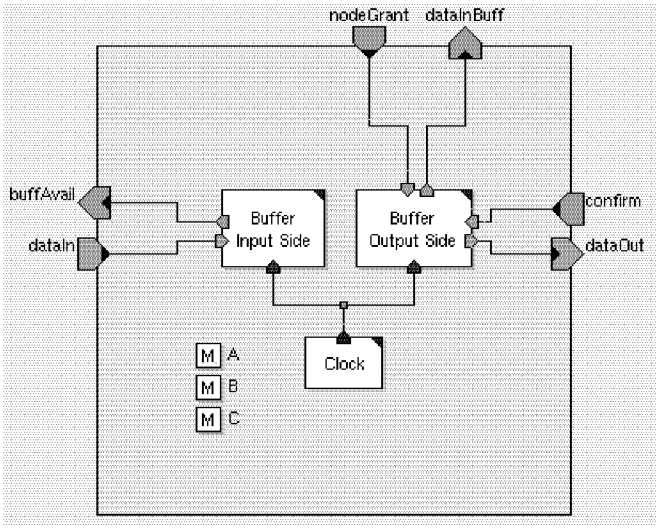
The input buffer is virtually divided into three different buffers A, B and C (see Figs. 5 (a) and (b)). We have provided flexibility in the size of these virtual buffers (the combined size is a fixed user-defined parameter). The input side (i.e. Fig. 5 (a)) is responsible for checking the available buffer space and allocates memory space for an incoming flit. The output side (i.e. Fig. 5 (b)) forwards a flit and releases the allocated memory.

We have implemented a handshaking protocol for forwarding a flit. The availability of data flits in the input buffer for further transmission is indicated by *dataInBuff*. If a grant comes in response to this request (*nodeGrant*), the flit stored in the buffer is forwarded to the corresponding Router. On receipt of *nodeGrant*, a data packet is forwarded by the output side of the input buffer through *dataOut*. Fig. 5 (c) shows the complete implementation of the input side and the output side of the *InputBuffer* class. Three virtual buffers as shown in Figs. 5 (a) and (b) are represented as memories (M) in Fig. 5 (c). A flit is not removed from the input buffer until a confirmation (via *confirm*) is received from the scheduler (from the output side of Fig. 5 (c)). If *confirm* is not received, the data flit is not removed from the input buffer; however, it will be queued for later forwarding. We provided three virtual channels (VC) per buffer. A VC controller inside the input buffer updates these virtual channels.

The customizable parameters for the *InputBuffer* class are the buffer size and the scheduling criteria. We can change the buffer size to any value. By changing the buffer size, we can understand its impact on latency, area and therefore the silicon cost. A buffer forwards data to the next block based on its scheduling criteria. We provided three scheduling criteria (also referred to as service levels, SL): first-come-first-serve (FCFS), priority-based (PB), and priority-based-round-robin (PBRR). In the FCFS scheduling criterion, all the data packets are treated in the same way. A data flit is forwarded to the next block based on its arrival time. The data flit with the earliest arrival time will be forwarded first. In the PB scheduling criterion, the input buffer first forwards all data packets with High priority, then those with Mid priority. It forwards Low priority data packets only when there are no High or Mid priority data packets present in the input buffer. In the PBRR scheduling criterion, the input buffer forwards data packets with different priorities in a specific rotation pattern. It first forwards a High priority packet, then a Mid priority packet, followed by a Low priority packet. This cycle is repeated throughout the simulation.



(a) (b)



(c)

Fig. 5: MLDesigner implementation of the *InputBuffer* class: (a) input side; (b) output side; (c) combined input and output sides.

C. Scheduler Class

A scheduler is instantiated from the *Scheduler* class. The data and control parts of a node have been separated (the *Router* class handles the data part and the *Scheduler* class handles the control signals) to manage concurrency issues and make the design more scalable and reusable. The actual data flows from one node to another through a router. The path of the actual flow of this data is defined by a scheduler. A scheduler is also responsible for synchronizing the input buffer with the router and the router with the output buffer while receiving and transmitting the data further. It schedules the incoming requests for data transmission to the next node, by checking for the availability of the output data path and by arbitrating the requests from various input buffers associated with it. The *Scheduler* class has been implemented in the DE domain. A scheduler is mainly responsible for synchronization, and thus the DE domain is the ideal model of computation (MOC) for its implementation. Fig. 6 shows the MLDesigner implementation of the *Scheduler* class. The scheduler is connected to five instances of *InputBuffer* (one for each direction in the 2-D mesh network and a fifth buffer for the local *Producer* class connected through a NI) and, similarly, five instances of *OutputBuffer* on the output side.

A scheduler accepts the requests from an input buffer via *dataInBuff* (the input signal on the left side of Fig. 6) and allocates the data path by asserting *nodeGrant* (the output signal on the left side of Fig. 6). The data path allocation is based on the availability of an output buffer and the route. We have embedded multiple algorithms in the *Scheduler* class as discussed in the previous section. A scheduler will select an input buffer from multiple input buffers requesting for transmission of a data packet. The router informs the scheduler about the physical output path for flit transmission via o*utputPort*. Availability of the data path is acknowledged by assertion of *confirm*. The *Scheduler* class is implemented in two different models of computation. The control part of the scheduler has been implemented with FSM. This FSM interacts with the DE domain for proper handshaking with the input buffer and router on the input side and the output buffer on the output side of the scheduler.

The customizable parameter for the *Scheduler* class is the scheduling criterion. The *Scheduler* class supports different scheduling criteria: FCFS, PB, round-robin (RR), and PBRR. Thus, in a network we can have a combination of scheduling algorithms.

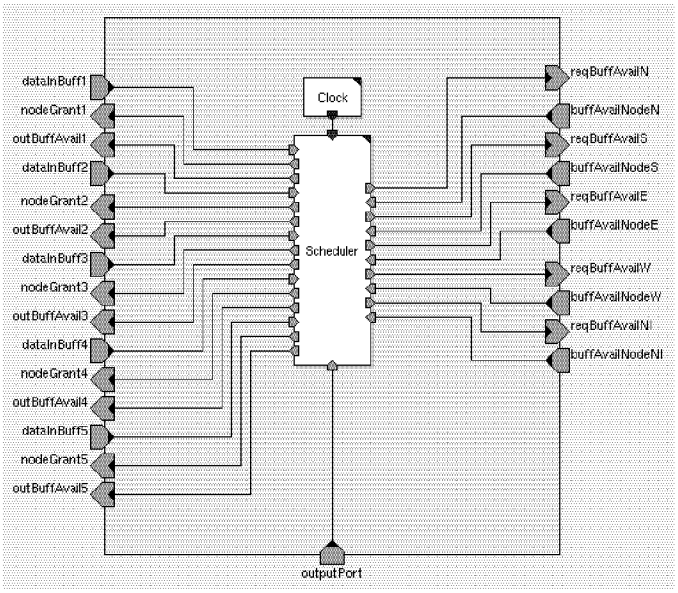


Fig. 6: MLDesigner implementation of the *Scheduler* class.

D. Router Class

A router, instantiated from the *Router* class, determines the output path and handles the actual data transfer on the implemented backbone. The router receives a certain number of data flits per unit time and is constrained by the data bandwidth for transmitting a fixed number of flits at the output. Thus, the *Router class* has been implemented in the SDF domain. A dimension-order routing protocol was implemented in the *Router* class for determining the output path. We have provided customization in routing algorithms as discussed earlier. Upon receipt of data, a router extracts the destination information and determines the physical output port for transmitting the data. This output port address is sent to the corresponding scheduler, which determines the availability of this port. Upon its availability, data flits are then forwarded to the corresponding output buffer for this port.

The type of routing algorithm is a parameter for the *Router* class. It can be set to: X-direction-first, Y-direction-first, and XY-random. In the X-direction-first algorithm, the data is routed to the X-direction first provided there is the possibility of the data to be routed to the Y-direction as well. The Y-direction-first algorithm works similarly. In the XY-random algorithm, if there is a possibility for the data to be routed to the X-direction as well as the Y-direction, the direction is

chosen in a randomized fashion with the same likelihood for the two choices.

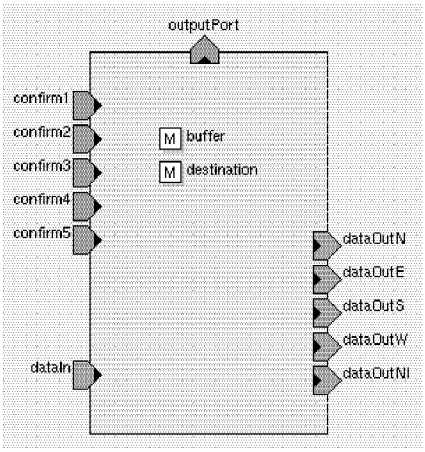


Fig. 7: MLDesigner implementation of the *Router* class.

E. OutputBuffer Class

An output buffer, instantiated from the *OutputBuffer* class, accepts the incoming flits from the router and forwards these flits to the input buffer of the next node. It is implemented in the form of two concurrently executing state machines. The received flits are stored in the output buffer memory. The input state machine accepts and stores the data flits while there is available memory in the output buffer. Upon the request of the scheduler by *reqBuffAvail*, the availability of buffer space in the output buffer is signaled by *buffAvail*. The output state machine senses the request for transmitting data from the output buffer of the next router via *outputBuffAvail* of that output buffer and forwards the data flit, if that signal was asserted.

The customizable parameters for the *OutputBuffer* class are the same as those for the *InputBuffer* class.

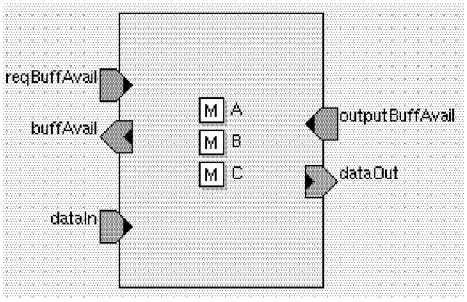


Fig. 8: MLDesigner implementation of the *OutputBuffer* class.

F. Consumer Class

A consumer, instantiated from the *Consumer* class, comprises a computing resource and a network interface. A consumer accepts data flits, strips off the header information, and forwards the remainder of the data to its internal computing resources. A consumer consumes data packets. Thus, as we did for the *Producer* class, we have implemented the *Consumer* class in the SDF domain.

IV. A NOC-BASED H.264 DECODER

The NoC architecture is effectively exploited when all the available cores on a multi-core architecture are kept busy by partitioning and mapping the multimedia application. At the application level, audio and video decoders can run on separate cores. However, to use all available cores, a multimedia application has to be partitioned into components and sub-components that are distributed among the available cores. The H.264 video compression algorithm has received significant interest from the industry and is expected to be used in a large number of mobile and embedded devices [8]. We developed a component model for a H.264 decoder that partitions the decoder based on functional components. In this approach, each component of the decoder is determined based on the well-defined subset of H.264 functions. The H.264 decoder is divided into the following key components: 1) entropy decoding; 2) inverse transform and quantization; 3) intra prediction; 4) inter prediction; and 5) deblocking. In addition to these processing components, the memory required can be treated as a separate component. Fig. 9 shows the components of a H.264 decoder.

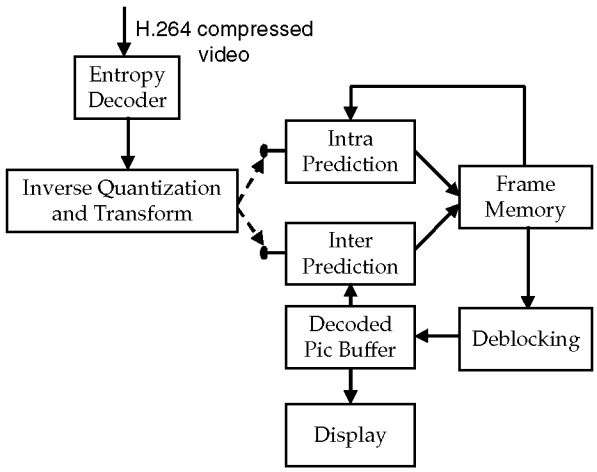


Fig. 9: Components of a H.264 decoder.

The memory components can be separated into the current frame memory (before the deblocking operation) and the decoded frame memory that is used for display and motion estimation. The access patterns to this memory and hence the traffic on the NoC core network varies depending on the macroblock (MB) coding modes.

The processing power required for each of these components varies and is also content-dependent. For example, when coded content contains more intra MBs, the intra component takes up the most resources. Keeping the NoC cores busy and balanced thus requires an understanding of the resources required by each component and the associated content dependencies. A resource estimation model for H.264 video decoding was presented in [9]. Based on the resource requirements for individual components, one or more instances of the decoder components are run on the NoC cores. While multiple inter prediction components may be necessary, a single intra prediction component is likely to be

sufficient for most applications. The work load distribution across the cores is managed by a scheduler, which is also responsible for synchronizing the flows among the cores.

Hereafter we briefly review our resource estimation methodology [9]. Fig. 10 shows the algorithm for extracting the communication dependency graphs using the VTune software tool. To extract the inter-component communication graphs, we first execute embedded C++ code for a Nokia H.264 decoder in the VTune environment. While executing this code, we must make sure that we are compiling the code for extracting performance parameters and call-graphs. As a result VTune generates a list of all the functions and the communication dependencies among these functions. As a next step, we group these functions into components. Thus, we get a profile of the total number of communications of a component and internal communications in that component. For executing this application on a NoC environment we do not need the internal communication details as this traffic does not get routed over the network. Therefore, we abstract out these details. As a next step we check the number of reads, writes, clock ticks, 1st level cache hits and cache misses, 2nd level cache hits and cache misses, and the bus activity. We further analyze the functional dependencies from the number of reads and writes and construct the final call-graphs for inter-component communications.

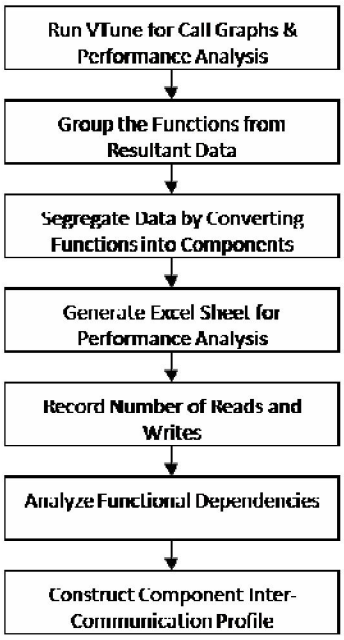


Fig. 10: Algorithm for extracting call-graphs using Intel VTune [9].

V. RESULTS

We implemented the NoC components on an FPGA to extract the total NoC area as well as the area information of each component. A system is defined in terms of its components. Using Impulse C, individual components of a system can be described as C functions. These functions are executed independently as processes. These processes can be declared as a hardware or software process. If the component is defined as a hardware process, then it gets implemented on the FPGA hardware. If defined as a software process, then the component is implemented in a soft-core processor. Impulse C code for all the processes in a system can be developed with any C++ compiler. A configuration function partitions the processes into running as either software or hardware. The configuration function also defines the way the process interfaces communicate, linking them all together as a whole system. The functionality of the prototype can be simulated and debugged, if needed, on a workstation before even being ported to the FPGA.

We designed the *InputBuffer* class as a smart buffer with a built-in scheduler, which forwards the data as per certain scheduling criteria. A buffer can be designed as being 32-bit or 64-bit wide. In the 64-bit buffer, the chip area will increase but the number of packets to be transmitted will be reduced to about half as compared to the 32-bit buffer. Therefore, it is likely that the latency of a system will be reduced by transmitting a larger packet. But it will require support for more parallel transmission lines (bus) in case of a parallel data transmission. Table 2 shows the number of gates taken by different buffer sizes for 32-bit and 64 bit buffers.

Table 2: Gate counts for different buffer sizes and bit widths.

Buffer Size	Gate Count for 32-Bit	Gate Count for 64-Bit
Size 1	11,427	16,733
Size 2	13,381	20,135
Size 3	14,510	22,207
Size 4	14,920	23,292
Size 5	15,526	24,594
Size 10	16,108	24,940

Note that the number of gates for a 64-bit buffer is not twice that of a 32-bit buffer. This is because every buffer has a scheduler for scheduling the data packets from the buffer. This scheduler occupies a fixed gate count that does not change with the buffer bit width.

Fig. 11 shows the number of gates needed to implement the NoC *Router* class, and the NoC *Scheduler* class with different scheduling algorithms: FCFS, RR, PB, and PBRR.

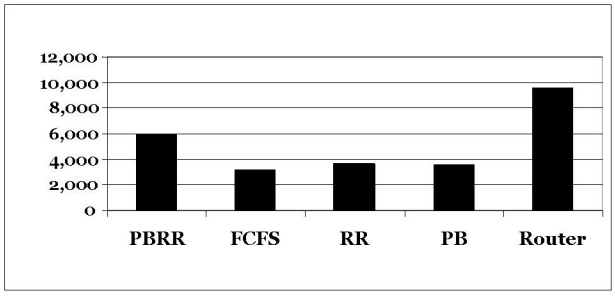


Fig. 11: Number of gates for NoC *Router* and *Scheduler* classes.

Using the resource estimation methodology of Section IV we abstracted the inter-component communication (see Fig. 12) for a *foreman* video bitstream using a H.264 Nokia decoder. The video had 15 frames per second with a resolution

of 176×144 pixels. The compressed bitstream had a bit rate of 15 Kb/sec. This traffic is modeled on a six-core NoC based-architecture. Arrows in Fig. 12 show the amount of traffic that flows from one component to another. We modeled this traffic on MLDesigner to calculate latencies offered by each H.264 component.

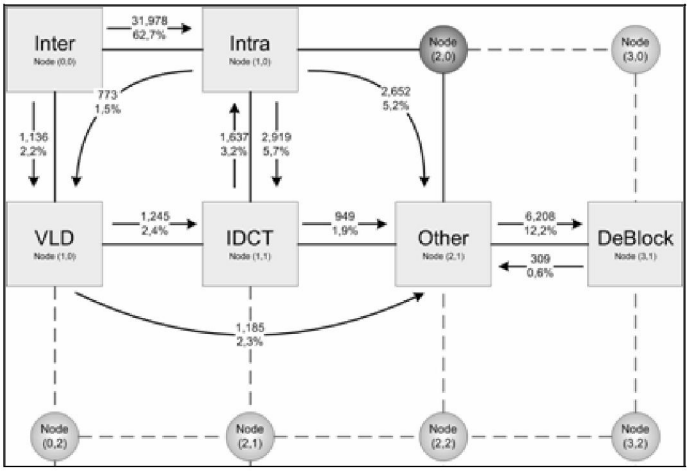


Fig. 12: Inter-component communication in the H.264 decoder.

We used a flit size of 64 bytes. Out of these 64 bytes, 8 bytes constitute the flit header used for sending the destination address, the source address and the priority level. We mapped the components with the highest intercommunication next to each other (single hop distance). This reduced the network latency. Fig. 13 shows the data packet arrival latency results for the *foreman* video (15 frames at 15 Kb/sec bitstream).

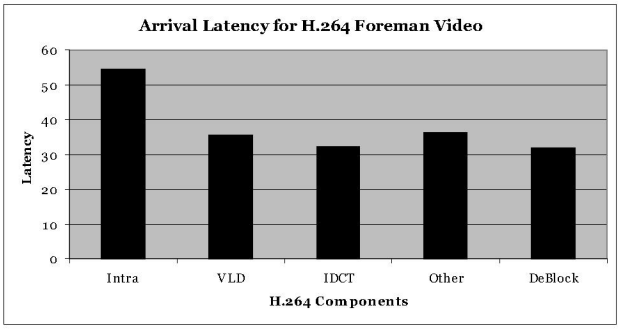


Fig. 13: Data packet arrival latency for H.264 *foreman* video.

VI. RELATED WORK

In [10], the authors used the OPNET network design and simulation environment to model an application-specific NoC and analyze its performance, the application being an H.264 HDTV decoder. The actual circuit design was done using SPICE. In [11], the authors mapped a H.264 decoder onto two networks-on-chip based on a mesh architecture and a fat-tree architecture, with the goal to maximize throughput while minimizing energy consumption, while in [12], they used tree-based NoC's; NS-2 was used for simulations. In [13], a MPEG-4 arbitrary-shaped video decoder is mapped onto a NoC, in order to demonstrate a hierarchical Quality-of-Service framework. In [14] the authors used the CASSE system-level tool to map a MPEG-4 decoder onto a multicore system-on-chip.

VI. CONCLUSION

We demonstrated a methodology for modeling and mapping applications onto a NoC architecture. A concurrency model for the NOC was first developed using FSP, and analyzed using LTSA. The NoC was then designed using the MLDesigner system-level modeling environment, where each component was represented as a class. We then abstracted area results for a 4×3 mesh-based NoC architecture. We mapped a Nokia H.264 decoder application onto the NoC and showed the latency results for a single abstracted bitstream.

REFERENCES

[1] W. J. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks", *IEEE International Conference on Design and Automation*, pp. 684-689, June 2001.

[2] L. Benini and G. De Micheli, "Networks on chip: a new SOC paradigm", *IEEE Computer*, vol. 35, no. 1, pp. 70-78, January 2002.

[3] A. Jantsch and H. Tenhunen. *Networks on Chip*, Kluwer Academic Publisher, 2003.

[4] G. Desoli and E. Filippi, "An outlook on evolution of mobile terminals: From monolithic to modular multi-radio, multi-application platforms", *IEEE Circuits and Systems Magazine*, vol. 6, no. 2, 2006, pp. 17-29.

[5] A. Agarwal and R. Shankar, "A concurrency model for NOC design methodology", *IEEE Conf. on High Performance Computing*, MIT, Sept. 2006.

[6] J. Magee and J. Kramer, *Concurrency: State Models and Java Programs*, 2nd edition, Wiley, 2006.

[7] P. Pratim Pande, C. Grecu, M. Jones, A. Ivanov, and R. Saleh, "Performance evaluation and design trade-offs for network-on-chip interconnect architectures", *IEEE Transactions on Computers,* vol. 54, no. 8, pp. 1025-1040, Aug. 2005.

[8] H. Kalva, "The H.264/AVC video coding standard," *IEEE Multimedia*, vol. 13, no. 4, Oct.-Dec. 2006, pp. 86-90.

[9] H. Kalva, R. Shankar, T. Patel, and C. Cruz, "Resource estimation methodology for multimedia applications," *Proc. of the 14th Annual ACM/SPIE* MMCN'07, San Jose, CA, Jan. 2007.

[10] J. Xu, W. Wolf, J. Henkel, and S. Chakradhar, "H. 264 HDTV decoder using application-specific networks-on-chip", *IEEE International Conf. on Multimedia and Expo*, July 2006, pp. 1508-1511.

[11] V.-D. Ngo, H.-N. Nguyen, and H.-W. Choi, "Realizing network on chip design of H.264 decoder based on throughput aware mapping", *First International Conf. on Communications and Electronics*, Oct. 2006, pp. 337-342.

[12] V.-D. Ngo, H.-W. Choi, Y. Bae, and H. Cho, "The optimized tree-based network on chip topologies for H.264 decoder design", *The 2006 International Conf. on Computer Engineering and Systems*, Nov. 2006, pp. 343-347.

[13] M. Pastrnak, P. H. N. de With, and J. van Meerbergen, "QoS concept for scalable MPEG-4 video object decoding on multimedia (NoC) chips", *IEEE Trans. on Consumer Electronics*, vol. 52, no. 4, Nov. 2006, pp. 1418-1426.

[14] L. García, G. M. Callico, D. Barreto, V. Reyes, T. Bautista, and A. Nunez, "Towards a configurable SoC MPEG-4 advanced simple profile decoder", *IET Comput. Digit. Tech.*, vol. 1, no. 5, Sept. 2007, pp. 451-460.