Hydra: an Energy-efficient and Reconfigurable Network Interface

M.D. van de Burgwal[1], G.J.M. Smit[1], G.K. Rauwerda[1] and P.M. Heysters[2]
{*m.d.vandeburgwal, g.j.m.smit, g.k.rauwerda*}*@utwente.nl, paul.heysters@recoresystems.com*

[1]Department of EEMCS, University of Twente, The Netherlands
[2]Recore Systems, The Netherlands

***Abstract*— In heterogeneous tiled System-on-Chip architectures a Network-on-Chip is used to transport messages between processing elements. A reconfigurable network interface is used to connect the processing elements to the Network-on-Chip, converting the messages between both domains. This paper introduces the Hydra: a network interface for the MONTIUM TP, a coarse-grained reconfigurable processor designed for DSP algorithms. We show that the Hydra is energy-efficient and provides the flexibility required to interface processing elements like the MONTIUM TP.**

***Index Terms*— Hydra, reconfigurable, network interface**

I. INTRODUCTION

Next generation multi-media appliances allow to communicate via wireless connections at any time and any place. Digital broadcast audio (DRM, DAB) and video (DVB) receivers decode high-bandwidth streams of data to reconstruct the original high quality signal, at the cost of computation intensive processing. For battery powered portable devices this may be a problem, as the energy source is limited. By optimizing the computational intensive kernels within an application, the energy consumption can be reduced significantly. Typically, the streaming multi-media applications mentioned have a regular communication scheme using connections that remain unchanged during several executions of the application. Since they have a strong temporal and spatial behaviour, these applications are quite suitable to be executed by a highly parallel System-on-Chip (SoC) platform. For efficiency reasons, SoCs are often designed as heterogeneous tiled architectures. These architectures consist of several types of tiles which are connected via a Network-on-Chip (NoC), as can be seen in Figure 1. Each tile has a processing element (a tile processor) and is connected to the NoC via a network interface. In this example SoC, several types of tile processors can be identified: application specific integrated circuits (ASIC), general purpose processors (GPP), fine-grained reconfigurable architectures (FPGA) and coarse-grained domain specific reconfigurable architectures (DSRA).

A. Networks-on-Chip

A lot of on-chip networks have been developed [1][2][3][4]. Roughly classified, there are two kinds of NoCs: circuit-switched and packet-switched networks. Generally speaking, circuit-switched networks consume a relatively small amount of energy since the routing information is configured in the routers [5]. Therefore, data arriving at a certain input channel is sent to a fixed output channel. When the routing has to be changed, the NoC has to be (partially) reconfigured. In packet-switched networks, the routers extract the routing information from the packets at runtime. This header extraction process requires a higher energy consumption for the routers, however, they do not need to be reconfigured when the communication pattern changes.

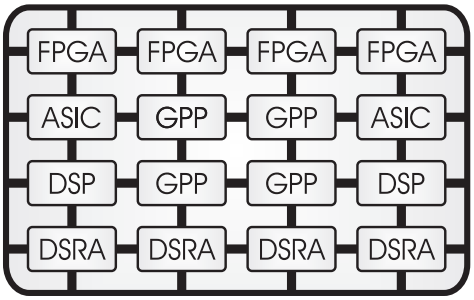


Fig. 1. SoC example with several different types of tiles

At the University of Twente two NoCs have been developed: a packet-switched version [6] and a circuit-switched version [7]. The circuit-switched NoC supports guaranteed throughput (GT) traffic only, while the packet-switched network supports guaranteed throughput as well as best effort (BE) traffic. Real-time and latency guarantees can be given for both types. However, the packet-switched network is more suitable for applications with changing communication patterns. For both NoCs the transported data words, called flits, are 18-bit: a 2-bit type field is used to provide control information alongside the 16-bit data field. The four types of flits that can be used are header (H), tail (T), data (D) and command (C). In the packet-switched NoC, the H flits are used to create a connection for all following C and D flits, while a T flit tears down the connection.

B. Tile Processors

Typical applications that are executed by tiled architectures are DSP algorithms like FFT, DCT and FIR filters. Such an application has to be partitioned in processes that can be executed by a tile processor. Each tile processor has a small local memory at its disposal to store input, output and

temporary data for such a process. On a SoC level, this can be seen as a distributed memory (with a typical storage size in the range of $10kB$ to $100kB$ per tile).

As a trade-off between energy-efficiency and flexibility, reconfigurable architectures turn out to be a good alternative. Coarse-grained reconfigurable architectures provide the flexibility needed for a lot of DSP algorithms, while the energy consumption is relatively small compared to the other architectures mentioned. The MONTIUM Tile Processor (TP) is a coarse-grained reconfigurable tile processor invented at the University of Twente [8] and is now further developed by Recore Systems [9]. The processing core consists of five ALUs, each of which has two local 1024×16-bit memories with private address generation units at its disposal. The ALUs are connected to these memories via ten global buses which can also be accessed by an external interface.

The MONTIUM TP has no internal communication controller. This paper proposes a network interface which supports the functionality required by the MONTIUM TP to communicate with our NoC.

C. Paper overview

In section III we give an overview of the operating environment and the requirements for the network interface. Using these requirements, we present a network interface architecture called Hydra in section IV. The implementation and synthesis results are shown in section V. Using some example DSP algorithms, the advantage of streaming-mode operation compared to block-mode operation is shown in section VI.

II. RELATED WORK

Remarkably, there is not much related work on network interfaces compared to the vast amount of papers on NoC and reconfigurable processors, although the design decisions in the NoC interface are important for the performance of the overall system [10]. Network interfaces are assumed to be straight-forward and are of little importance for NoC communication [3][11]. Therefore, they are often presented as a minor addition to a NoC.

Together with designing a NoC, Philips [4] has implemented a network interface that supports various on-chip communication protocols. The throughput and latency required by each of these protocols can be guaranteed by the Philips NoC (Æthereal) routers and the network interfaces. By designing the network interfaces in a modular way it is relatively easy to add support for new communication protocols. However, these network interfaces are meant to be a bridge between a NoC communication scheme and any other communication protocol instead of being a controller for the tile processor.

Another approach for implementing both control and communication tasks is to use a dedicated network for each of these tasks [12]. This is, however, quite different from a situation in which one NoC has to serve both communication and control tasks, since with two networks the guarantees that can be given about the latency and throughput are independent for each task.

III. REQUIREMENTS

The network interface is the medium between the NoC and the tile processor. The most important design decisions depend on the requirements enforced by either the NoC or the tile processor.

A. Operation mode

We have two mechanisms for transferring data between the NoC and the tile processor. Some processes require all the input data to be in the local memories before the execution can be started. This operation mode is called *block-mode*. Typically, a block-mode operation is done in three stages: the input data is loaded into the local memories, the process is executed and the output data is fetched from the local memories. During the data transfers, the tile processor is halted to make sure the execution is not started until all data is valid. In this operation mode, the network interface acts as a master for the tile processor.

Some tile processors support reading input data and writing output data while they are processing, using the network interface as a slave for performing data transfers. This operation mode is called *streaming-mode*. Typically, during the execution of a streaming-mode process connections for the input data and output data remain open. Therefore, the output data does not need to be packaged. This is an advantage for both the sender as well as for the receiver, because the packaging is an overhead for both. Since the packaging is not done in streaming-mode, the latency of communication streams is small compared to block-mode.

Whether block-mode or streaming-mode is used, is determined by the application programmer and strongly depends on the characteristics of the application process. When the application operates in block-mode, no computation and communication occurs at the same time. This increases the ease of programming at process level, but gives some overhead at application level. For streaming-mode applications the programmer has to carefully plan how and when the communication takes place. This can be hard, especially when the results are not ordered linearly or when the ordering depends on one or more parameters.

B. Communication to Computation ratio

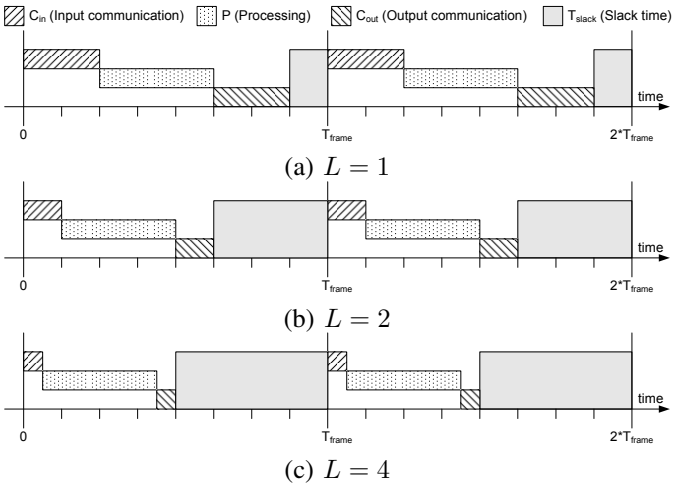
The ratio of communication time to computation time is called the Communication to Computation (C/C) ratio:

$$C/C = \frac{T_{comm}}{T_{comp}} \tag{1}$$

where T_{comm} is the communication time needed to transfer both input and output data while the processor is not processing and T_{comp} is the computation time required to process the data.

Figures 2 and 3 illustrate the relation between the operation mode and the communication and communication times. Two entire process executions (called frames) are shown for both modes. The load L of a data transfer is defined as the number of words that are transferred per cycle. The dashed blocks in

the upper part of each picture represent input data transfers (indicated by C_{in}), the lower dashed blocks represent output data transfers (indicated by C_{out}) and the dotted blocks in the middle part show the processing (indicated by P). The gray shaded area is the remaining time in a frame, called slack time (T_{slack}).



(a) $L = 1$

(b) $L = 2$

(c) $L = 4$

Fig. 2. Block-mode operation

From these figures we define:

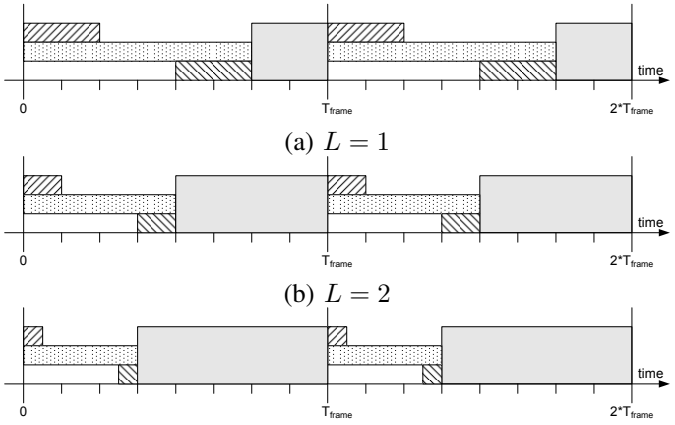
$$T_{comp} = time\,(P) \quad (2)$$

$$T_{comm} = time\,([C_{in} \cup C_{out}] \setminus P) \quad (3)$$

where $time\,(t)$ denotes the time required for t.

Block-mode processes can only be started when all input data is received. This means there is no overlap in time between the communication and computation and, therefore, a change in the communication does not influence T_{comp}. However, delays caused during the communication (for example due to blocking in the network) do influence T_{comm}.

Figure 3 shows the same process in streaming-mode operation. Note that, when $L = 4$, T_{comp} is equal for both modes.



(a) $L = 1$

(b) $L = 2$

(c) $L = 4$

Fig. 3. Streaming-mode operation

Streaming-mode processes behave slightly different. Since communication and computation are done in parallel as much as possible, T_{comm} is reduced significantly. Note that this also makes the total computation time T_{comp} dependent on delays caused during the communication. While T_{comp} is comparable for both modes, T_{comm} for block-mode processes is a lot bigger than for the streaming-mode processes. Therefore, the C/C ratio for a block-mode process is considerable higher than the C/C ratio for a streaming-mode version of this process. Figure 2 shows that a high parallelism in data transfers can be meritable, particularly for block-mode operation. The C/C ratio can be decreased considerably when L is increased.

C. Driving model

In block-mode operation, tiles are control driven. The process execution is started when an external source has prepared the input samples and gives a start command. As long as no command is received, the process halted. Using this operation mode, communication and computation are fully separated. This may be useful in case the input data is provided on an irregular basis (for example non-linear addresses, variable input size, continuously changing parameters).

In streaming-mode operation, the system is data driven. A process on a tile is started as soon as enough samples are available. This may be even before the last sample has been received, providing a parallel execution of communication and computation. For streaming-mode processes the usage and order of input data has to be unambiguous. This implies that such a process needs to know what to do with the next input data: a reordering may have to be done before the data can be used. Clearly, this also holds for the output data.

D. Real-time guarantees

After the decomposition of an application, the processes are mapped on tile processors. Typically, each tile processor is capable of executing only a few processes. In order to have the entire application meet its real-time constraints, all processes need to meet these real-time constraints. Therefore, guarantees need to be given to be sure none of the processes can endanger the real-time execution behavior of the application. Since we have a NoC that supports guaranteed throughput at a fine-grained level, the NoC is not a limitation for the real-time guarantees that are required for the application. The delay caused by communication via the network interface needs to be relatively small compared to the communication time in order to satisfy the real-time guarantees of the process.

E. Flexibility and Energy-efficiency

Typically, each of the processors in a tiled architecture runs at a different clock speed. The clock frequency of the NoC (f_{NoC}) is 100 MHz and the frequency of a tile processor (f_{TP}) is derived from this clock by a clock divider that uses a parameter n:

$$f_{TP}\,(n) = \frac{f_{NoC}}{2^n}, \qquad n \in \{0, 1, 2, 3, 4\} \quad (4)$$

Data transfers crossing these clock boundaries have to be synchronized. For a data transfer between the NoC and a tile processor using L channels of B bits, we define the bandwidth BW (in bits per second) as:

$$\begin{aligned} BW &= L \cdot B \cdot \min\left(f_{NoC}, f_{TP}\right) \\ &= L \cdot B \cdot f_{TP} \end{aligned} \tag{5}$$

As B is restricted by the hardware design, it can be seen that the bandwidth only depends on the load of the data transfer and on the selected f_{TP}.

It is likely that, at a certain moment in time, the tile processor has finished its computation before new data samples have arrived. To save energy, the network interface halts the tile processor until these new samples arrive.

In [13] an approximation is given for static power consumption:

$$P = \alpha \cdot C \cdot V_{dd}^2 \cdot f \tag{6}$$

where α is the switching activity, C is the capacitance, V_{dd} is the supply voltage and f is the frequency.

The clock net has the highest α of all parts of an ASIC and the C is large since all synchronous components in an ASIC are driven by the clock, hence the clock distribution net consumes a considerable part of the power. When running a processor at a lower clock frequency, it can also be run at a lower core voltage. Combined, this can give a significant reduction in the power consumption. Therefore, it is useful to slow down or shut down the tile clock whenever possible.

Figure 2 shows that, when L is incremented, the communication time decreases and thus more slack time is left. Assuming that, for the streaming application domain, T_{comm} and T_{comp} remain constant for a reasonable time and that no blocking occurs during the input or output data transfers, some estimations can be made about the required clock frequency. If $T_{slack} > \frac{T_{frame}}{2}$, the clock divider setting n can be increased without consequences. In fact, the number of increments of n can be estimated as follows:

$$n = \min\left(\left\lfloor \log_2 \frac{T_{frame}}{T_{comm} + T_{comp}} \right\rfloor, 4\right) \tag{7}$$

From this estimation we can conclude that a smaller C/C ratio contributes to less energy consumption. However, this choice for n also influences T_{comm}. Obviously, the delay (in NoC clock cycles) increases when the tile processor clock is slowed down. This may give communication problems for the tile processor that has to receive the output data. Therefore, n has to be chosen carefully.

IV. The Hydra architecture

In the 4S project we are designing a verification platform which is based on a System-on-Chip with several devices. One of these devices is a reconfigurable fabric containing four Montium TPs and a circuit-switched NoC. We expect to get a silicon design of this platform in the end of 2006. As an implementation of a network interface for connecting these tile processors to our NoC, we have developed the Hydra[1] architecture (see Figure 4). Its functionality embraces two tasks: controlling the Montium TP and providing a communication mechanism between the Montium TP and a NoC.

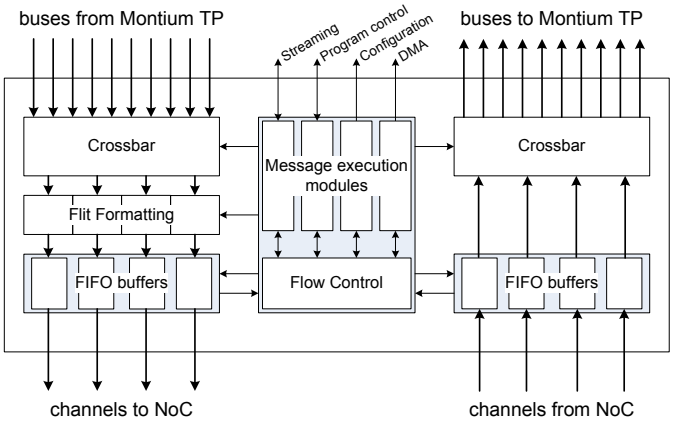


Fig. 4. Hydra structure

A. Tile Processor control

The core of the verification platform is an ARM9 processor, which is used as a centralized configuration manager that controls the other devices on the SoC. The ARM controls the Montium TP by communicating with the Hydra, using a lightweight message protocol. The Hydra, on its turn, controls the program execution and communication for the Montium TP. Table I shows the messages available within this protocol and their encoding.

TABLE I
Message protocol

Encoding	Message	Format
000	Configuration	$C_{cfg}\left[H\,[D]^+\right]^+ T$
001	DMA load	$C_{load}\left[H\,[D]^+\right]^+ T$
010	DMA retrieve	$C_{retr}\,[HD]^+ T$
011	Get status	C_{status}
100	Run	C_{run}
101	Wait	C_{wait}
110	Reset	C_{rst}

The format field gives the order in which the flits are used, using a regular expression notation. Each message starts with a C flit (with the encoded command, as can be found in the leftmost column in Table I) and additional information is added using H flit (for address selection) and D flits (for data or parameters). A T flit indicates the end of the message. The messages are executed as soon as a C flit has been received, interrupting the execution of previously received messages. As the Hydra executes messages on a flit-by-flit base, it will not reject incorrectly formatted messages but it will try to execute the parts of the message that are formatted correctly.

[1]The Hydra was a monster from the Greek mythology with many heads, symbolizing the parallelism of our network interface

The received messages are parsed and executed by dedicated hardware modules (depicted in the center of Figure 4). Using this modular approach the message protocol can be adapted easily, requiring only little effort for altering the hardware design. As there are no separate data and control networks, the control messages and data arrive at the input channels in a mixed fashion. The flow control component guarantees that the Hydra will not stop reading the input channels, to avoid deadlocks in message handling when input data and control arrive at the same time.

Figure 5 shows an example *Configuration* message to clarify the formatting as mentioned in Table I as well as the usage of the flit payload.

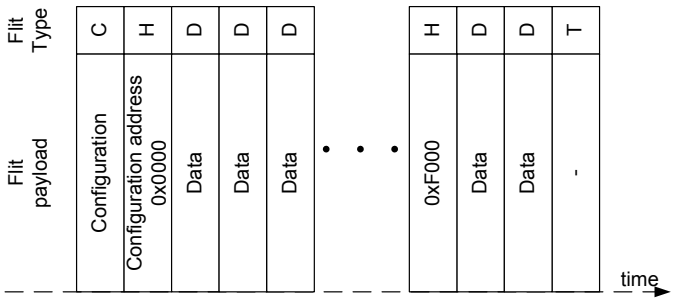


Fig. 5. *Configuration* message

When a *Configuration* or *DMA load* message is executed, the accompanying hardware module takes care of the addressing information. Each H flit that is received is used as an offset address and, for *DMA* messages, for the selection of DMA entities within the MONTIUM TP. For the following D flits the addresses are incremented automatically. Therefore, the address overhead in messages is reduced as much as possible.

Additional control functionality includes *Get status*, *Wait* and *Reset* messages. The *Get status* message is used fordebugging purposes and can be used for synchronization with other processors. The *Wait* message halts the MONTIUM TP until a new message is sent. If needed, the MONTIUM TP can be reset using a *Reset* message.

Typically, for block-mode operations the message protocol is used in the following way. First, a *Configuration* message is sent to configure the MONTIUM TP. After that, the input samples are loaded with a *DMA load* message. When all input samples are loaded, a *Run* message is used to start the process that was configured in the MONTIUM TP. Subsequently, with a *DMA retrieve* message the results are read from the MONTIUM TP memories. Now, the cycle of actions (*Configuration*, *DMA load*, *Run*, *DMA retrieve*) can be executed again.

As mentioned before, streaming applications tend to be data driven. Typically, the tile processor and network interface are initialized with a *Configuration* message. Next, the tile processor is activated with a *Run* message. The configured process manages the input and output streaming of data and may give an interrupt when it has finished an operation cycle.

B. Communication mechanism

Being an interface component basically implies moving data between two domains. Since, in our architecture, it is possible that these two domains are running at different clock speeds, synchronization between these domains is inevitable. A common used solution is to use of FIFO buffers (see lower left and right components in the figure). Interference between channels is not desired; hence, for each channel a dedicated FIFO is used to be sure that flows from two channels do not influence each other. Looking at the difference between the clock frequencies, it would not be useful to introduce buffers with a capacity of a large number of flits per channel while the tile processor is executing at half clock speed compared to the NoC. On average, storing up to four flits per channel turns out to be reasonable for providing enough bandwidth.

For bandwidth intensive processes, a high throughput with minimal latency is one of the key requirements. Therefore, the data path contains a crossbar which supports maximal connectivity (connections between all outputs of the NoC to the tile processor and from the tile processor to all inputs of the NoC). Together with a FIFO buffer for each of these inputs and outputs, the latency is reduced to only one clock cycle (of the tile processor's clock frequency). In Figure 4, the data path is the leftmost part (communication from the MONTIUM TP to the NoC) and the rightmost part (communication from the NoC to the MONTIUM TP).

Another communication-related task for the Hydra is the formatting of output data, which is done just before the data is sent to the NoC (shown in the middle of the leftmost part of Figure 4). Memory contents, requested with a *DMA retrieve* message, are always formatted as D flits. For streaming-mode processes the MONTIUM TP determines the formatting that has to be applied by the Hydra, by selecting a decoder instruction in the Hydra which contains information about the flit format for each of the outgoing channels.

Additionally, the MONTIUM TP can also address small configurable ROMs in the Hydra, which can store up to four flits per output channel. Using these ROMs, the MONTIUM TP programmer has some flexibility to implement mechanisms for synchronization, command generation, et cetera. The decoder and ROMs form the reconfigurable part of the Hydra and its configuration address space complements the configuration address space of the MONTIUM TP. Therefore, the Hydra can configure either the MONTIUM TP or itself when it receives a *Configuration* message, depending on the selected address. With this mechanism, streaming-mode applications can be bootstrapped for the initialization.

When mapping an application on the MONTIUM TP the programmer does not have to deal with the implementation of communication mechanisms. The MONTIUM TP compiler tool abstracts the physical input channels and output channels to logical sources and destinations, while the hardware layer halts the MONTIUM TP program when data transfers can not be performed (for example when input buffers are empty or when output buffers are full).

V. Implementation and results

Before implementing an RTL model, the message protocol was developed based on a list of functionalities that were required. Since a large bandwidth was required for both block-mode and streaming-mode operation, the data path (buffering and crossbar) was designed as a basis for the RTL model in VHDL. Next, the flow control block was implemented to control the data path. For the execution of messages we implemented the message execution modules. Finally, the flit formatting was added.

A. Synthesis results

The VHDL model was synthesized in 0.13 μm technology, while the clock frequency was constrained to 200 MHz. With this constraint the area was $19k$ gates (0.106 mm^2), which is about 5% of the area of the Montium TP. Table II shows how this area is distributed over the several components:

TABLE II
Area distribution

Component	#gates	Area (mm^2)	Area (%)
Crossbar	1810	0.010	9.5
Flit formatting	3695	0.021	19.3
Flow control	3893	0.022	20.4
Buffering	7920	0.044	41.5
Message execution	1788	0.010	9.4
Total	19106	0.106	

A large part of the total area is needed for the input and output buffering. The flow control and flit formatting each contribute 20% of the total area, which is caused by the storage ROMs and the instruction decoder. For the other components (the crossbar and the message execution modules) the area is related to the multiplexers in the data path and the control logic around it.

VI. Case study: Implementation of DSP algorithms

We have successfully mapped parts of several wireless applications to the Montium TP architecture: DRM, HiperLAN/2, Bluetooth and UMTS. This section gives, for each of these applications, an example of one or more kernels.

A. DRM and HiperLAN/2

The kernel operation in DRM [14] and HiperLAN/2 [15] is the inverse OFDM procedure, which primarily consists of a FFT. This transform is responsible for roughly half of all computations of the entire baseband processing. For DRM mode B, every 20 ms an FFT-256 has to be executed on an OFDM symbol, while HiperLAN/2 requires an FFT-64 to transform one symbol every 4 μs. An FFT-N operates on a data set of N 16-bit complex samples and the results are N 16-bit complex values. The total number of computation cycles for a mapping of an FFT-N (where N is a power of two) on the Montium TP architecture is $\left(\frac{N}{2}+2\right)\log_2(N)$ [16]. For the block-mode version of the FFT, the total number of tile clock cycles required to process one FFT symbol is:

$$\begin{aligned} T_{block} &= T_{comm} + T_{comp} \\ &= 2 \cdot \frac{2N}{L} + \left(\frac{N}{2}+2\right)\log_2(N) \\ &= \frac{4N}{L} + \left(\frac{N}{2}+2\right)\log_2(N) \end{aligned} \quad (8)$$

Our mapping for streaming-mode operation is capable of reading the input data and writing the results during the computation:

$$\begin{aligned} T_{streaming} &= T_{comm} + T_{comp} \\ &= \left(\frac{N}{2}+2\right)\log_2(N) \end{aligned} \quad (9)$$

In the streaming-mode case, the output results are produced in a bit-reversed order and need to be reordered [16]. Such a reordering can be done in the Montium TP in $\frac{N}{2}$ more clock cycles. Therefore, in the HiperLAN/2 case the worst-case C/C ratio (with $L = 1$) for the block-mode operated FFT-64 is $\frac{256}{204} = 1.25$ while for the streaming-mode it is only $\frac{32}{204} = 0.16$ (considering the reordering process to be a communication issue).

Theoretically, the total time required for one FFT-64 should be less than 4 μs, which means that the clock frequency for a streaming-mode FFT-64 would be:

$$f_{TP} = \frac{T_{streaming}}{4\mu s} = \frac{236\ cycles}{4 \cdot 10^{-6}\ s} = 59MHz \quad (10)$$

and for the block-mode operation:

$$f_{TP} = \frac{T_{block}}{4\mu s} = \frac{268\ cycles}{4 \cdot 10^{-6}\ s} = 67MHz \quad (11)$$

assuming that $L = 4$ equals the bandwidth used by the streaming-mode version. When L is decreased to 1, the minimal required frequency for the block-mode FFT-64 reaches 115 MHz. As this clock frequency is not available (see equation 4), block-mode operation is not always feasible.

B. Bluetooth and UMTS

In Bluetooth [15], the core computation is a FIR filter that requires the most computations. Similar to the FFT operation, we mapped the FIR algorithm to our architecture for both operation modes. For an M-taps FIR filter and an input data set of size N, the total clock cycles needed by the block-mode filter is:

$$\begin{aligned} T_{block} &= T_{comm} + T_{comp} \\ &= 2 \cdot \frac{N}{L} + N \cdot \left\lceil \frac{M}{5} \right\rceil + 1 \end{aligned} \quad (12)$$

The streaming-mode FIR filter reflects the advantages of streaming-mode processes over their block-mode counterparts:

$$\begin{aligned} T_{streaming} &= T_{comm} + T_{comp} \\ &= N \cdot \left\lceil \frac{M}{5} \right\rceil + 1 \end{aligned} \quad (13)$$

For a 5-tap FIR filter and an input data size of 1024 samples, the worst-case C/C ratio for the block-mode version is $\frac{2048}{1025} = 2.00$ (again, $L = 1$), whereas the streaming-mode version has a C/C ratio of $\frac{1}{1025} = 0.00$.

Additionally to a FIR filter, UMTS uses a RAKE receiver [17]. The RAKE receiver is not computational intensive, but it does require a lot of I/O transactions in parallel ($L = 4$). Every other clock cycle, the RAKE receiver reads two complex data samples (fingers) and processes these in two clock cycles. Meanwhile, for every four fingers, it receives a complex coefficient (scrambling code) which is used during four clock cycles, to process the four data samples. After four of these iterations, the results are summed and streamed out. This process can be done at a clock frequency of about 10 MHz. Since this bandwidth can be provided by the Hydra, the tile clock can be slowed down to 12.5 MHz (by setting the clock divider $n = 3$).

VII. Conclusion

In this paper we presented the design and implementation of the Hydra, an energy-efficient and reconfigurable network interface for the MONTIUM TP and a Network-on-Chip. It supports both block-mode and streaming-mode data transfers and is controlled by a lightweight message protocol. The bandwidth provided by the Hydra is only limited by the chosen NoC, as its data path provides a maximum connectivity between all input channels from the NoC to the MONTIUM TP and vice versa. If constrained to operate at a maximum clock frequency of 200 MHz, the Hydra has a total area of $19k$ gates or about 0.106 mm^2 in 0.13 μm ASIC technology.

Generally, not much attention is paid to network interfaces. Communication patterns are considered being NoC dependent, while the program execution and communication in a processor are assumed to be independent. Bandwidth turns out to be a key requirement for the design of a network interface.

With an example we show that a high level of parallelism during data transfers results in a short communication time. When the communication time is decreased, the tile clock can be slowed down while the tile processor is still capable of performing enough computations to finish the process within its required period. This can contribute to a significant reduction in energy consumption.

Using the Communication to Computation ratio C/C for some typical DSP algorithms that have been mapped on the MONTIUM TP, we showed the advantages of streaming-mode communication to their block-mode counterparts. The FIR filter showed a tremendous difference in the communication times of the block-mode and streaming-mode operation. Although the block-mode and streaming-mode FFT-64 seem to be similar, the block-mode FFT-64 can not be executed fast enough in a worst case scenario.

The design of a lightweight message protocol developed to support the functions that are required by the processor contributes to a straight-forward implementation of the network interface.

Although the Hydra is presented as a MONTIUM TP specific network interface, the basic mechanisms are reusable for the design of a network interface for any processor architecture.

Acknowledgement

This research is conducted within the Smart Chips for Smart Surroundings project (IST-001908) supported by the Sixth Framework Programme of the European Community.

References

[1] W. J. Dally, B. Towles, *Route Packets, Not Wires: On-Chip Interconnection Networks*. DAC, June 2001, pp. 684-689

[2] J. Liang, S. Swaminathan, R. Tessier, *aSOC: A Scalable, Single-Chip Communications Architecture*. Proceedings of the IEEE International Conference on Parallel Architectures and Compilation Techniques, October 2000, Philadelphia, USA

[3] S. Kumar, A. Jantsch, J. Soininen, M. Forsell, M. Millberg, J. Öberg, K. Tiensyrjä, A. Hermani, *A Network on Chip Architecture and Design Methodology*. Proceedings of ISVLSI'02, 2002

[4] J. Dielissen, A. Rădulescu, K. Goossens, E. Rijpkema, *Concepts and Implementation of the Philips Network-on-Chip*. Proceedings of IP-Based SoC Design, November 2003, Grenoble, France

[5] P.T. Wolkotte, G.J.M. Smit, G.K. Rauwerda, L.T. Smit, *An Energy-Efficient Reconfigurable Circuit Switched Network-on-Chip*. Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - 12th Reconfigurable Architecture Workshop (RAW 2005), p. 155a, ISBN 0-7695-2312-9, April 4-5, 2005, Denver, USA

[6] N.K. Kavaldjiev, G.J.M. Smit, P.G. Jansen, *A Virtual Channel Router for On-chip Networks*. Proceedings of IEEE International SOC Conference, Sep. 2004, pp. 289-293, Santa Clara, USA

[7] P.T. Wolkotte, G.J.M. Smit, N.K. Kavaldjiev, *Energy Model of Networks-on-Chip and a Bus*. Proceedings of the International Symposium on System-on-Chip (SoC 2005), ISBN 0-7803-9294-9, pp. 82-85, November 15-17, 2005, Tampere, Finland

[8] P.M. Heysters, *Coarse-Grained Reconfigurable Processors: Flexibility meets Efficiency*. Ph.D.-thesis, ISBN 90-365-2076-2, ISSN 1381-3617, November 2004, Enschede, The Netherlands

[9] *http://www.recoresystems.com*. 2005

[10] M.H. Wiggers, N.K. Kavaldjiev, G.J.M. Smit, P.G. Jansen, *Architecture Design Space Exploration for Streaming Applications Through Timing Analysis*. Proceedings of Communicating Process Architectures (WoTUG-28), pp. 219-233, ISBN 1-58603-561-4, Published by IOS Press, Amsterdam, September, 2005, Eindhoven, The Netherlands,

[11] L. Ost, A. Mello, J. Palma, F. Moraes, N. Calazans, *MAIA - A Framework for Networks on Chip Generation and Verification*. ASP-DAC, Jan. 05

[12] T. Marescaux, J.Y. Mignolet, A. Bartic, W. Moffat, D. Verkest, S. Vernalde, R. Lauwereins, *Networks on Chip as Hardware Components of an OS for Reconfigurable Systems*. In: Field-Programmable Logic and Applications (FPL'03), September 2003.

[13] E.F. Weglarz, K.K. Saluja, M.H. Lipasti, *Minimizing energy consumption for high-performance processing*. Proceedings of ASP-DAC 2002. 7th Asia and South Pacific and the 15th International Conference on VLSI Design, pp. 199-204, Januari 7-11, 2002

[14] P.T. Wolkotte, G.J.M. Smit, L.T. Smit, *Partitioning of a DRM receiver*. Proceedings of the 9th International OFDM-Workshop, pp. 299-304, September 15-16, 2004, Dresden, Germany

[15] G.K. Rauwerda, G.J.M. Smit, L.F.W. van Hoesel, P.M. Heysters, *Mapping Wireless Communication Algorithms to a Reconfigurable Architecture*. Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'03), pp. 242-251, ISBN 1-932415-05-X, June 23-26, 2003, Las Vegas, USA

[16] P.M. Heysters and G.J.M. Smit, *Mapping of DSP Algorithms on the Montium Architecture*. Proceedings of RAW 2003, April 2003, Nice, France

[17] G.K. Rauwerda, G.J.M. Smit, *Implementation of a Flexible RAKE Receiver in Heterogeneous Reconfigurable Hardware*. Proceedings of the 2004 IEEE International Conference on Field-Programmable Technology, pp. 437-440, ISBN 0-7803-8651-5, ISBN 0-7803-8652-3, December 6-8, 2004, Brisbane, Australia