2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)

μDMA: An Autonomous I/O Subsystem For IoT End-Nodes

Antonio Pullini*, Davide Rossi†, Germain Haugou*, and Luca Benini*†
*Integrated Systems Laboratory, ETH Zürich, Gloriastr. 35, 8092 Zurich, Switzerland
†DEI, University of Bologna, Via Risorgimento 2, 40136 Bologna, Italy

***Abstract*—The Internet of Things revolution requires long-battery-lifetime, autonomous end-nodes capable of probing the environment from multiple sensors and transmit it wirelessly after data-fusion, recognition, and classification. Duty-cycling is a well-known approach to extend battery lifetime: it allows to keep the hardware resources of the micro-controller implementing the end-node (MCUs) in sleep mode for most of the time, and activate them on-demand only during data acquisition or processing phases. To this end, most advanced MCUs feature autonomous I/O subsystems able to acquire data from multiple sensors when the CPU is in idle state. However, in these traditional I/O subsystems the interconnect is shared with the processing resources of the system, both converging in a single-port system memory. Moreover, both I/O and the data processing subsystems stand in some power domain. In this work we overcome the bandwidth and power-management limitations of current MCU's I/O architectures introducing an autonomous I/O subsystem coupling an I/O DMA tightly-coupled with a multi-banked system memory controlled by a tiny CPU, which stands on a dedicated power domain. The proposed architecture achieves a transfer efficiency of 84% when considering only data transfers, and 53% if we consider also the overhead of the runtime running on the controlling processor, reducing the operating frequency of the I/O subsystem by up to 2.2x with respect to traditional MCU architectures.**

***Index Terms*—I/O subsystem, energy efficiency, area optimized, ultra-low-power.**

I. Introduction

A new generation of micro-systems periodically probing the environment from a wide variety of sensors and transmitting "compressed information" such as classes, signatures or events to the cloud after some processing, recognition and classification has emerged in the last few years [1]. Collectively referred to as the end-nodes of the Internet of Things, these devices share the need for high processing capabilities and extreme low-power for long-life autonomous operation. These micro-systems are usually built around simple micro-controllers (MCUs), a variety of sensors which depends on the specific applications targeted, a wireless transceiver, and batteries or harvesters.

To deal with these tight power constraints, data acquisition and processing hardware should be active only for the necessary amount of time required to probe, process and transmit the information, while it should be idle for the rest of the time. This mechanism is usually referred to as duty-cycling, and it is widely adopted in power-constrained MCU applications [2]. Figure 1(a) shows the typical profile of a generic IoT application. For most of the time, the system is idle, waiting for an event that can either be generated by an external sensor or by an internal timer. With reference to Figure 1 we can differentiate two different flavours of events. The first one, referred to as *data event* is triggered when a new data or data stream has to be acquired from the sensing subsystem. The second one, called *processing event* is triggered when enough data is acquired and stored into the MCU memory. Depending on the specific application, the two events can be overlapped (i.e. the processing starts as soon as a new data chunk is being acquired) or not, as in a more generic scenario.

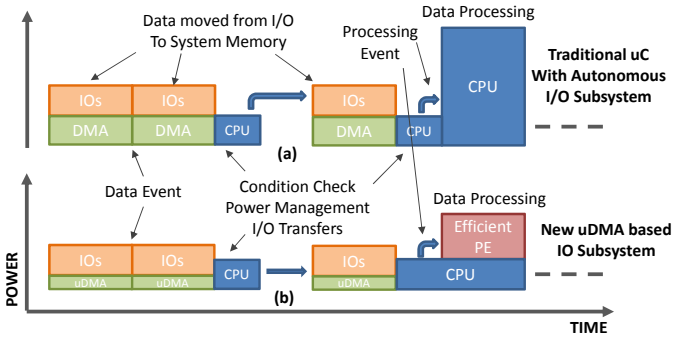

Fig. 1: Profile of a generic IoT application

To minimize power consumption during the different phases, today MCUs feature several power modes, where depending on the circumstances the system can be put in a low-power sleep mode, which can be either state retentive or not. These power states may have a very different duration, duty cycles and performance requirements. For example if an application has to deal with frequent wake-ups, paying at each wake-up the price of saving and restoring its full state could be highly inefficient and a power mode with higher sleep current but with state retention would be much more valuable. One approach to optimize the power consumption of microcontrollers is to put the different subsystems of the platforms, such as processor, the I/O subsystem and memories in different clock and power domains, where each domain can be clock gated or power gated independently.

Peripheral subsystems of state-of-the-art off-the-shelf MCUs usually include an extensive set of peripherals needed to connect to the wide variety of sensors required by IoT applications. Although MCUs traditionally feature low bandwidth interfaces like UART, I2C, I2S or standard SPI, a new generation of higher bandwidth peripherals such as USB, camera and hyperbus [3] interfaces has been recently introduced. This can lead to an aggregate bandwidth up to 2Gbit/s. While in traditional MCUs I/O peripherals were constantly supervised by the CPU, responsible for handling events and regulate data transfer to/from memory, leading

978-1-5090-6462-5/17/$31.00 ©2017 IEEE

to a very poor utilization of resources, recent MCUs have increased the complexity of the I/O subsystem to support more power modes to be able to selectively activate on-demand the required peripherals only when needed. Further optimizations have been achieved by providing to the I/O subsystem "embedded intelligence" capabilities by means of programmable inter-peripherals connectivity and smart DMAs. In some of these platforms, this approach is pushed to the limit, deploying a tiny microprocessor (e.g. Cortex M0) for managing data acquisition, while exploiting a more powerful and power hungry CPU (e.g. CortexM4) for the processing phases [4].

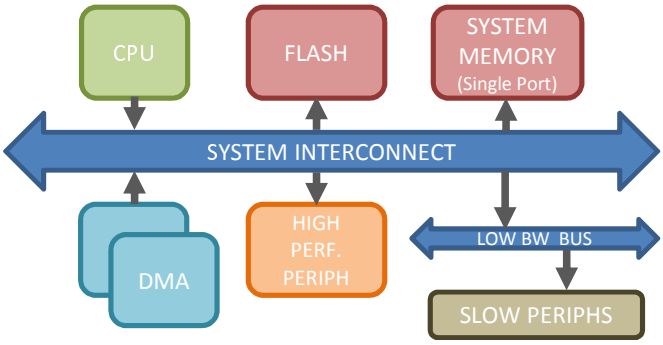

Fig. 2: Generic architecture of traditional MCUs

In this scenario, the I/O subsystem is waken-up during frequent data acquisition phases, while the more power-hungry computation subsystem (e.g. the micro-processor) can sleep (clock gating) for most of the time, being activated during data-processing phases only. However, the totality of state-of-the-art MCUs has the peripheral subsystem connected to the global system bus, which converge on a single ported SRAM memory(see Figure 2). For this reason, the overhead caused by the contention on the access to the memory subsystem, together with the bus arbitration overheads (e.g. AHB), cause a poor utilization of the resources, and form a bottleneck that limits the bandwidth sustainable by the peripheral-bus-memory subsystem.

In this work we propose an autonomous I/O subsystem for next generation end-nodes of the IoT able to sustain the bandwidth of most advanced peripherals. The main contributions of this work include:

- Improve the theoretical limits for the I/O data transfer exploiting a multi-banked, multi-ported system memory
- Dedicate lightweight DMA channels to each peripheral data stream and connect the peripherals memory subsystem, avoiding to use the system interconnect
- Use a tiny processor dedicated to the control of I/O transfers and micro manage the peripherals (e.g. filesystem handling, protocol stacks)
- Have a dedicated power domain for the proposed I/O subsystem separated from the computational resources of the system to enable aggressive power management.

The proposed approach increases by 1.7x the bandwidth (at fixed frequency) when only I/O to memory transfer is considered and up to 2.2x when considering the offload of the DMA transfers plus the contention on the memory system due to the program execution of the controlling processor. On the other hand, for a given aggregate bandwidth requirement (constrained by specific applications), the proposed approach allows to operate at a much lower frequency (up to 2.2x lower), which leads to a reduction of power consumption for the I/O subsystem. With respect to the theoretical limits of the architecture, the proposed subsystem operates with an efficiency of 83% when considering only the I/O to L2 transfers, and an efficiency of 55% when we also introduce the contention introduced by the controlling processor, which execute the resident firmware on the same memory.

The rest of the work is organized as follows. Section II gives an overview of existing devices. Section III introduces the proposed SoC architecture followed by Section IV where we describe the *µDMA* architecture in more details. Section IV introduces the software support to better explain the results presented in Section V.

II. Related Work

In traditional MCUs the I/O subsystem is constantly supervised by the CPU, which is responsible for handling peripherals events and regulate data transfer between peripherals and memory. Recently introduced MCUs have increased the complexity of the I/O subsystem to support several power modes to selectively turn on from clock gating peripherals only when needed [5][6]. In these recent architectures, the intelligence of the peripheral subsystem has been increased providing the ability to run without CPU supervision in a much more efficient way, even when the CPU is in sleep mode. This "smart peripheral" approach is increasingly being used in most advanced MCUs. Although it has many variants with different names, the general concepts are common to all architectures.

The more conventional approach to configurable peripherals lies in connecting several peripherals together. This concept is employed, for example, in Microchip's Configurable Logic Cell (CLC)[7]. A microcontroller have several CLC blocks featuring a selectable set of inputs and outputs with limited logic capability so that the output of one peripheral can be fed to the input of another one. This "linked peripherals" approach can handle simple algorithms in a much more efficient way than the CPU. ST Microelectronics' STM32 also exploits this concept by means of "autonomous peripherals" leveraging a "peripheral interconnect matrix" to link peripherals such as Timers, DMAs, ADCs, and DACs together. The same concept can be applied to analog peripherals. Comparators, references, DACs and ADCs can be linked together to generate complex triggers [5].

In its latest SAM-L2 family of MCUs ATMEL proposed the "SleepWalking" feature, which enable events to wake-up peripherals and put them back to sleep when DMA data transfers are performed [8]. Renesas' RL78 has a "snooze mode" where the analog-to-digital converter (ADC) can autonomously acquire data when the processor is asleep [9]. Cypress Semiconductor's PSoC flexible family is one of the pioneers of the configurable digital and analog smart peripherals. This feature provides the ability to link peripherals together without involvement of the CPU. Their latest PSoC 4 BLE (Bluetooth Low Energy) can operate the BLE radio while

the CPU is idle [10]. The trend towards more functionality and complexity is highlighted by Microchip's PIC16F18877 family, where 10 ADC tied to the computational unit can do accumulation, averaging, and low-pass filtering calculations in hardware. The CPU can sleep until the filtered result exceeds programmed limits [11].

The common limitations of the presented MCUs lie in the connection of the peripherals to a global system bus shared with the processing subsystem, both converging to a single ported SRAM memory (see Figure 2). This leads to a significant overhead of the transfer efficiency due to the contention on the access to the memory, and an additional overhead due to bus arbitration (e.g. AHB), causing a poor utilization of the resources, and forming a bottleneck that limits the bandwidth sustainable by the peripheral-bus-memory subsystem. In the proposed architecture, a lightweight multi-channel I/O DMA is tightly-coupled to a multi-banked system memory rather than communicating through the system interconnect, allowing to improve the bandwidth by 2.2x with respect to traditional approaches when operating at a given frequency, or saving the same amount of dynamic power leveraging frequency scaling to achieve a given bandwidth. To achieve the autonomous operation, the I/O subsystem is controlled by a tiny CPU, and stands on a dedicated power domain, while the processing subsytem has its own dedicated domain.

III. SoC Architecture

Figure 3 shows the SoC architecture targeted by the proposed smart I/O subsystem. The SoC is divided in 3 different power domains. The always-on domain includes a power manager, a Real-Time Clock (RTC) and the wake-up logic. The other two domains are switchable, the first one including a tiny CPU (*fabric controller*), the peripheral subsystem, the clock generators and the main memory; while the second one includes the processing subsystem (e.g. a multi-core cluster). Both the SoC domain and the processing domain can be switched off and the L2 and clock generators have retentive capabilities to achieve very fast wakeup times.

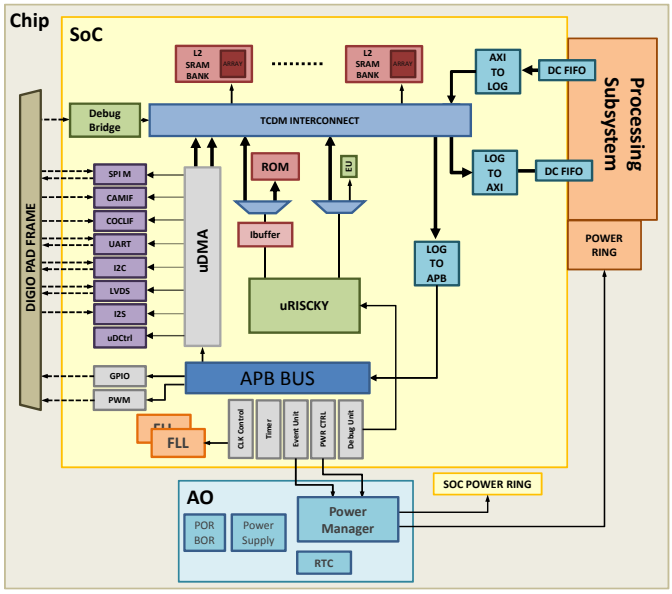

Fig. 3: SoC Architecture

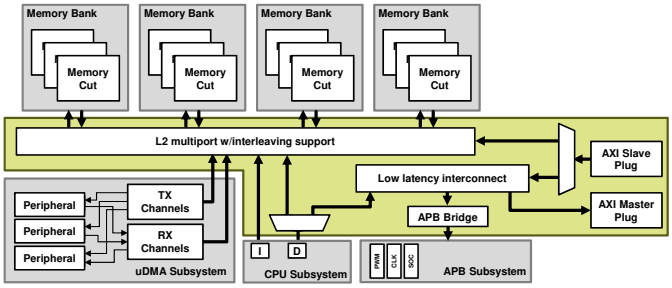

Fig. 4: Memory Interface

The SoC is based on a tiny RISCV CPU optimized for area. The CPU is connected to memory, accelerator and peripheral subsystem via a low latency logarithmic interconnect [12]. The main memory consists of 512KB of SRAM arranged in 4 banks with word level interleaving to minimize banking conflicts during parallel accesses through multiple ports of the interconnect. The SoC has an APB subsystem including pad GPIO and multiplexing control, clock and power controller, timer, *µDMA* configuration port and PWM controller. The connection with the accelerator is done thought 2 asymmetric AXI plugs with a 64-bit width for accelerator to memory communication and 32-bit for memory to accelerator communication. The *µDMA* access the memory through 2 dedicated ports on the logarithmic interconnect and on the other side connects to the peripherals. The peripheral offer of the *µDMA* subsystem includes 2 SPI (Serial Peripheral Interfaces) with QuadSPI support, 2 I2S channels with DDR (Dual-Data-Rate) support embedding 4 decimation filters allowing to connect up to 4 digital microphones, an hyperRAM/FLASH interface, a parallel camera interface and 2 I2C interfaces.

Figure 4 shows an overview of the *µDMA* subsystem architecture and its connections toward the rest of the system. The 2 ports of the *µDMA* connects directly to the interleaved memory area. The instruction port of the CPU connects either to the boot ROM or to the main memory while CPU data port and AXI slave connects to both interleaved region and peripheral interconnect since they can issue transactions for both system memory, peripheral subsystem and processing subsystem.

IV. *µDMA* Architecture

A generic peripheral capable of receiving and transmitting data off-chip is shown in Figure 5. A peripheral can have one or more data channels depending on its bandwidth requirements and bi-directional capabilities. Channels are mono-directional, hence a minimum of 2 channels is needed for a peripheral that supports both input and output. Typical bidirectional peripherals are SPI or I2C while unidirectional peripherals are camera and microphone interfaces. The *µDMA* has 2 ports connecting to the SoC interconnect directly to the interleaved memory area. Those direct connections limit the *µDMA* to access only the system memory and do not allow direct transfers to the processing subsystem or to other peripheral mapped on the APB bus. This assumption implies a bounded latency toward the memory reducing significantly the need for buffering on the datapath. Out of the 2 ports, one

is write only and dedicated to the RX (Receive) channels and one is read only and dedicated to the TX (Transmit) channels.

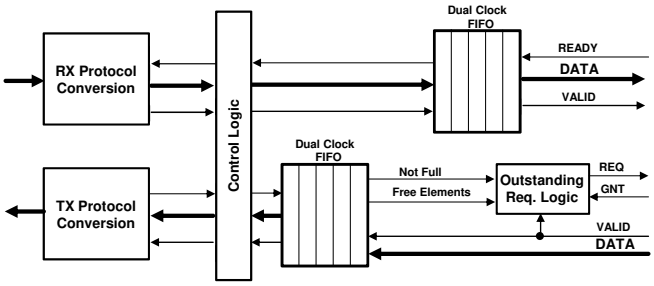

Fig. 5: Generic Peripheral Architecture

The ports towards the memory are 32-bit wide while the single channel interfaces toward the peripherals may have smaller widths. The widths supported by the μDMA are 8,16 and 32bits, either set at design-time or run-rime to fit the specific features of the attached peripherals. For example UART and I2C use 8-bit wide channels since at each data transfer they always send a single byte. Other peripherals like SPI or I2S feature a configurable data widths since the transfer width depends on the device they are controlling.

The peak bandwidth per port depends on the architecture and in our case is fixed to $32 * fsoc$(bit/s) where $fsoc$ is the frequency of the CPU and the I/O subsystem. Data transfer on TX and RX channels are completely decoupled and not synchronized. This means that if a peripheral request data on the TX channel and writes data on the RX channel there is no guarantee that the 2 transfers will take the same amount of time so, if synchronization is needed and TX and RX happens at the same time, it has to be implemented on the peripheral side. The CPU can enable or disable the channels and when the channel is enabled data transfer to/from memory is always triggered by the peripherals and the CPU is not involved in the data transfer.

Starting a transaction on a channels requires only 3 accesses in addition to the peripheral configuration: the software has to program the source or target pointer, the transfer length in bytes and send the start command. At the end of each transaction on each channel the μDMA generates a dedicated event to notify the *fabric controller* that is possible to queue another transaction. Each channel can have one transaction running and one enqueued that is automatically started at the end of the running transaction to support continuous data transfers based on a double buffering scheme.

A. RX channels

The block diagram of the logic that handles the handshake with the peripherals and performs the transfers to the memory is shown in Figure 6, while the channel handshake is shown in Figure 7. All the RX channels share the same port towards the memory. When a data word is available to be transferred from the peripheral to the memory, the peripheral rises the valid signal and notifies the μDMA. The μDMA performs an arbitration between all the active channels and acknowledges (with the ready signal) the data transfer to the winning peripheral. The peripheral considers the data transferred and can perform another transfer the following cycle. The μDMA logic stores the ID of the winning channel among with the data and the data-size of the channel. The next cycle the channel ID is used to select the channel resource and get the address for the memory transfer.

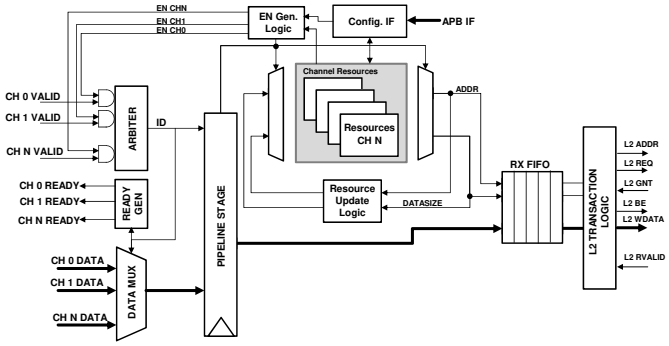

Fig. 6: RX channels Architecture

With channels resource we refer to a set of registers describing the channel status. The information stored in the channels resources are memory pointer, bytes remaining, status of pending transfers and channel enable. In the same cycle the selected channel resource is updated. The address, data, channel width and channel ID are then pushed to the RX transfer FIFO. On the other side of the FIFO the memory interface logic pops the transfer information, generates the byte enables and performs the transaction to the memory. The RX transfer FIFO is needed to absorb the latency introduced by the memory interface due to contention on the memory access and avoid continuous back propagation of stall signals.

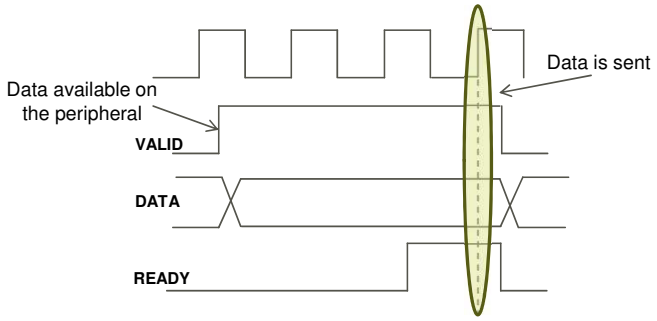

Fig. 7: RX channel handshake

The μDMA logic is fully pipelined and capable of handling 1 transfer per cycle when there is no contention on the memory banks. As discussed in the previous section the memory access is arbitrated at each cycle so that the worst case latency to the memory is always bounded. This assumption allows to use a shallow RX transfer FIFO without suffering from significant performance loss.

B. TX channels

Similarly to the RX channels, the TX channels share the same write port to the memory. The block diagram is presented in Figure 8 and the channel handshake in Figure 9. The

channel handshake for the TX is more complex than the RX part and has separate request and response paths. The split is needed to support outstanding requests which high bandwidth peripherals can perform to avoid the bandwidth degradation due to the round trip latency from request to response. In the TX channel when a peripheral needs data rises the request, the *µDMA* arbitrates the requests and gives the grant to the winning channel. The peripheral doesn't have to wait for the data in response and can issue another request.

To keep the handshake logic as simple as possible we assume that the response data cannot be stalled by the peripheral. This means that the number of outstanding requests in flight issued by the peripheral should be always lower or equal than the available spots in the peripheral FIFO. Peripherals that due to their low bandwidth do not suffer from the round trip latency can disable the outstanding requests and save area reducing the buffer size. Once the request from the peripheral is granted by the *µDMA* ,the ID of the winning channel is stored in a pipeline stage. In the next cycle, the address and datasize are fetched starting from the stored ID and the next address and number of bytes left are calculated and updated. The address with ID and datasize are then pushed to the TX FIFO.

At the output of the FIFO the memory transaction logic pops an element from the FIFO and performs the memory read. The received data is then sent to the proper channel that is selected by looking at the stored ID. The same considerations on output buffer size and pipeline capabilities done for the RX channels still apply.

A single TX channel with outstanding support has the possibility to completely saturate the TX port while a channel without this support can only reach a peak bandwidth of 1/4 the maximum bandwidth of the port. This limit is introduced by the round-trip latency from request to response.

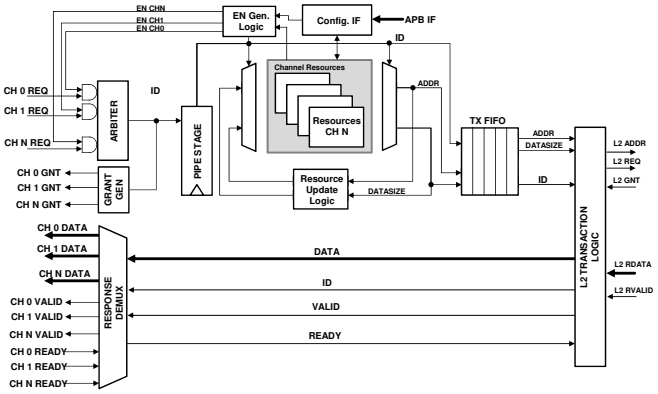

Fig. 8: TX channels Architecture

V. Software Runtime support

The runtime running on the *fabric controller* is managing it as a classic micro-controller which can delegate processing tasks to the processing subsystem. It thus contains classic services such as scheduling, memory allocations, drivers as well as dedicated services for communication with the accelerator.

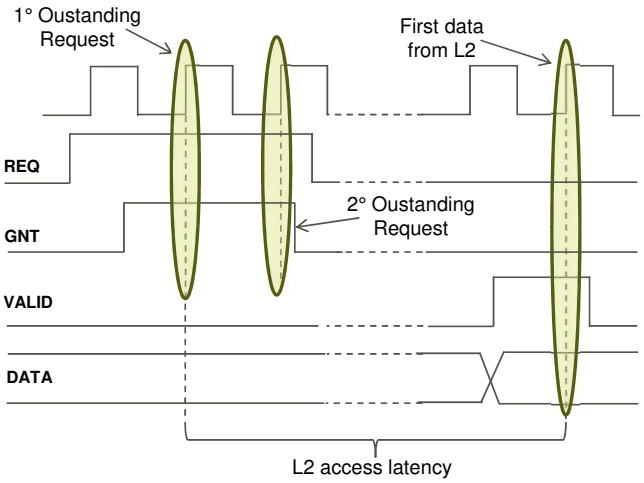

Fig. 9: TX channel handshake

A. Scheduler

As it is targeting ultra-low power systems with a small amount of memory, the scheduler is a run-to-completion task scheduler with no preemption. This allows using a single stack for all the tasks and avoid storing saved contexts in the memory. A task is just an entry point (a function and an argument). The scheduler takes the first available task from a FIFO, executes it until it returns, and then continues with the next one, until the FIFO is empty, in which case he goes to sleep. A running task can enqueue a task while it is being executed to defer some work. The processing subsystem can also enqueue tasks remotely in order to execute notification on the *fabric controller*. Hardware events (such as end of transfer) handled through interrupt routines on the *fabric controller* can also enqueue tasks to handle events out of the interrupt routine.

B. Drivers interactions with scheduler

In order to correctly handle all the events which will happen, such as end of transfers, timer events and so on, every asynchronous event can be attached a task or a handler, which will be enqueued or executed when the event occurs. Tasks will be enqueued and executed by the scheduler when the task is scheduled while the handler is executed immediately from the interrupt handler. This is for example useful to re-enqueue a transfer when one has just finished. The interrupt handler of the *µDMA* is called when the transfer is finished. It sees there is one task attached to the channel of the transfer, and thus enqueues it to the scheduler and leaves. Then later on, the task is scheduled, which enters the function of the task, which can then enqueue another transfer. If the delay implied by the scheduler layer can be a problem, the task can be replaced by a handler, in which case, the transfer is enqueued directly by the interrupt handler.

C. Data transfer with serial interfaces

The runtime is getting and sending data through serial interfaces using the *µDMA*. For that, it can delegate up to 2 transfers per channel to it, to let the *µDMA* always have one buffer ready. Once one transfer is done, an interrupt routine

is executed on the *fabric controller* to handle the end of transfer. This routine will enqueue the task or execute the handler attached to the corresponding channel. This task is usually supposed to feed again the *µDMA*, so that a continuous stream of data is transfer, and also to delegate the buffer to someone else, like the processing subsystem. For that it can enqueue a remote task to it.

D. Use case

A common use-case involving data transfers with peripherals and computing on the processing subsystem is the following: some data are sampled from several sensors and transferred to the system memory through several peripherals using the *µDMA*. This data are then processed by the processing system which produces new data in the system memory. This output data are then transferred outside through other interfaces still using the *µDMA*.

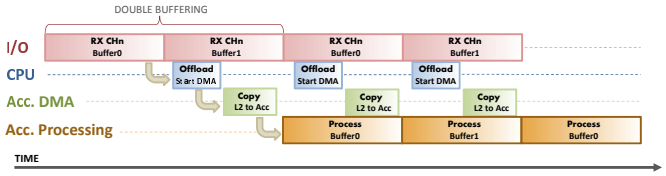

Fig. 10: Data flow with double buffering

Here is how this use-case is implemented. As the transfer will happen asynchronously between the peripherals and the system memory, it allocates 2 buffers for each data transfer, in order to let the *µDMA* use one buffer to transfer from the peripherals to the L2, while the other buffer is processed by the processing subsystem. Then it starts the use-case by enqueueing the 2 first transfers to the *µDMA*, and that for each channel.

Each time a transfer is finished on a channel, the interrupt handler is enqueueing a task to the cluster to delegate the processing of the buffer. Figure 10 shows how the different resources in the system are used during a data transfer and processing for a single RX channel.

VI. Results

To validate the *µDMA* and and measure its behavior when operating under extremely high bandwidth conditions we added, in parallel to the real peripherals, 3 traffic generators. Each traffic generator has 1 TX and 1 RX channel and for each channel has a software configurable I/O bandwidth. The architecture of the traffic generator is very similar to the one presented in Figure 5 where the protocol conversions are substituted with behavioral bandwidth generators.

The datarate at which words are pushed to the dual clock FIFO is determined by a configuration register. We used the traffic generators because, unlike the real peripheral that can be stalled when starving or when the FIFOs are full, they can inject a constant data bandwidth and monitor for data loss. This allows to asses the maximum bandwidth that can be sustained continuously.

Figure 11(a) shows the maximum sustained bandwidth that we can support during I/O to the system memory transfers for different frequencies and different data transfer sizes. The maximum bandwidth is tested when all 3 traffic generators are active with both TX and RX channels enabled. We can see that the impact of transfer length is negligible and we can easily saturate the two ports. If we compare with a traditional 32-bit system with a single ported system memory, a CPU and a central DMA the maximum bandwidth that can be transferred from I/O to the system memory is equal to $32 \times f_{sys}$ bit/s. With a frequency of 50Mhz, this means 1.6Gbit/s assuming the presence of an ideal autonomous I/O subsystem capable of reaching the physical limit. In our case we can reach on the implemented system 2.6Gbit/s including all the runtime overheads needed to reprogram the data transfers in a double buffering scheme.

In Figure 11(b) we see how the bandwidth remains constant when changing the number of TX or RX channels. This changes when we move from TX or RX only to have both active at the same time. This is because in the *µDMA* TX and RX channels are decoupled and use different ports.

We performed similar experiments but including the transfer to the accelerator internal memory. The runtime running on

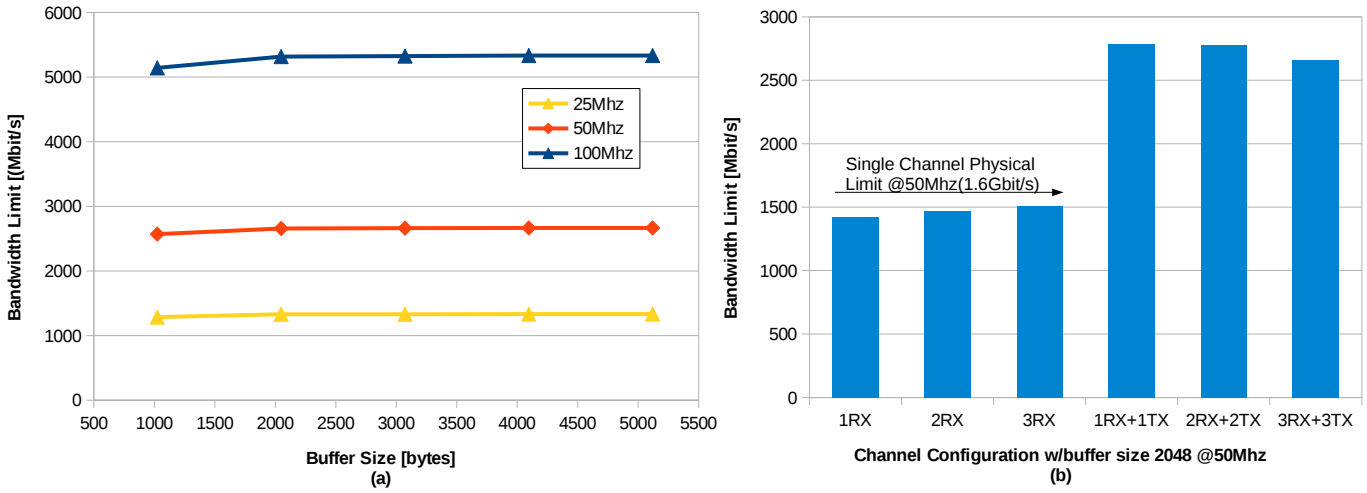

Fig. 11: Maximum Bandwidth from/to System Memory

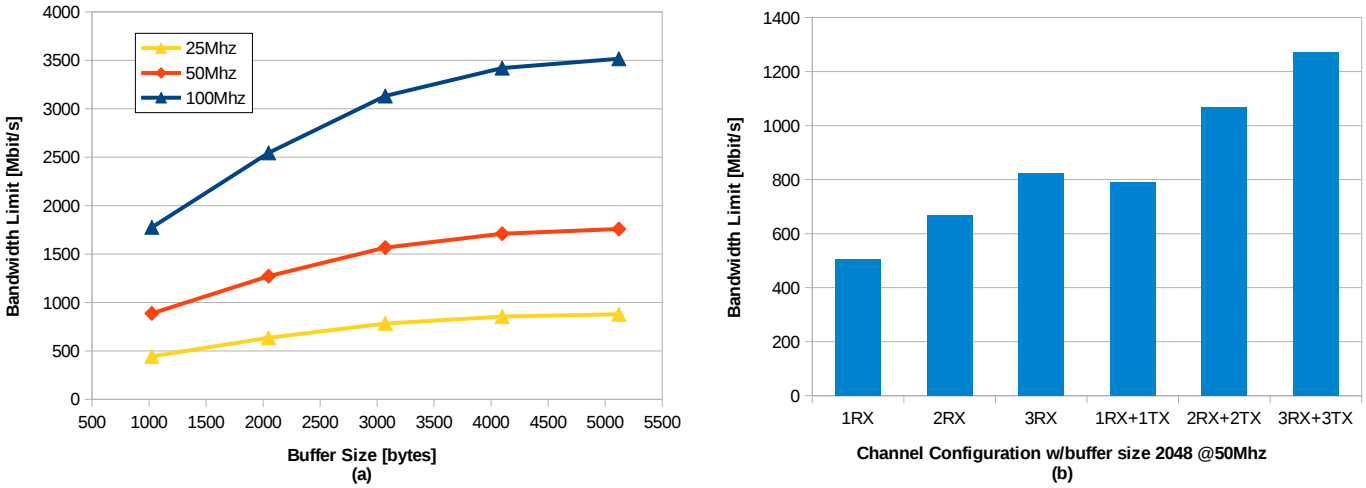

Fig. 12: Maximum Bandwidth I/O to/from Processing Subsystem

the CPU notifies the processing subsystem when a data chunk is available on the system memory and communicates the pointers to the data structures so that the processing subsystem can start a DMA transfer and fetch the data. The data flow with double buffering scheme used for those experiments is shown in Figure 10 and was explained in details in the previous section.

In Figure 12(a) we see the maximum bandwidth transferred from I/O to accelerator for 3 different frequencies and for different transfer sizes. Contrary to the simple transfer case the performances are highly dependent on the transfer size. The cause are the CPU cycles needed to exchange information with the processing subsystem. In high bandwidth scenarios with multiple concurrent channels active this can easily saturate the CPU utilization when the transfer size is small and the *µDMA* generates a high number of events. Same as before we can do a simple estimation of what is the maximum bandwidth that a traditional MCU can sustain. In this case we should consider also that the data has, not only to be stored in the system memory, but also it needs to be read. This implies that for a 32bit system running at 50Mhz the maximum data rate at which the system can store and then process the data is maximum 800Mbit/s.

With our architecture we can see that under the same operating frequency we can sustain data rate of 888Mbit/s in case of 1KB transfers and up to 1.76Gbit/s for transfer sizes of 5KB.

Figure 13 shows in the same graph the maximum bandwidth in case of I/O to system memory and I/O to processing subsystem transfers. In the same figure we plotted the architectural limits for both our case and for the traditional MCUs. In our architecture the physical limit is given by the maximum data that can be transferred on the 2 ports that connect the *µDMA* to the system interconnect. The limit does not change when we transfer data to the processing system since there is a dedicated connection from the processing subsystem to the system memory.

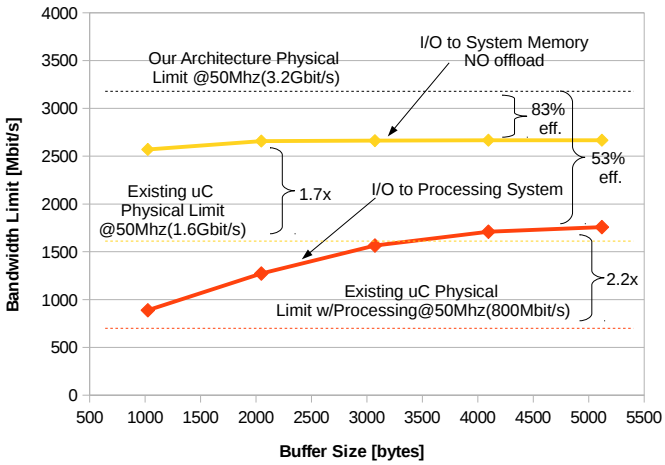

Fig. 13: Comparison

When we consider a transfer from the I/O to the system memory we can transfer data at a rate which is 83% of the physical limit. The drop is due to the contention on the memory by the TX and RX channels. If we consider the transfer including the offload to the processing subsystem the efficiency drops to 53% and this is due to the contention of TX and RX channels on top of the contention generated by the data transfers from the processing subsystem and from the *fabric controller*.

Peripheral	I/O Frequency(MHz)	Bandwidth(Mbit/s)
HyperRAM/FLASH	70	1120
QuadSPI	70	280
Camera	15	120
StandardSPI	25	25
Audio Interface	6	12
	Total Peak Bandwidth:	**1557**

TABLE I: Peak bandwidth for an example use case

In both cases our efficiency in transferring data in and out of the chip is much higher compared to traditional MCU. This

allows, in a fixed bandwidth scenario (ex. Table I), to operate at a much lower frequency. We can lower the frequency by 1.7x when just transferring to/from the system memory or up to 2.2x when considering transfers to the processing subsystem.

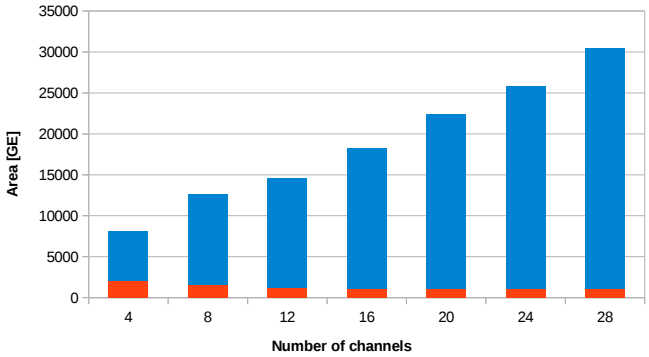

Fig. 14: *µDMA* area with different channel numbers

In Figure 14 we can see the area occupied by the *µDMA* in different channel configuration. The red bar shows the area of a single channel. As expected when the number of channels grows the area per channels goes down because the common logic is shared between more channels resulting in a lower overhead per channel. The area of a single channel goes from a maximum of 2kGE with a 4 channel configuration to a minimum of 1.08kGE when the maximum number of channels (28) is instantiated.

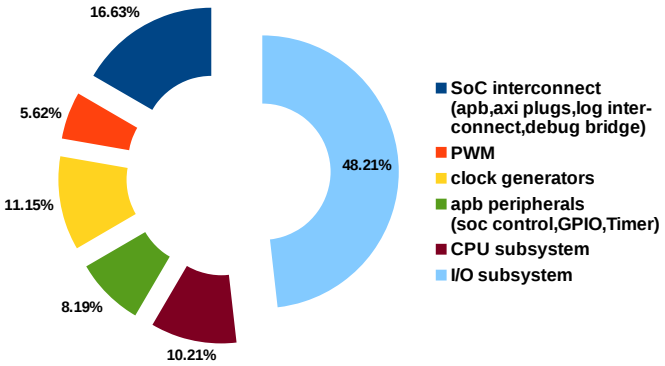

Fig. 15: *µDMA* SoC area breakdown

Figure 15 shows the area breakdown for the proposed SoC architecture but with the exclusion of the SRAM cuts. The whole I/O subsystem accounts for 48.2% of the SoC area, Table II shows in detail which are the main contributors. As we can see the *µDMA* is only 10.1% of the I/O subsystem area. The technology used during the synthesis trial is TSMC40nm Low Power and the target frequency has been set to 200Mhz.

VII. Conclusion

This work presented an autonomous I/O subsystem for managing data streams coming from multiple peripherals, targeting next generation MCU architectures for IoT end-nodes. The proposed architecture is built around an I/O DMA tightly-coupled with a multi-banked system memory controlled by a tiny controller. This approach overcomes the bandwidth limitations of state-of-the-art MCU's I/O subsystems, that often leverage a bus shared with the data processing subsystem, converging on a single-port memory. The proposed architecture achieves a transfer efficiency of 84% when considering only data transfers, and 53% if we consider also the overhead of the runtime running on the controlling processor, reducing the operating frequency of the I/O subsystem by up to 2.2x with respect to traditional architectures, for a given bandwidth requirement.

	Area(GE)	Percentage of I/O subsystem
µDMA	15363	10.12
HyperRAM/FLASH	44916	29.59
4xI2S+FILTERS	52404	34.52
Camera	5663	3.73
2xSPI	28118	18.52
2xI2C	5324	3.51
UART	2140	1.39
Total Area:	**151788**	

TABLE II: Area breakdown for I/O subsystem

References

[1] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, ser. MCC '12. New York, NY, USA: ACM, 2012, pp. 13–16. [Online]. Available: http://doi.acm.org/10.1145/2342509.2342513

[2] D. Rossi, I. Loi, A. Pullini, and L. Benini, *Ultra-Low-Power Digital Architectures for the Internet of Things*. Cham: Springer International Publishing, 2017, pp. 69–93. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-51482-6_3

[3] Cypress, "S6J3200 Series, Datasheet," Tech. Rep. 002-05682 Rev.E, April 2017.

[4] NXP, "LPC4350/30/20/10 32-bit ARM Cortex-M4/M0 flashless MCU Product Data Sheet," Tech. Rep. Rev. 4.6, March 2016.

[5] ST Microelectronics, "STM32L4x5 advanced ARM®-based 32-bit MCUs," Tech. Rep. RM0395 Reference Manual, February 2016.

[6] Ambiq Micro, "Ultra-Low Power MCU Family," Tech. Rep. Apollo Datasheet rev 0.45, September 2015.

[7] Microchip, "Configurable Logic Cell Tips 'n Tricks," Tech. Rep. DS41631B, Jannuary 2012.

[8] Atmel, "SMART ARM-based Microcontrollers," Tech. Rep. SAM L22x Rev. A Datasheet, August 2015.

[9] Renesas, "RL78/G13 User's Manual Hardware," Tech. Rep. Rev. 3.20, July 2014.

[10] Cypress, "Designing for Low Power and Estimating Battery Life for BLE Applications," Tech. Rep. AN92584 Rev. C Application Note, March 2016.

[11] Microchip, "Analog-to-Digital Converter with Computation Technical Brief," Tech. Rep. TB3146 Application Note, August 2016.

[12] A. Rahimi, I. Loi, M. R. Kakoee, and L. Benini, "A fully-synthesizable single-cycle interconnection network for shared-l1 processor clusters," in *2011 Design, Automation Test in Europe*, March 2011, pp. 1–6.