Ultra-Low-Latency Lightweight DMA for Tightly Coupled Multi-Core Clusters

Davide Rossi[1], Igor Loi[1], Germain Haugou[2], Luca Benini[1,3]
[1] Dept. of Electrical, Electronic and Information Engineering, University Of Bologna, Italy
[2] ST Microelectronics, Grenoble, France
[3] Dept. of Inf. Technology and Electrical Engineering, Swiss Federal Institute of Technology Zurich, Switzerland

davide.rossi@unibo.it, igor.loi@unibo.it, germain.haugou@st.com, luca.benini@unibo.it

ABSTRACT

The evolution of multi- and many-core platforms is rapidly increasing the available on-chip computational capabilities of embedded computing devices, while memory access is dominated by on-chip and off-chip interconnect delays which do not scale well. For this reason, the bottleneck of many applications is rapidly moving from computation to communication. More precisely, performance is often bound by the huge latency of direct memory accesses. In this scenario the challenge is to provide embedded multi and many-core systems with a powerful, low-latency, energy efficient and flexible way to move data through the memory hierarchy level. In this paper, a DMA engine optimized for clustered tightly coupled many-core systems is presented. The IP features a simple micro-coded programming interface and lock-free per-core command queues to improve flexibility while reducing the programming latency. Moreover it dramatically reduces the area and improves the energy efficiency with respect to conventional DMAs exploiting the cluster shared memory as local repository for data buffers. The proposed DMA engine improves the access and programming latency by one order of magnitude, it reduces IP area by 4x and power by 5x, with respect to a conventional DMA, while providing full bandwidth to 16 independent logical channels.

Categories and Subject Descriptors

B.5.1 Design, B.6.3 Optimization, C.1.2 Multiple Data Stream Architectures (Multiprocessors).

Keywords

Many-core, multi-core, DMA, low latency.

CF'14, May 20 - 22 2014, Cagliari, Italy
Copyright 2014 ACM 978-1-4503-2870-8/14/05…$15.00.
http://dx.doi.org/10.1145/2597917.2597922

1. INTRODUCTION

As the computational power of electronics devices continues to scale with Moore's Law, an increasing number of applications are becoming limited by memory latency and bandwidth. This fact is mainly evident in multi- and many-core Systems-On-Chip (SoC), where the enormous computational power available on-chip moves the bottleneck of the applications from computation to data transfers. For this reason, efficient memory transfers are of primary importance to achieve the performance and energy efficiency targets of applications in multi and many-core systems.

The problem of efficient data transfers has been addressed in the past mainly adopting two solutions. Cached systems allow handling access to external memory in a complete transparent way from the programming viewpoint, thus providing a user friendly way to handle the problem. Although cached systems are already a standard for desktop machines, the hardware complexity required to keep the data history and maintain the coherency is extremely high, thus often unsuitable for many embedded applications. A more effective way of handling data transfers in embedded systems is that of coupling scratchpad memories with direct memory access (DMAs) controllers: It has been demonstrated that scratchpad memories are more energy efficient than caches for embedded applications [1]. However, programming requires the explicit configuration and trigger of memory transfers. Once this is done, memory transfers can be performed in parallel with computation (if data dependencies are managed appropriately), allowing for higher efficiency thanks to the overlap between computation and communication.

A widely used technique to decouple and overlap computation and communication in DMA + scratchpad architectures is *double buffering*, where data access is organized in chunks and processors work on a "current" chunk while DMA fetches the "next" chunk from a remote memory and copies it to the local scratchpad. In many systems, the large programming latency of full-featured DMAs can prevent double buffering technique to be effective, especially when adopted in applications requiring transfers of small data chunks [2]. When the applications allow for that, a possible way to hide the DMA latency consists of increasing the size of the buffers. However, this approach is often limited by the size of L1 memory, and it cannot be always efficiently exploited. A wide class of applications including reduction operations, contended object updates, and data walks leverages on small and data-dependent access to the memory hierarchy based on active messages [2]. For these applications the huge latency of conventional DMAs is destructive for both performance and energy efficiency. Another issue related to

DMAs in multi and many-core platform is that of contention on the access to logical (programming) channels. Increasing the number of cores the contention on the programming channels of the DMAs limits the transfer efficiency forming a huge bottleneck for application performance. On the other hand coupling each processor with a large DMA engine is not feasible, as it causes relevant overheads in area and power, which is not suitable for many embedded applications.

This paper introduces an ultra-low-latency, highly energy efficient DMA engine optimized for clustered shared memory many-core systems. The proposed IP is equipped with independent per-core command queues, enabling lock-free sharing of the DMA channels among the cores. Moreover it avoids utilization of large internal store buffers by exploiting the cluster's shared memory as local repository avoiding the usage of large internal data buffers, thus reducing the area and power consumption with respect to conventional DMAs.

The paper is structured as follows. Chapter 2 gives an overview of the related works, section 3 provides a high level description of the system architecture targeted in this work, on the DMA integration on the system, and on the programming interface. Section 4 describes the DMA micro-architecture. Section 5 shows the experimental results and comparison with state-of-the-art DMAs used as reference. Section 6 shows a system level exploration providing use-case examples based on real-life applications. Section 7 provides some concluding remarks.

2. RELATED WORK

Direct memory access mechanisms have been widely studied in the past years, with the objective of overlapping as much as possible data-transfer and computation in order to achieve the Amhdal's limit of the applications. Most of modern commercial systems-on chip for embedded and mobile applications, such as TI KeyStone [4], NVIDIA TEGRA [5], and Qualcomm Snapdragon [6] feature a *system DMA*. In these platforms the optimization of the power and area of the DMA is not of primary importance as it form a very small portion of the overall System-On-Chip area. Contrarily, an important concern of the system DMA is to be configured by all the programmable resources of the System-On-Chip. For this reason such kind of systems usually include several DMAs featuring multiple independent channels and support for complex memory transfers.

In the context of MPSoCs, DMA mechanisms are widely used to efficiently move data between the different levels of memory hierarchy. A striking example is the Cell processor [7]. Composed of one PowerPC general purpose core and eight Synergistic Processing Units (SPU), the architecture of this high performance multiprocessor is focused around the use of the DMA units to move data from the external memory to the local memories of the SPUs to conceal memory access penalty. Many high-end multi-processor systems, such as Intel Core 2 [8] and AMD Opteron [9] address the problem of hiding latency and data transfer using structured cache hierarchy with support for hardware and software prefetching. Double buffering is possible and effective due to HW/SW prefetching with explicit instructions support. On the other hand, hardware prefetching architectures require large prefetch history table to efficiently predict the prefetch address, and this overhead is often not suitable for embedded applications. Moreover, the optimal prefetch distance is difficult to determine in advance for several reasons, and experiments are required to empirically find the optimal prefetch distance. Finally, prefetch is not effective for applications with irregular or unpredictable data-flows [9].

Most heterogeneous multi-core platforms for embedded computing feature several general purpose or specialized computing elements coupled to one or more DMA engines. The Morpheus platform provides a distributed DMA mechanism that allow the central processing unit to configure, trigger and synchronize data transfers among the specialized units [10]. The SARC Architecture is composed of multiple specialized processors and a set of user-managed DMA engines that let the runtime system to automatically allocate the tasks and overlap data transfer and computation phases [11]. DME is a flexible, fully micro-coded dual-core controller optimized for handling memory transactions and for managing the memory and address space in NoC-based multi-core platforms [12]. The presented approaches are effective, as providing private per-core, or per NoC-node DMAs minimize the contention and maximize the throughput. However, the area overhead of coupling each processor with a DMA, or to each NoC node is usually huge.

When moving to the clustered many-cores domain, the systems are usually composed of clusters including several, small, general purpose processing elements. In this domain, coupling each processing element with a DMA engine becomes counter-productive, as the area and power consumption of the DMA engines easily overcome that of processors, causing large overheads. For this reason, clustered many cores usually share one or few DMA engines among the cores of a cluster, usually ranging between 8 and 32. An example of this kind of platform is P2012/STHORM [13]. In this platform, two DMA engines featuring multiple channels, are shared among the 16 cores of the cluster. The DMAs feature multiple out-of-order outstanding transaction, complex data access patterns (e.g. 1D, 2D), store-and-forward mechanism, and they can be configured through a linked list of programming sequences stored in the cluster memory. Although this DMA can efficiently handle multiple extremely complex data transfers, completely offloading the cores from address calculations, its configuration phase is extremely burdensome, and it forms a relevant bottleneck, especially for latency bounded applications. In a tightly coupled multi-core cluster, part of the processors computation can be used for the address generations, as for many algorithm it form an irrelevant computational load with respect to the whole computation. As long as this is coupled to an ultra-low-latency, ultra-low overhead mechanism to access the DMA logic channels, a relevant latency improvement with no bandwidth penalty can be achieved.

A specific field in which the programming latency is minimized is that of GPGPUs. GPGPUs features automated hardware-assisted low-latency programming of DMA transfers when coalesced threads are intercepted [14]. However, coalescence is not always feasible due to divergence of threads. Moreover, within GPGPUs the users do not have a direct visibility of the DMA channel, this makes double buffering extremely challenging. A way of eluding the problem is CudaDMA [14]. CudaDMA is an Application Programing Interface (API) able to emulate DMA behavior on GPUs. It exploits warp specialization by providing primitives that allow to create DMA warps and compute warps that runs concurrently on the same thread block. This technique enables double buffering while avoiding thread divergence (within a warp). On the other hand, it requires a strong restructuration of applications, avoiding the adoption of the regular data-space

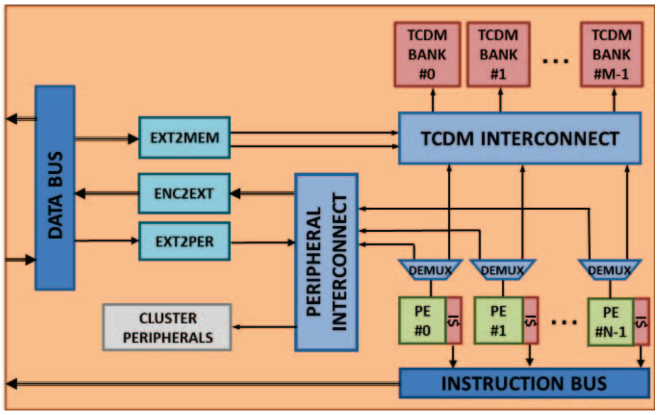

Figure 1. Reference Cluster Architecture

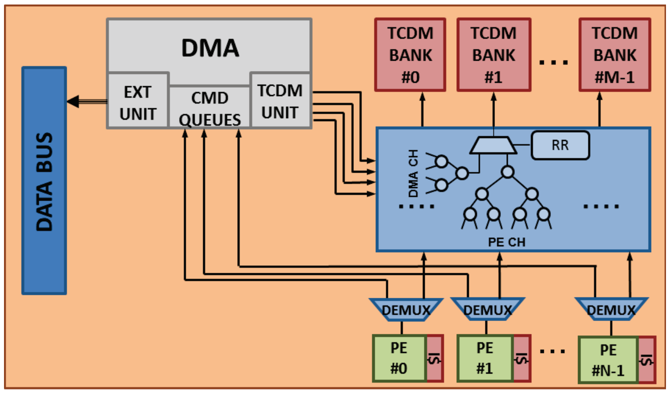

Figure 2. Integration of the DMA within the cluster

decomposition provided by programming language such as CUDA [15] and OpenCL [16]. Also, it is not useful in applications with non-uniform memory access patterns.

The DMA proposed in this work is explicitly designed and optimized for integration in multi-core tightly coupled clusters of processors. With respect to conventional DMAs used in multi- and many-core systems, it introduces the following innovations:

- Simple programming interface that reduces the programming latency and leverage on the software programmability of processors to improve flexibility.
- A hardware mechanism to enable each core within a cluster to provide a software transparent access from the cores toward the DMA logic channels avoiding DMA channels reservation penalty.
- Exploitation of the local memory of clusters as repository for the transfer buffers, to reduce the hardware complexity providing significant benefits in terms of area and power consumption.

3. A SYSTEM VIEW

This section presents the overall system architecture where the proposed DMA is deployed, providing also a description of its integration on the multi-core cluster and its programming model.

3.1 Target cluster architecture

The considered cluster architecture, presented in Figure 1, features a parametric number of Processing Elements (PEs) composed of OpenRISC cores, each one with a private instruction cache. The refill ports of the I-caches converge on a common cluster instruction initiator port through a cluster instruction bus. The PEs do not have private data caches, avoiding memory coherency overhead and increasing area efficiency, while they all share a L1 multi-banked tightly coupled data memory (TCDM) acting as a shared data scratchpad memory. The TCDM has a number of ports equal to the number of memory banks providing concurrent access to different memory locations. Intra-cluster communication is based on a high bandwidth logarithmic interconnect (TCDM interconnect), implementing a word-level interleaving scheme in order to reduce the access contention to TCDM banks. It consists of a Mesh-of-Trees (MoT) interconnection network supporting single cycle communication between PEs and memory banks (MBs) [13]. In case of multiple conflicting requests, for fair access to memory banks, a round-robin scheduler arbitrates the accesses.

Two additional ports are present in the TCDM interconnect to provide access from the resources external to the cluster, through the EXT2MEM adapter. A peripheral interconnect provides access to all the peripherals and to all the resources external to the cluster. The arbitration and protocol adaptation necessary for the processors to communicate to the TCDM and peripheral interconnect is implemented the DEMUX, connected to the data interface of each PE. Finally, two adapters (EXT2PER and ENC2EXT) implement the protocol conversion between the peripheral interconnect and the cluster data bus.

3.2 DMA Integration

As shown in Figure 2, the proposed DMA engine targets the integration on the tightly coupled cluster of processors described in the previous subsection.

One peculiar feature of the proposed DMA consists of the configuration interface toward the processors. Most of conventional DMAs for multi- and many-core systems provide a single configuration interface. In many cases, such in the case of the CELL BE processor, this is a private interface as each DMA is dedicated to a single processing element. As described previously in the paper this solution is not scalable, thus not suitable for many-core systems. In other cases, a single or few configuration interfaces are shared among all the cores of a cluster. With this architecture, to get the ownership of the control interface, each PE needs to check its status (*free* or *busy*), lock it by setting a *busy* flag, configure and trigger the transfer, and unlock the *busy* status flag once the transfer is finished. This mechanism introduces two forms of overhead. First, an implicit, always present overhead formed by the lock/unlock phases required to get the ownership of the DMA. Second, an access contention, that depends on the number of processors concurrently trying to access to the DMA configuration port. This second scenario might be relevant when executing data parallel processing typical of many-core programming models such as OpenCL, where the same program (including data transfers) is executed by all the cores of the cluster. To mitigate this, some DMAs implement a configuration mechanism based on linked lists stored on the local memory. The processors write in memory the configuration structures containing data transfer directives and trigger the DMA. The DMA fetches the "nodes", which are organized as a linked list and executes them. This reduces blocking in the processors, but still does not solve DMA control bandwidth issues and it makes the

DMA more complex because it needs to manage the fetching of nodes and pointer chasing in the memory node list. This is what happens, for example, in the STHORM platform.

To overcome these overheads, we introduce a FIFO based, private command queue for each core. Each processor has the ability to configure DMA transfers sending commands to a dedicated memory mapped control interface accessed through an additional port of the DEMUX, without any needs for reservation (lock/unlock). Each processor can enqueue to its own control interface an arbitrary number of transfers. When the command queue of a processor is full and the processor tries to write a new command, the incoming request is stalled, and it is committed as soon as one location in the command queue is released. Moreover, the arbitration among the transfer requests coming from the PEs, is performed internally by the DMA. The whole enqueue and arbitration mechanism is completely transparent to the software, eliminating all the overheads related to the access contention on the DMA configuration port.

For what concern the data path, the proposed DMA features two ports customized to serve the specific requirements of the memory subsystems of the platform: the TCDM within the cluster, and a typical off-chip memory on the external port. The external unit implements all the features typical of advanced non-blocking bus protocols such as AXI or STBUS. The interface toward the bus subsystem, 64-bit wide in the current implementation, features decoupled read and write channels to optimize the performance when concurrent TX and RX transactions are required. In order to hide the latency typical of DDR memory access, the external unit is designed to support a parametric number of outstanding transactions (16 in the current implementation). Moreover, to handle consistency of concurrent transfers from different levels of memory hierarchy it handles on-the-fly reordering of incoming packets.

On the TCDM side, the interface has been customized for integration on the single-cycle latency TCDM interconnect. With respect to the external unit, this interface is therefore much simpler in terms of hardware complexity, since it does not need to implement multiple out-of-order outstanding transactions. On the other hand, to maximize the bandwidth avoiding bottlenecks when concurrent read and write transfers are performed, it implements 4 32-bit ports for the connection toward the TCDM interconnect. 2 ports are used for read only operations and 2 ports for write only operations (Figure 2). To implement a fair arbitration and reduce the complexity, the TCDM interconnect is implemented as two separate trees, one handling requests form processors and one is used for the DMA. The two trees only share one node implementing a fair round-robin arbitration that allocates, in case of multiple conflicting requests, half bandwidth to the DMA and half bandwidth to the processors. It should be noted that a typical configuration of the banking factor (# of TCDM banks/# of cores) is two (e.g., 32 memory banks for 16 cores), ensuring, on average, sufficient bandwidth for both computation and communication.

3.3 Programming model

The programming model of the proposed DMA engine consists of a set of basic commands enabling low programming latency configuration of DMA transfers from the TCDM memory to the system memory hierarchy and vice-versa. In contrast to many of conventional DMAs for multi- and many-core systems, able to handled complex transfer patterns, (e.g. 2D transfers) thus requiring complex configuration structures to be initialized, the approach adopted in this work leads to a very simple, ultra-low latency configuration procedure.

The rationale behind this assumption is that when dealing with many-core systems, the overhead required to handle address calculation for non-linear transfers can be easily managed by the computational power available by the cluster. In this scenario, the crucial point to improve latency is to reduce as much as possible the commitment of the first transfer, while the calculation of the addresses for the following transfers can be overlapped with the completion of the first one. In order to avoid overheads caused by the offset calculation, all the configuration registers are seen at the same address from the processors, each one targeting its private command queue on the DMA.

The DMA API is composed of four macros, each one consisting on one write or read operation implemented by few processor instructions, thus extremely fast to be executed:

- *set_tcdm_addr(TCDM_ADDRESS)*. Set the address of the buffer on the TCDM memory.
- *set_ext_addr(EXT_ADDRESS)*. Set the address of the buffer on the system memory hierarchy.
- *enqeue_command(COMMAND)*. Write a command on the command queue. The supported commands are power of two load and store operations, ranging from 1 byte to 32Kbytes.
- *get_queue_status()*. Get information about the status of completion of the transfers (i.e. number of missing bytes). Synchronize the processor execution with the end of the DMA transfer.

4. DMA ARCHITECTURE

An overview of the micro-architecture of the proposed DMA engine is presented in Figure 3. It is composed of four modules. The control unit (CTRL UNIT) handles the arbitration of requests coming from the processors, forwards the synchronization of transfer to the proper command queue, and generates the control signals for the other units to properly handle data transfers. The external unit is responsible for providing an interface toward the external bus, implementing a full-featured STBus T3 protocol interface. The TCDM unit implements the interface toward the cluster local memory. Finally, the transfer unit contains two FIFOs, one for TX transfers, one for RX transfers used to decouple data packets flowing through the two interfaces.

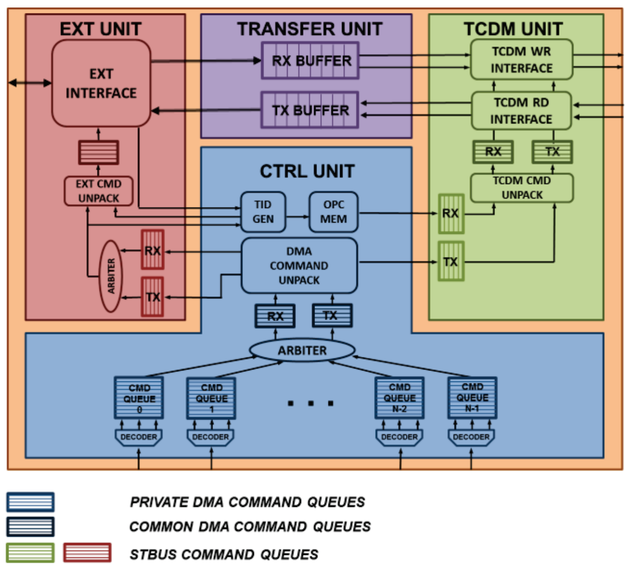

Figure 3. DMA Architecture

4.1 Overview

The private command queues (one per core) include memory mapped registers implementing the three fields required to initialize each transfer. This entity, composed of TCDM address, external address and the operation is called *command*. The operation field of a command may include all the operations supported by the API, thus load and store transfers ranging from 1 byte to 32Kbytes. Commands incoming from the private command queues are arbitrated using the round-robin policy, and forwarded to a common command queue (Figure 3). In order to completely decouple RX and TX DMA commands, that can be partially overlapped on the STBus interconnect due to its dual-channel (request and response) structure, two different queues are implemented on all the DMA sub-modules. A command unpack module splits the commands incoming from the common command queue into several STBus commands and forward them to the interfaces. A STBus command consists of a transfer entity implementing a subset of the operations supported by the DMA commands coupled with all the information necessary to perform the transfer on the STBus channels (both request and response). More precisely it implements load and store operations ranging from 1 to 64 bytes. Within the EXT UNIT and TCDM UNIT, STBus commands are split again into independent cells, implementing the basic operations that are forwarded to the cell queue and committed cycle-by-cycle by the TCDM and external interfaces.

This hierarchical packaging allows to minimize the area required to store temporary transfer information on the DMA queues. DMA commands, that provide the maximum density (#of bits required to store the transaction information/size of transfer) are stored in the largest queue. On the other hand, the additional information required to handle STBus transactions are generated and forwarded to the STBus command queue and to the cells queue only when necessary, minimizing the area of these two FIFOs as well. For this reason, the STBus command queue and the STBus cells queue are kept as small as possible in the implementation of the DMA, not affecting its overall performance. On the other hand, the size of the DMA command queue determines to the overall number of outstanding DMA commands that can be enqueued by the processors before being stalled, thus potentially affecting the transfer performance when issuing complex, non-linear data transfers composed of several linear transfers.

4.2 Support for multi-core

The proposed DMA is designed to implement a completely software transparent multi-core enqueue, arbitration and synchronization mechanism with the minimum impact on area. To be capable to serve requests coming from multiple command queues (capable of initiating transfers) a synchronization mechanism that notify the end of transfer is required. However, the hardware cost of synchronization, as well as the all the hardware cost of the modules that are replicated per-core, should be kept as small as possible in order to not affect the scalability of the DMA. To minimize the area overhead of the command queues the number of locations of the private command queue is kept at the minimum (2 locations), while a larger number of locations is assigned to the common DMA command queue.

To achieve synchronization, each private DMA command queue features its own ID. When a DMA transfer is issued from a private command queue, the source ID associated to the private queue is forwarded to the common DMA command queue together with the DMA command. At the same time, a counter within each private command queue is incremented by the number of cells present in the operation submitted. Every time a transfer of the STBus command completes, both the EXT UNIT and TCDM UNIT notify the end of transfer to the command queue associated to the ID of the transfer, which decrement the value of the counter. When the value of the counter is equal to zero, the transfer is complete.

The status of the counter is accessible through the control interface of the command queue, and it can be read at any time by the processors. This way, each processor is capable to know at any time the status of the requested transfers. The choice of notifying the processors with the number of STBus commands is a good trade-off between the notification of the number of cells, that would provide redundant information with a huge hardware complexity required for its implementation, and the number of DMA commands, that allows for a smaller hardware complexity, but does not provide more information about the status of the transfer than its completion.

4.3 High bandwidth, low cost architecture

A key objective of the proposed DMA is to provide full bandwidth with the minimum cost in terms of area and power by exploiting the intrinsic characteristics of the cluster integrating it. Most of conventional DMAs engines implement a store and forward mechanism to avoid the propagation of blocking behavior from one physical channel to another (e.g. from the TCDM interface to the EXT interface). This implies each data packet needs to be pre-buffered within the DMA before being forwarded from one to the other physical channels. This happens because DMAs are usually realized as generic IPs designed to be coupled to few interconnects rather than close to a specific level of the SoC memory hierarchy.

One specific target of all advanced DMAs is that of providing efficient external memory accesses, being capable to hide latency of hundred cycles of magnitude, typical of DDR memories. To

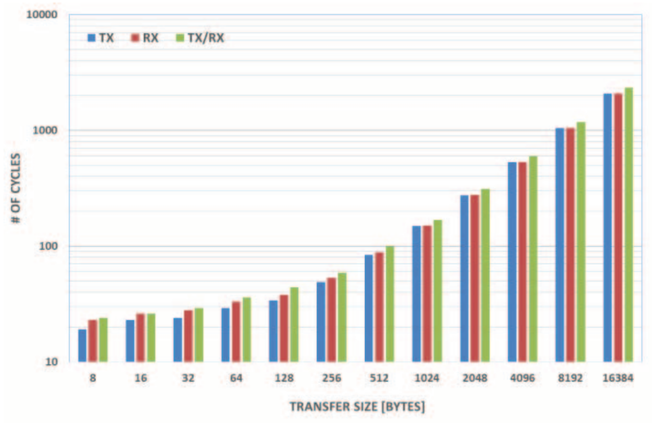

Figure 4. Transfer time for transfers of different data size

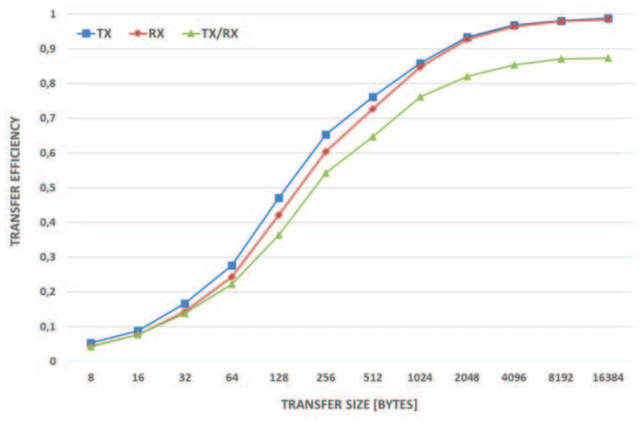

Figure 5. Transfer efficiency for transfers of different size

hide this huge latency, it is necessary to implement a multiple outstanding transactions mechanism. This means that a significant number of transactions can be issued on the DMA ports without receiving an acknowledgement of the first transaction completion from the interconnect. For this reason, all the DMA architectures implementing a store and forward mechanism requires a buffer at least as large as the number of packets required to hide the latency of external memory access. This number needs to be multiplied by the number of logical channels (channels that can be programmed to perform concurrent transfers) supported by the DMA, usually ranging from 1 to 16.

A key advantage of the proposed architecture is that of being capable of forwarding on-the-fly the data coming from the two physical ports, thus avoiding the usage of large FIFOs. We can afford to do this because our DMA is connected to the TCDM memory through a predictable, single-cycle latency interconnect. In this scenario, the expected wait time of a packet coming from external memory is extremely small compared to its transfer time into the TCDM. Hence the by far most common case is that the internal ports toward the TCDM absorb the data much faster than it comes in from the external port.

Some buffering in the DMA is still needed to support multiple outstanding transactions with out of order completion, but only storing control information. To implement the multiple out-of-order outstanding transaction mechanism, the proposed DMA includes a small table (OPC BUF) that associate to each transaction request identified by a transaction identifier (TID) all the information required to handle the response packet. When a new request packet is issued a new TID is generated by a dedicated module (TID GEN) and all the information is stored in the OPC buffer. Once the response packet comes back from the interconnect the control data associated to the response packet TID is fetched by the OPC buffer, and the TID is released to allow another request packet to be issued. For this reason, in the proposed DMA the only storage structure affected by the number of outstanding transaction is the OPC buffer, featuring a cost of only 25 flip-flops per entry. The only purpose of the FIFO between the TCDM UNIT and the EXT UNIT is to absorb temporary peaks of contention in TCDM that might affect the bandwidth on the external interface.

5. EXPERIMENTAL RESULTS

This section provides a quantitative evaluation of the presented DMA both from architectural and implementation viewpoint. Moreover it shows results describing the mapping of real-life applications on a multi-core cluster equipped with the proposed DMA. Finally it shows a comparison with state of the art, conventional DMAs: the CELL BE DMA and the P2012 STHORM DMA.

5.1 Performance evaluation

This section introduces an architectural evaluation of the proposed DMA performance. From the architectural viewpoint the main evaluation metrics are two. The first is the ability to guarantee an optimal transfer efficiency (i.e., a bandwidth close as much as possible to the one provided by the interconnect). This is mainly important when concurrent TX and RX transfers are accomplished, as this is the typical situation that occurs when performing double buffering. The second is the ability to provide low latency accesses. This is mainly important for small memory transfers, as for this kind of transfer the initial latency can be difficulty hidden by the overall transfer time. The overall transfer latency only partially depends on the DMA, as the latter also consists of the latency of the programming and the latency of the memory access. In the following, all the results are obtained performing cycle accurate simulations on a RTL platform including one cluster of OpenRISC processors as described in Figure 1, connected to a L2 memory that store the program and the data buffers used for the experiments. The clock cycles measurements are performed using a timer integrated on the cluster peripherals module. The 64-bits interconnect and L2 memory utilized for the experiments are capable of providing full bandwidth (8 bytes/cycle), allowing to precisely analyze the performance of the DMA itself.

Figure 4 and Figure 5 show the time and the efficiency of DMA transfers of different type and size. The transfer efficiency is calculated as the ratio between the theoretical optimal bandwidth available on the cluster bus, and the actual bandwidth achieved during simulations. The latency of the DMA, which can be seen on the left side of Figure 4, highlights that the cost of very small memory transfers (8 to 32 bytes) is very small, ranging between 15 and 20 cycles. The configuration cost of the DMA, consisting of the cost required for the processors to accomplish three store operations, ranges between 7 and 10 cycles depending on the presence or not of the addresses in the register file. The internal latency of the DMA, from the completion of the configuration to the beginning of the transfer is 4 cycles. It consist of the cycles required to cross the private DMA command queue, the common

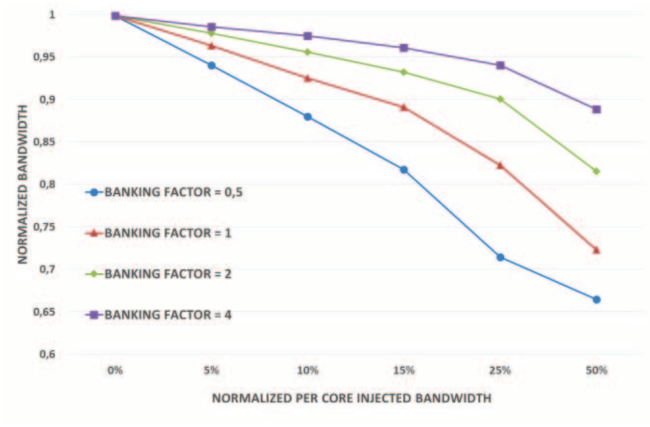

Figure 6. Bandwidth on the DMA external interface for different configurations of the cluster

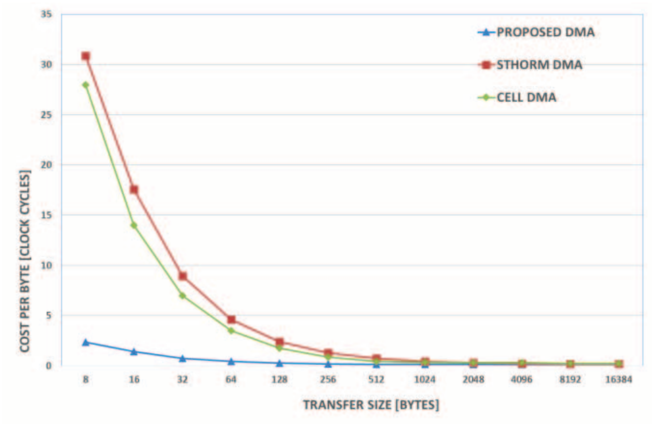

Figure 7. Cost per byte for transfers of different data size

DMA command queue, the STBus command queue and the cell queue. The remaining 6-9 cycles are required for synchronization.

Considering bandwidth, it is possible to note that for separate TX and RX operations, the proposed DMA is capable to provide the full bandwidth for DMA transfers large enough to absorb the initial latency. On the other hand, when performing concurrent RX and TX transfers, the peak transfer efficiency is close to 0.88. This is caused by a limitation on the STBus interconnect, that featuring two channels only (with respect to AXI interconnect that features 5 independent channels) requires a partial serialization of RX and TX cells (1 control cell every 8 data cells).

Since the proposed architecture speculates on the high bandwidth and low latency available toward the TCDM to avoid internal store-and forward buffers, an important situation to analyze in is what happens when a DMA transfer occurs concurrently with accesses of processors to the TCDM. In this condition, since the DMA requires full bandwidth to the TCDM interconnect, the fair arbitration between DMA and processor accesses to the TCDM may significantly reduce the DMA transfer bandwidth. Figure 6 shows the bandwidth available on the DMA external port when varying the access bandwidth of each processor toward its port on the TCDM interconnect. We study a configuration of the cluster with 16 cores, varying the banking factor from 0.5 to 4. With the described configuration of the cluster, the number of TCDM banks are 8, 16, 32, 64 for banking factors of 0.5, 1, 2, 4, respectively. On the processor side, an injected bandwidth of 10% means that the processor perform one random access on the TCDM memory every 10 cycles, 50% means one every two cycles, and so on. It is important to notice that for a strongly memory bound algorithm (for example a reordering algorithm) a processor inject at most a bandwidth of 25%. Thus 50% is an unreachable limit for a real application, added in the figure as a reference for extremely pessimistic temporary only condition.

Results show that for a memory access bounded algorithm (25% core bandwidth), a configuration with a banking factor of 2, which is a typical configuration of the cluster, is capable to provide 90% bandwidth on the external interface. For more typical computing algorithms (e.g. matrix multiplication) featuring memory access bandwidth ranging between 5% and 15%, the same configuration of the cluster is capable to provide a bandwidth larger than 95%, thus with less than 5% of penalty. It should be noted that in many complex architectures, the bandwidth provided by the interconnect never provides the 100% of theoretical bandwidth, for example due to the clock domain crossing usually adopted to break the critical path in the global interconnect. For example, the cell CELL BE interconnect is capable to provide a peak transfer efficiency of 53%, while the peak transfer efficiency of the P2012 STHORM platform is 67%, both when performing transfer across the on-chip memory hierarchy. We can conclude that the approach adopted in the proposed DMA to eliminate internal data buffering does not form a bottleneck for the memory transfer performance of clustered many-core systems.

In order to analyze the benefits of the ultra-low-latency application programming interface, joint to the queue based configuration channels, the cost per byte of data transfers of several sizes has been compared with the P2012 STHORM DMA and the CELL BE DMA. The two DMAs feature a programming cost of ~200 cycles. The results of Figure 7 show that the programming latency time is amortized on the conventional DMAs for a transfer size of 512 bytes, while the proposed DMA feature significantly better performance for transfer sizes ranging of 8 bytes to 256 bytes leading to speed-ups ranging from ~11x to ~6x, respectively.

5.2 Implementation results

This section analyzes the implementation of the proposed DMA engine in STMicroelectronics 28 nm UTB FD-SOI technology [17]. The results reported in this section, refers to the post-synthesis implementation of the DMA, as well as all the cluster, performed with Synopsys design compiler configured in topographical mode. All the estimations are carried out assuming a supply voltage of 1.0V, transistors operating in worst case at a temperature of 125°C, which is the worst case commercial condition for the FD-SOI technology at the considered supply voltage. The DMA is capable to work at the maximum frequency of 1.72 GHz, with a dynamic power density of 12 µW/MHz, and a leakage power consumption of 280 µW at 25°C in its largest configuration. The dynamic power has been estimated extracting the traces from a back-annotated simulation on the post synthesis netlist of the cluster, and performing the power simulation with Synopsys PrimeTime PX.

As the proposed DMA targets the integration on a multi-core cluster of processor, one important aspect is to analyze its scalability with the number of cores. To scale with the number of cores, the private command queue are replicated enabling each

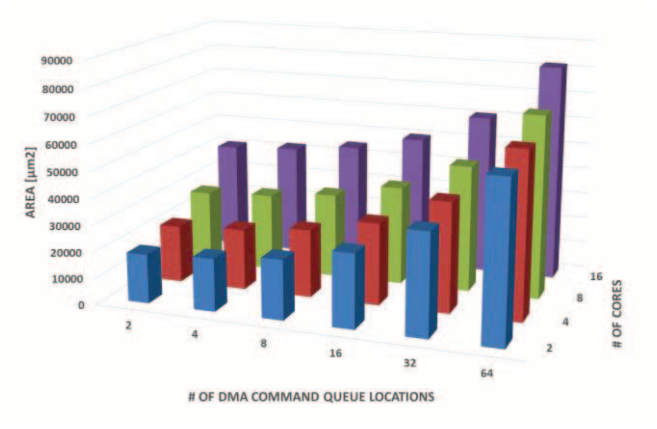

Figure 8. Exploration of the DMA area by number of DMA command queue locations and number of cores

processor to access the DMA logical channels in a transparent way. For this purpose we consider the cluster to be scalable up to 16 processors, as it has been studied that more complex clusters form computational and implementation inefficiencies independent by the design of the DMA itself. Another key parameter of the proposed of the DMA is the size of the common DMA command queue. Indeed, the size of this component affects the number of DMA outstanding commands that can be issued by the processors before being blocked. Figure 8 shows the results of the exploration. In its smaller configuration (2 cores, 2 command queue) the area of the DMA is ~20000 µm^2, roughly the area of one OpenRISC core without instruction cache. When we increase the size of the common command queue the area increases up to ~60000 µm^2. This is caused by the fact that each location of the DMA command queue stores the external address, TCDM address and command, requiring 47 flip flops for each location. On the other hand, when we increase the number of cores attached to the DMA up to 16, maintaining unchanged the size of the command queue, the area of the DMA is ~40000 µm^2. It should be noted that each private command queue contains two locations so that the overall number of outstanding DMA commands in this configuration is 18 (2 per private queue + 2 in the common queue).

Considering the integration of the DMA at system level, Figure 9 and Figure 10 show the area and power breakdown of the cluster, respectively. The considered configuration of the cluster includes 16 OpenRISC cores with 4KB of instruction cache each, and 32 banks of TCDM memory 8KB each, for an overall TCDM size of 256KB. It is possible to note that with this cluster configuration the DMA occupy only the 5% of the overall cluster area, and only the 3% of the overall power, where the power of the cluster has been estimated when running a matrix multiplication application featuring overlapped computation and DMA transfers.

With respect to a conventional DMA such as the P2012 STHORM DMA, the area of the proposed DMA is more than twice smaller, even if the P2012 DMA does not feature any specific multi-core features. Moreover to make available to the processors four logical channels overall, the P2012 STHORM clusters has been equipped with 2 DMAs. In this perspective, the proposed DMA provides 16 independent logic channels, the same bandwidth with 1/4 of the area and 1/5 of the power consumption of the P2012/SHORM DMA subsystem. With respect to the

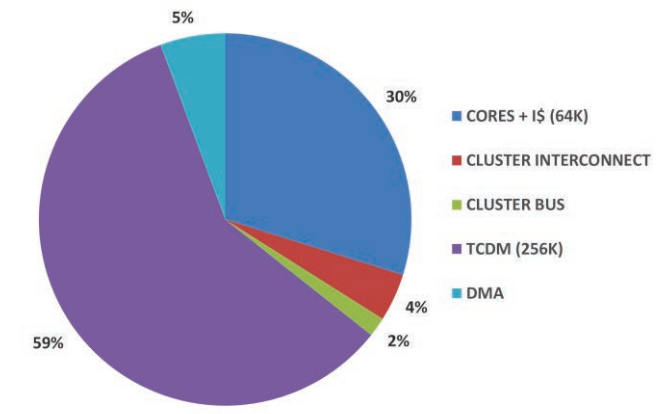

Figure 9. Area breakdown of a 16-cores, 32-TCDM banks cluster integrating a DMA featuring 96 outstanding commands

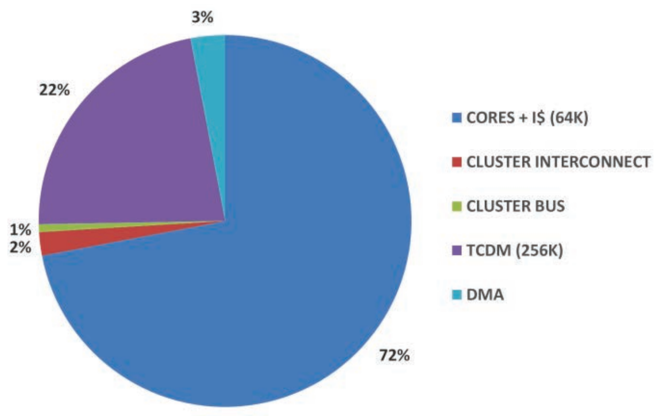

Figure 10. Power breakdown of a 16-cores, 32-TCDM banks cluster integrating a DMA featuring 96 outstanding commands

CELL BE DMA, the proposed DMA features 1/5 of the area, scaled to the 28nm technology utilized as reference.

5.3 System level exploration

In order to validate the benefits of the proposed DMA from a system-level viewpoint, a functional model of the IP has been integrated by the authors on a virtual platform of the P2012 STHORM many core fabric [13].

The virtual platform models a complete system featuring four 16-cores clusters and the L2 memory, running at 430 MHz. Moreover it models the external memory access subsystem composed of a 400MHz DDR memory controller and an external DDR memory featuring 500 clock cycles of latency. Since the focus of this work is the evaluation of the DMA itself, and not an analysis of the global communication system, only one cluster is considered in this context. To evaluate the DMA we run two OpenCL applications with different features, matrix multiplication and face detection. Matrix multiplication is an application which is dominated by date the transfers (in idle mode 75% of the time waiting for data). Face detection is dominated by computation. The particularity is that there is a first small conversion kernel that takes data from L3 and store the result to L2. Then the detect kernel is fully working between L2 and L1.

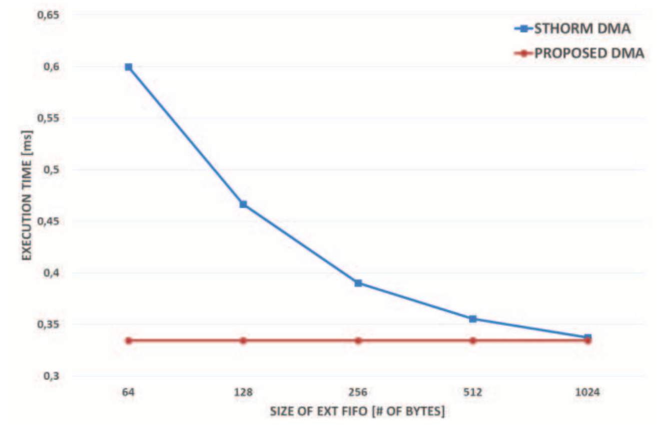

Figure 11. Execution time of a matrix multiplication application increasing the size of the DMA buffers

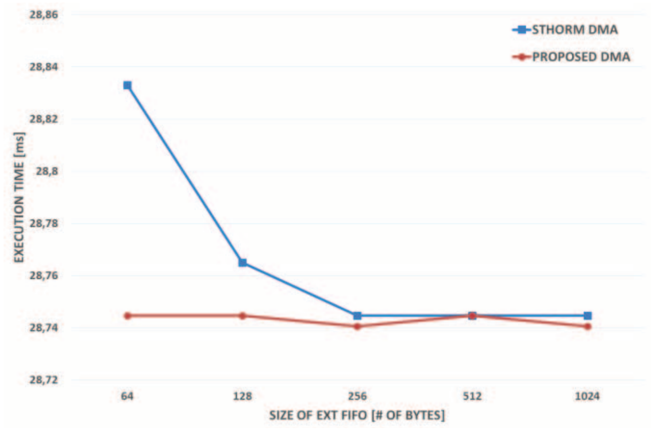

Figure 12. Execution time of the face detection application increasing the size of the DMA buffers

Figure 11 and Figure 12 show the results of the exploration on matrix multiplication and face detection. They compare the execution time of the two applications on a platform including a conventional DMA and a platform including the proposed DMA on its clusters, when increasing the size of the DMA data FIFOs that determines, in a conventional implementation, the number of outstanding transactions that can be performed by the DMA to hide the external memory latency. As expected, when running face detection, the data transfers occupy only a minimal time of the overall application time. Thus, even for a small size of the FIFOs, the benefits of the proposed DMA are smaller than 0.1%. On the other hand, when running matrix multiplication, for a FIFO size of 64 bytes the platform including the proposed DMA outperforms by 1.76x the platform with the conventional DMA, while the conventional DMA can reach the same performance only with a FIFO size of 1024 bytes.

The results show that the main advantage of the proposed DMA with respect to a conventional DMA (i.e. P2012 STHORM DMA), is that the needed FIFO size in the silicon does not depends on the system characteristics (latency, bandwidth, size of TCDM) and on the applications characteristics (memory bound, computation bound, latency bound). In a conventional implementation, the more bandwidth and latency we have, the bigger FIFO we need. On the proposed DMA it is not required to choose the worst case. For this reason, the proposed DMA is able to provide, on the considered applications, the same performance of the P2102 STHORM DMA with a FIFO size of 64 bytes (instead of 1024 for the considered applications) resulting in 4x improvement in terms of silicon area 5x improvement in terms of power.

6. CONCLUSION

This paper presents an ultra-low-latency lightweight DMA targeting the integration within a class of modern many-core clustered architecture. The main features of the proposed DMA engine are three. First, it provides a hardwired mechanism that allow multiple cores to enqueue an arbitrary number of commands to the DMA programming channels in a completely transparent way from the software viewpoint. Second it uses a simple, ultra-low latency programming interface that allows processors to exploit their software based programmability to improve the flexibility of DMA transfers. These two features allow to reduce the programming latency by a factor of 20x with respect to conventional DMAs. We believe that this improved configuration latency and flexibility will open new possibilities for applications using very small, non-uniform transfer patterns that are currently strongly affected by the memory access latency. Third, thanks to its optimized architecture it allows to support multiple outstanding transactions required to hide the huge latency of external memory accesses without a large internal memory for temporary storage of incoming or outgoing data. Thanks to this feature, the proposed DMA outperforms by 1.76x the P2012 STHORM DMA when configured with 64-bytes FIFO with 25% of the silicon area cost and consuming 20% of the power.

7. ACKNOWLEDGMENTS

This work is supported by European FP7 Project Phidias (318013) and the FP7 ERC project MULTITHERMAN (291125).

8. REFERENCES

[1] Banakar R.; Steinke S.; Lee B.; Balakrishnan M; Marwedel. P. 2002. Scratchpad Memory: A Design Alternative for Cache On-chip memory in Embedded Systems. *International Symposium on Hardware/Software Co-design (CODES)*.

[2] Khunjush, F.; Dimopoulos, N.J. 2008. Extended characterization of DMA transfers on the Cell BE processor. *IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008.*

[3] Harting, R. C; Parikh, V.; Dally W. J. 2012. Energy and Performance Benefits of Active Messages. *Technical Report 131, Concurrent VLSI Architectures Group, Stanford University*.

[4] http://www.ti.com/lsds/ti/dsp/keystone_arm

[5] http://www.nvidia.com/object/tegra.html

[6] www.qualcomm.com/snapdragon

[7] Flachs, B.; Asano, S.; Sang H.Dhong; Hofstee, H.P.; Gervais, G.; Roy Kim; Le, T.; Peichun Liu; Leenstra, J.; Liberty, J.; Michael, B.; Hwa-Joon Oh; Mueller, S.M.; Takahashi, O.; Hatakeyama, A.; Watanabe, Y.; Yano, N.; Brokenshire, D.A.; Peyravian, M.; Vandung To; Iwata, E. 2000. The microarchitecture of the synergistic processor for a cell processor. *IEEE Journal of Solid-State Circuits*.

[8] Zhang N. Resolving a L2-prefetch-caused parallel nonscaling on Intel Core microarchitecture. 2011. *Journal of Parallel and Distributed Computing*.

[9] Sancho, J.C.; Kerbyson, D.K., "Analysis of double buffering on two different multicore architectures: Quad-core Opteron and the Cell-BE," *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium*, vol., no., pp.1,12, 14-18 April 2008.

[10] Rossi, D.; Campi, F.; Spolzino, S.; Pucillo, S.; Guerrieri, R. 2010.A Heterogeneous Digital Signal Processor for Dynamically Reconfigurable Computing. *IEEE Journal of Solid-State Circuits.*

[11] Ramirez, A.; Cabarcas, F.; Juurlink, B.; Alvarez Mesa, M.; Sanchez, F.; Azevedo, A.; Meenderinck, C.; Ciobanu, C.; Isaza, S.; Gaydadjiev, G. 2010. The SARC Architecture. *IEEE Micro*.

[12] Xiaowen Chen; Zhonghai Lu; Jantsch, A.; Shuming Chen. 2010. Supporting Distributed Shared Memory on multi-core Network-on-Chips using a dual microcoded controller. *Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[13] Benini, L.; Flamand, E.; Fuin, D.; Melpignano, D. 2012. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. *Design, Automation & Test in Europe Conference & Exhibition (DATE)*.

[14] Bauer M.; Cook H.; Khailany B. 2011. CudaDMA*:* optimizing GPU memory bandwidth via warp specialization. *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*.

[15] NVIDIA Corporation. 2011. NVIDIA CUDA C programming guide, v4.0. http://www.nvidia.com/.

[16] The Khronos OpenCL Working Group. 2011. OpenCL - The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/.

[17] Mazure, C.; Ferrant, R.; Nguyen, Bich-yen; Schwarzenbach, W.; Moulin, C. 2010. FDSOI: From substrate to devices and circuit applications. 2010 *Proceedings of the ESSCIRC*.