MCAPI-compliant Hardware Buffer Manager Mechanism to Support Communication in Multi-Core Architectures

Thiago Raupp da Rosa Thomas Mesquida Romain Lemaire Fabien Clermidy

{firstname.lastname@cea.fr}
Univ. Grenoble Alpes, F-38000 Grenoble, France
CEA, LETI, MINATEC Campus, F-38054 Grenoble, France

Abstract—High performance and high power efficiency are two mandatory constraints for multi-core systems in order to successfully handle the most recent applications in several fields, e.g. image processing and communication standards. Nowadays, hardware accelerators are often used along with several processing cores to achieve the desired performance while keeping high power efficiency. However, such systems impose an increased programming complexity due to the lack of software standards that supports heterogeneity, frequently leading to custom solutions. On the other hand, implementing a standard software solution for embedded systems might induce significant overheads. This work presents a hardware mechanism in co-design with a standard programming interface (API) for embedded systems focusing to decrease overheads imposed by software implementation while increasing programmability and communication performance. The results show gains of up to 97% in latency and an increase of 40 times in throughput for synthetic traffics and an average decrease of 95% in communication time for an image processing application.

I. INTRODUCTION

The application constraints driving the design of embedded systems are constantly demanding higher performance and lower power consumption. Since pushing single-thread performance is not viable to scale microprocessor performance up [1], multi-core architectures are being widely used to meet these requirements. To further increase power efficiency, embedded systems are employing hardware accelerators to execute dedicated tasks. Due to the increasing complexity of underlying systems, developing embedded applications is becoming a key challenge for two main reasons [2]. First, embedded application workload will continue to grow in diverse fields (e.g. spatial, automotive, etc.) [16]. Second, the software technologies are not evolving as fast as hardware architectures, leaving a gap in the full system design [1].

In order to decrease this gap, several works present implementation of standard APIs (Application Programming Interface) [3][4]. However, the induced performance overhead might compromise the application performance and, consequently, violate the initial constraints. A standard specifically designed for embedded systems could decrease this overhead, though none is widely used. A promising solution of software standards for embedded system is proposed by the Multi-Core Association (MCA) [16]. The Multi-core Communication API (MCAPI) is presented in works such as [5][6][7]. Yet, as shown by these works, only MCAPI itself cannot extract the full performance provided by the hardware architecture, leading to sub-optimal performance.

The addition of hardware mechanisms to handle inter-process communication can increase the overall performance by speeding-up the communication. On the other hand, most of the time, these mechanisms are not flexible to couple with different types of architectures and/or do not take into account the increased complexity in software development to manage them. Thus, it is mandatory to co-design hardware and software to increase the programmability of multi-core architectures and achieve expected performance requirements. Nevertheless, the use of a standard software API is essential for code reuse and compatibility.

Therefore, the main contribution of this work is to propose the co-design of a hardware mechanism to handle data transfers in multi-core architectures and MCAPI. The mechanism handles inter-process communications in a socket-like fashion. It is demonstrated that the mechanism can be programmed by the software API with no additional complexity and the performance is improved when compared to a DMA solution.

The rest of this paper is organized as follow: Section II presents related work regarding hardware communication mechanisms and implementation of standard APIs for embedded systems, showing that limited effort has been made to co-design software and hardware. Section III describes the reference architecture of this work and solutions for improvement. Section IV presents the proposed hardware mechanism and MCAPI mapping over the architecture. Finally, Section V shows the experiments and results, while Section VI concludes the paper.

II. RELATED WORK

The communication performance in multi-core architectures is subject of many works. Most of these works present solutions to improve the performance by implementing hardware mechanisms that perform operations usually handled by software, or by improving a given point of the target architecture. A hardware mechanism for distributed memory architectures is proposed in [8]. The mechanism is composed of several components that handle the communication of the processing nodes (Memory Sever Access Points – MSAP), a control network and a data network. Each MSAP has several input and output ports connected to dedicated FIFOs for data transfers. The connection is created by linking an input to an output port. Though the results show the scalability of the proposed mechanism and a performance increase for the evaluated applications, there is no mention about software support or how the application is able to benefit from the hardware mechanism. References [9] and [10] present solutions to decouple communication and computation for message passing systems. Yet, the programmability of the architectures targeted in these works is not addressed. In [11], a hardware mechanism based offering the possibility to program transfers by informing task IDs with a software support is presented. Still, the software solution is provided by a set of custom, which limits code portability and reusability.

On the other hand, the programmability issue for multi-core architectures is addressed in works such as [12] and [13]. In [12] it is proposed a communication aware programming approach that enables to specify communication requirements at application level using a standard programming language. These requirements are handled by the operating system, which configures the hardware accordingly. Although providing a solution that increases the application performance in multi-core architectures, this approach relies on the target architecture to provide mechanisms that can boost the performance. Reference [13] introduces a hardware mechanism for data transfers between processing cores, hardware accelerators and local memories inside a cluster. The programmability of the target architecture is increased by using a standard software API. However, the hardware mechanism provides support only at local interconnection, without addressing the communication between processes executing in different clusters.

Thus, we present a hardware mechanism to manage inter-process communication co-designed with a standard software API. Compared to the works previously mentioned, the proposed solution increases the global system performance while maintaining the overall system programmability.

III. System Architecture

The architecture used as reference is composed of several clusters connected by a Network-on-Chip (NoC). Each cluster comprises several CPUs (MIPS R3000 core) with their respective private memories, input and output NoC modules (CPU subsystem), a Shared Memory, a Network Interface (NI) and a Communication and Synchronization subsystem, as depicted in Fig. 1.

The CPU subsystem sends and receives control messages through the output and input modules, respectively. In turn, the Communication and Synchronization subsystem is employed for data transfers. This subsystem is composed of a DMA and an Event Synchronizer (ES) [14]. The DMA is able to perform data transfers through requests issued by the CPUs. These

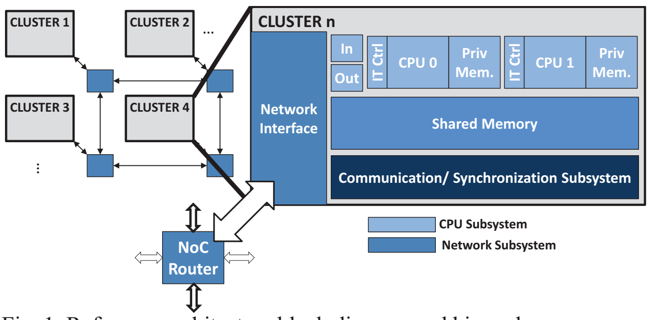

Fig. 1. Reference architecture block diagram and hierarchy.

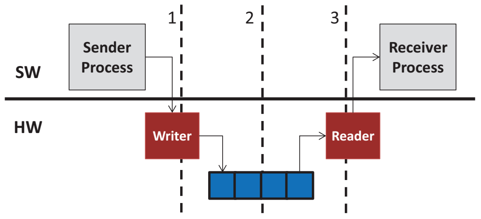

Fig. 2. Partitioning schemes for implementation of a buffer management solution in hardware.

requests provide usual parameters: source buffer address, destination buffer address, transfer size and transfer identification (ID). The Event Synchronizer is a hardware module that works as a programmable interrupt controller. It stores the state of CPU communication activities in synchronization event registers (SER). A synchronization mask can be programmed by the CPU and, when the value of a SER matches the value stored in the synchronization mask, an event is generated. Thus, the CPU can be notified when the transfer with a given ID is completed (e.g. for buffer reallocation) by setting the synchronization mask accordingly and by the DMA signalling the Event Synchronizer that this given transfer was completed.

From the software point of view, a set of functions is provided for inter-process communication. The communication is handled through FIFOs, which benefits a wide range of data-flow applications. The FIFOs are placed in the Shared Memory and controlled by software routines. Thus, the application does not need to manage local or remote addresses in data transfers, decreasing programming complexity. Additionally, as the address mapping is global, the set of shared memories can be seen as a single distributed shared memory with non-uniform memory access time (NUMA), making it possible for any CPU to write in any FIFO of any cluster.

However, the implementation of a FIFO mechanism in software creates processing and traffic overheads. Although the data transfer is performed by the DMA, the FIFO status checking, pointer exchanges and pointer updates have to be handled by the CPU and impact the performance during data transfers. Moreover, when sender and receiver processes are in different clusters, the pointers must be exchanged through the NoC, resulting in traffic overhead and higher communication latencies. Thus, in order to completely decouple computation and communication and increase system performance, a hardware mechanism to manage buffers/FIFOs with flexible configuration and low control overhead should be exploited.

Fig. 2 illustrates how the mechanism could work: two hardware blocks, Writer and Reader, are introduced to manage data access to the shared FIFO buffer in order to detach the low-level communication management from Sender and Receiver processes. Three partitioning schemes between clusters can be considered, depicted by the numbers 1, 2 and 3. Each number represents a partitioning of blocks among sending and receiving clusters: left side of the dotted line is implemented in the sender side, while the right side is implemented in the receiver side.

Partitioning 1 and 3 are equivalent by implementing the FIFO buffer in one or the other cluster with the respective blocks for writing and reading. On the other hand, the partitioning 2 completely separates the writing and read blocks in their respective clusters. Any of the partitioning eliminates the processing overhead of controlling the FIFO from the CPU. However, the partitioning 2 presents a higher overhead in network traffic, since the Writer and Reader blocks must exchange pointer addresses. Furthermore, partitioning 3 requires remote reads while partitioning 1 requires remote writes, which presents a better performance in the reference architecture. Thus, the FIFO communication mechanism presented in this paper is implemented using the partitioning scheme 1.

IV. BUFFER MANAGER MECHANISM

The proposed mechanism, called Buffer Manager Mechanism (BMM), targets to decrease the computation and traffic overheads in inter-process communication. This is achieved by providing fast configuration, accelerating FIFO management by using hardware implementation, avoiding pointer exchange to decrease the number of control messages and abstracting addresses through port IDs. The BMM replaces the DMA in the Communication and Synchronization subsystem to handle data transfers. The CPU can access the BMM through address-mapped registers. However, contrary to the DMA, which handles only send operations, the BMM handles write and read requests.

The CPU generates the requests by writing information such as source and destination addresses and transfer size in specific memory-mapped registers. The requests are divided in three types: ***address-based***, ***direct stream-based*** and ***indirect stream-based***. The ***address-based*** request is used to perform transfers in a DMA-fashion way, where the source and destination addresses are explicitly provided. This request replaces the requests previously handled by the DMA. On the other hand, the ***stream-based requests*** abstracts the destination addresses through connection IDs, herein called ports, implementing a connection-based communication, where resources are allocated previously to data exchange. In the ***direct stream-based*** request a *single word* of 32-bits is transmitted from a source port ID to a target port ID, while in the ***indirect stream-based*** request a *buffer* of 32-bits words is transmitted from a source port ID to a target port ID.

Table I presents the write and read requests encoding. The first row represents the address bits, while the second row represents the encoding of the respective bit(s). The BMM is accessed by setting the address bits 22, 21 and 20 to the value "011". The request type (read or write) is encoded in the bit 19. The bits 18 and 17 are used to inform the request type (address-based, indirect or direct stream based), while the bit 16 informs the request has been completed in cases where it takes more than one write. The port ID is encoded from the bit 15 to the bit 8 and informs to the respective block the local port to which the request is related. The CPU ID is encoded between the bits 7 and 4 and is used to store the requests in their respective queues. The address bits not mentioned are set to value '0'.

The number of writes required to generate a request vary according to the request type. The ***address-based transfer*** request is used only for sending data and requires three writes in the memory-mapped registers: source address, destination address and transfer size. The stream-based requests require one and two writes for direct and indirect requests, respectively. For ***direct stream-based*** requests the data to be transmitted is written (write requests), or the received data is returned (read requests). In ***indirect stream-based requests*** the written data are the source buffer address (write request) or target buffer address (read request), and the transfer size.

The BMM is composed of four main blocks: Buffer Manager Interface (BMI), Buffer Manager Read (BMR), Buffer Manager Write (BMW) and Credit Manager (CM). The BMI is used only in the sender side, while the BMR and BMW are used only in the receiver side. The FIFO control is performed in hardware by BMR and BMW modules. These blocks decouple the FIFO control of sender and receiver processes, as previously discussed in Section III. From the sender point of view, no knowledge about the FIFO structure is needed. However, a flow control police must be implemented to avoid sending data when there is no available space to receive it. Thus, the CM implements a credit-based flow control. Additionally, from the software point of view, there is no need to manage remote addresses since the sender process exclusively uses connection IDs, decreasing programming complexity.

The BMI block is responsible for handling write requests created by the Sender process, while the BMR is responsible for handling read requests created by the Receiver process. The BMW receives the data sent by the BMI and store it in the respective FIFO accordingly the target port ID information. The CM is responsible for sending and updating credits. Credits are sent always that a read request is completed or a programmable threshold is reached. The credit update is performed when a write request is completed (decrease of credits) or when a credit update message is received (increase of credits). Finally, three table structures are used by the BMM to control the communication requests: Connection Table, which stores the remote port ID connected to its respective local port ID; Credit Table, which stores the number of available credits for a respective local port; and Buffer Table, which stores the read and write pointers of the FIFO used by the respective port. In this work the tables are implemented as registers and the FIFOs are placed in the cluster Shared Memory. However, since the mechanism is flexible, the tables could be placed in a memory and FIFOs could be implemented as a hardware block.

Fig. 3 depicts how the mechanism works during an ***indirect stream-based*** request. In both sides there is a set-up phase, which consists of the CPU initializing the Connection Table with the remote port IDs that will be sending/receiving data for the respective local ports. Next, in the Sender cluster, the CPU writes data in a buffer in the local memory and creates a write request for the BMI. Then, the BMI retrieves the remote port ID in the Connection Table and the available credit for the local port ID. In case of available credits the data packet is sent through the NoC with the target port ID information, and the CM is notified to update credits for the respective local port ID.

In the Receiver cluster, the data is received by the BMW, which accesses the Buffer Table based on the target port ID and writes the data in the respective FIFO. At a given moment, the

TABLEI - Address encoding used to create a request.

19	18 17	16	15 8	7 4
R/W	**T**	**E**	**port ID**	CPU ID
R/W – Read or Write operation; T – Type; E – End of Request;				

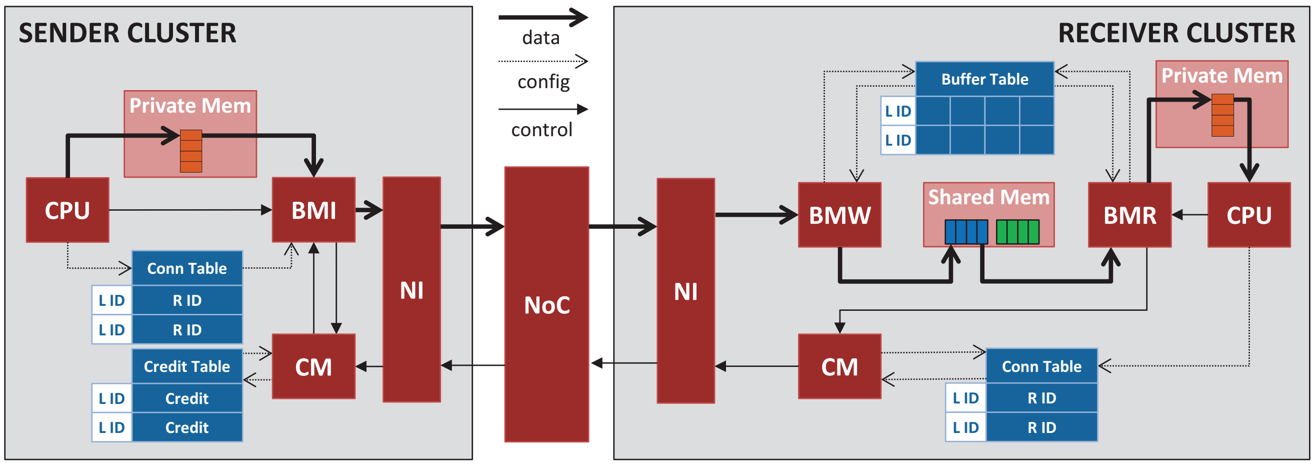

Fig. 3. Buffer Manager Mechanism functional scheme.

CPU in the Receiver cluster creates a read request for the BMR. Then, the BMR access the Buffer Table based in the local port ID and copies the data from the respective FIFO to the target address. Finally, the BMR notifies the CM to generate credits to the remote port ID. Both BMI and BMR have queues to store the requests created by each CPU independently. The request selecting process is performed using a round-robin algorithm. Nevertheless, a request is selected only if the needed number of credits is available.

A. *MCAPI Mapping*

The Multi-core Communication API (MCAPI) defines an inter-task communication and synchronization API for embedded systems specified by the Multi-core Association [16]. Its main objective is to provide code portability, scalable communication performance and low memory footprint. Several works present an implementation of this standard [1][5][6][7][15], showing this is a promising solution for communication standard in embedded systems.

The MCAPI specification defines two levels of hierarchy: domains and nodes. A domain is composed of one or more nodes. A node is defined by the specification as an independent thread of control, i.e. an entity that can exclusively execute a sequential flow of instructions, such as a process, thread, processor, hardware accelerator, etc. As the reference architecture is divided in clusters, each one represents a domain, while each CPU inside a cluster represents a node.

The communication is performed between nodes through a pair of socket-like termination points called endpoints. Three communication modes are supported: messages, packet channels and scalar channels. The main difference between messages and packet or scalar channels is the flexibility, allowing data transmission between nodes without having to establish a connection. On the other hand, channels provide a higher performance due to the resources previously allocated in the connection set-up phase. Both packet and scalar channels transmit data in a FIFO manner. The difference between packet and scalar channel is the size of data transfers. Packet channels are able to transfer data chunks of variable sizes, while Scalar channels support transfers of 8, 16, 32 or 64-bits only.

The Buffer Management Mechanism has been specified so that the implementation of MCAPI primitives can be achieved with a limited software complexity. In the target architecture, the endpoint structure is mapped in a 32 bits integer and encodes the port, the CPU and the Cluster identification numbers (IDs). The endpoint ID is directly mapped to an available port ID in the BMM. Thus, when executing MCAPI functions, the decoding of an endpoint tuple to a port ID is a simple shift. Moreover, the three communication modes can be fully handled by the BMM due to the different request types it supports. MCAPI messages specify source and target address and a transfer size, which can be handled with address-based requests. Packet and scalar channels require dedicated connections and are handled through channel identifiers. These characteristics are supported by indirect and direct stream based requests. Thus, the MCAPI implementation only needs to decode the channel identifier into a port ID and create the respective requests to perform send and receive operations. Therefore, as the BMM configuration can be performed with a maximum of 3 CPU write operations in memory-mapped registers, the overhead induced by the MCAPI implementation is limited, thus, preserving performance.

V. EXPERIMENTS

A. *Environment Setup*

The proposed mechanism is evaluated on a simulation environment based on a SystemC/TLM (Transaction-Level Model) description with timing annotation. The evaluations are conducted in order to compare the BMM and DMA performances with their respective MCAPI implementations. The CPUs and peripherals are running at 200 MHz and the NoC at 500 MHz in all scenarios. The “ping-pong” benchmark [7] is used to characterize throughput and latency, while the system performance is evaluated with the application *susan* [18] by measuring communication and execution times. The following sections provide details on the scenarios used to collect the results and their respective performance figures.

B. *Software Implementation*

The main difference between the software layer implementation for the DMA and BMM is the FIFOs management. When using the DMA, a set of software functions referred as FIFO API performs the FIFO management and is used by MCAPI to implement send and receive calls. The FIFO API size is around 1.5KB, corresponding to 5% of the total

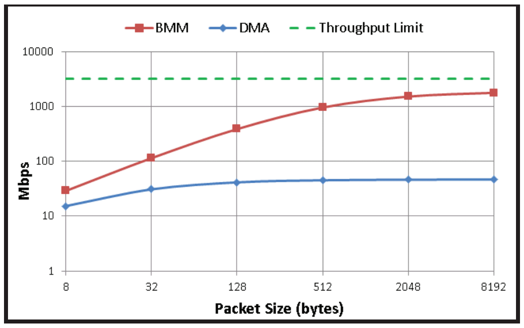

Fig. 4. Throughput comparison between BMM and DMA.

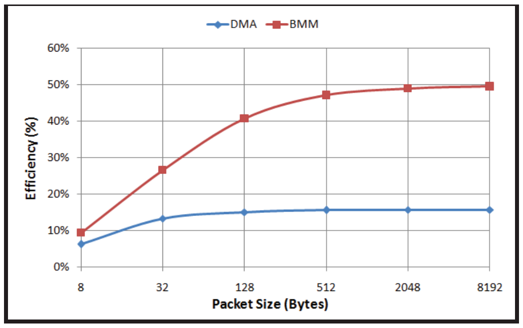

Fig. 5. Throughput efficiency comparison between BMM and DMA.

code size (considering the low level software functions that access the reference architecture and the MCAPI implementation). On the other hand, when using the BMM, the MCAPI layer directly creates write and read requests to perform sends and receives, decreasing the total size of software code. For instance, the MCAPI send and receive functions are implemented with 125 and 271 instructions, respectively, when using the BMM. This represents a decrease of 49% in the number of instructions when compared to the FIFO API implementation.

C. *Throughput*

The throughput is measured with two processes in the ping-pong application. However, only the first phase is executed, i.e. the data is sent only in one direction. Fig. 4 shows the throughput for DMA and BMM related to the packet size. The total amount of data transmitted in this scenario was 16KB. The throughput limit represented by the dotted line is calculated accordingly to the architecture characteristics. Taking into account that the MIPS CPU is running at 200 MHz and the fact that it takes 2 cycles to write each data at the bus, the maximum achievable throughput is 3200 Mbps. The result shows that the communication throughput using the DMA and FIFO API ranges from 15 Mbps to 46 Mbps, compared to a range of 29 Mbps to 1780 Mbps when using the BMM. This represents an increase of 40 times using the BMM when compared to the solution including the FIFO API and DMA. Nevertheless, other scenarios with higher amounts of data were simulated. The results show that the maximum throughput achieved with BMM gets closer to the theoretical limit as the amount of data transmitted is increased, while the maximum throughput using the DMA stays constant.

The throughput efficiency is evaluated in Fig. 5. The efficiency can be calculated as the ratio between the total number of user-data bytes sent and the total number of bytes sent through the NoC, which includes the protocol overhead. The evaluation was performed for a total of 16KB of user-data transmitted. The total number of bytes takes into account both producer and consumer processes, considering credit packets and channel set-up messages. The result shows that the BMM dramatically increases the throughput efficiency, since the overhead induced by the software implementation due to the FIFO pointer exchanging is removed.

D. *Latency*

The latency comparison is performed with two processes in the ping-pong benchmark and measures the round-trip-time of the communication. Two scenarios were evaluated: (i) a single packet of variable size and (ii) 8KB of data transmitted varying the packet size. The results are presented in Fig. 6 and Fig. 7 respectively. Fig. 6 shows that the BMM is able to decrease the latency for small packet sizes around 60% and up to 97% for bigger packet sizes. In Fig. 7 the total latency is higher for small packet sizes due to the higher number of packets needed to complete the transmission. The decrease in latency is similar to the one presented in Fig. 6, with higher gains for bigger packet sizes.

E. *Communication and Execution Times*

The communication and execution times are evaluated using the application *susan*. This application is used for recognizing corners and edges in images. The application was evaluated with different number of CPUs processing the input file (256x256 pixel image). All the CPUs run the same code, varying the amount of data that each one processes. Fig. 8 presents the results for the average communication time related

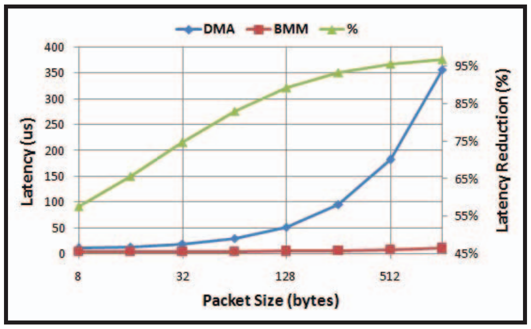

Fig. 6. Round-trip time latency for different packet sizes.

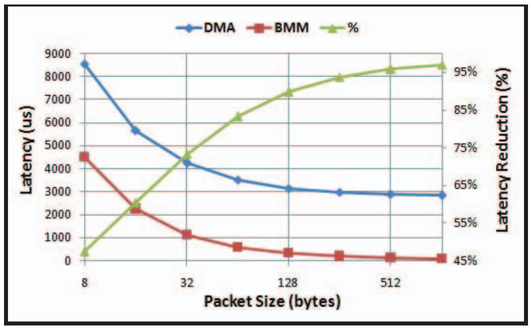

Fig. 7. Round-trip time for a fixed number of data at different packet sizes.

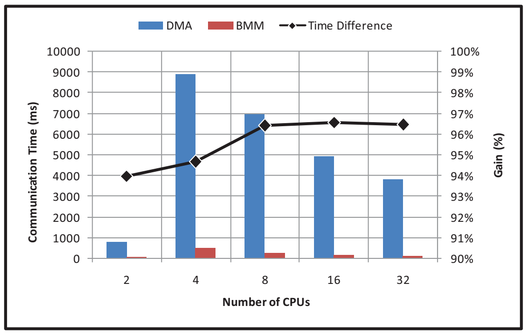

Fig. 8.Communication time comparison between BMM and DMA.

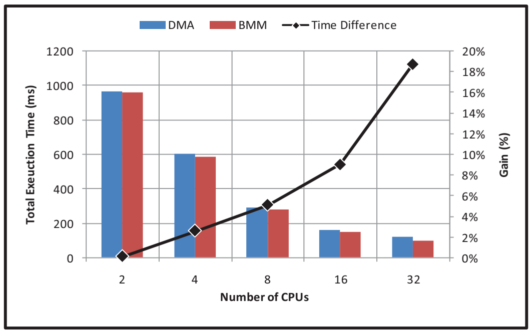

Fig. 9.Total execution time comparison between BMM and DMA.

to the number of CPUs used to process the image. The BMM is able to decrease the communication time up to 96.5%. Fig. 9 shows the evaluation of the total execution time for the same scenarios. It is possible to notice that, with fewer CPUs, the significant decrease in the communication time presented in Fig. 8 has lower impact in the application performance, since the CPUs spend considerably more time processing than communicating. However, as the number of CPUs increase, the gain in the application performance also becomes significant, due to the higher impact of the communication time over total execution time.

VI. CONCLUSION

This work presented a hardware mechanism, called BMM, for managing FIFO-like communication in multi-core architectures co-designed with MCAPI. The mechanism is flexible, being programmed through memory-mapped registers. This characteristic allows the BMM to be used not only for CPUs but also with other hardware accelerators. The buffer table structure allows the FIFOs to be dynamically allocated in the system following a connection initialization phase. The results show that the mechanism was integrated with the MCAPI implementation with no software complexity increase. Moreover, the performance gains provided by the BMM are very significant in terms of throughput (increase of 40 times), latency (decrease up to 97%), and application performance (decrease of total execution time up to 19%). Future works will be oriented on table structures resource sharing in order to limit hardware overhead and extension of the buffer mechanism to be interfaced with hardware accelerators in order to exploit the heterogeneity aspect of MCAPI.

ACKNOWLEDGMENT

This work was supported by the European CATRENE program in the frame of the CA112 HARP project.

REFERENCES

[1] J.Holt, A. Agarwal, S. Brehmer, M. Domeika, P. Griffin and F. Schirrmeister, "Software Standards for the Multicore Era," Micro, IEEE, vol.29, no.3, pp.40,51, May-June 2009.

[2] A.M. Jallad, L.B. Mohammad, "Comparative analysis of middleware for multi-processor system-on-chip (MPSoC)," International Conference on Innovations in Information Technology (IIT), 2013, pp.113,117, 17-19 March 2013.

[3] J.L. Abellán, J. Fernandez, M.E. Acacio, "CellStats: A Tool to Evaluate the Basic Synchronization and Communication Operations of the Cell BE," PDP 2008, pp.261,268, 13-15 Feb. 2008.

[4] W.H. Minhass, J.Öberg, I. Sander, "Design and Implementation of a Plesiochronous Multi-Core 4x4 Network-on-Chip FPGA Platform with MPI HAL Support," Proceedings of the 6th FPGA world Conference, 2009, pp 52-57.

[5] L. Matilainen, E. Salminen, and T. Hämäläinen, "MCAPI abstraction on FPGA based SoC design," Proceedings of the Annual FPGA Conference, 2012.

[6] L. Matilainen, L. Lehtonen, J. Maatta, E. Salminen and T. Hamalainen, "System-on-Chip deployment with MCAPI abstraction and IP-XACT metadata," International Conference on Embedded Computer Systems (SAMOS), 2012, pp.209,216, 16-19 July 2012.

[7] C. Clauss, S. Pickartz, S. Lankes, and T. Bemmerl, "Towards a Multicore Communications API Implementation (MCAPI) for the Intel Single-Chip Cloud Computer (SCC)," 11th International Symposium on Parallel and Distributed Computing (ISPDC), 2012, pp.148,155, 25-29.

[8] Sang-Il Han; A. Baghdadi, M. Bonaciu, Soo-Ik Chae, A.A. Jerraya, "An efficient scalable and flexible data transfer architecture for multiprocessor SoC with massive distributed memory," Design Automation Conference, 2004, pp.250,255, 7-11 July 2004.

[9] D. Buono, T. De Matteis, G. Mencagli, M. Vanneschi, "Optimizing message-passing on multicore architectures using hardware multi-threading," Euromicro International Conference on Parallel, Distributed and Network-Based Processing, 2014, pp.262,270, 12-14 Feb. 2014.

[10] S. Gao, B. Huang, R. Sass, "The Impact of Hardware Communication on a Heterogeneous Computing System", FCCM 2013, pp. 234.

[11] S. Kumar, M.T.A. Djie, R. van Leuken, "Low Overhead Message Passing for High Performance Many-Core Processors," 1st International Symposium on Computing and Networking (CANDAR), 2013, pp.345-351, 4-6 Dec. 2013.

[12] J. Heisswolf et al., "CAP: Communication aware programming," Design Automation Conference (DAC), 2014, pp.1,6, 1-5 June 2014.

[13] P. Burgio, A. Marongiu, R. Danilo, P. Coussy, L. Benini, "Architecture and programming model support for efficient heterogeneous computing on tightly-coupled shared-memory clusters," Design and Architectures for Signal and Image Processing, 2013, pp.22,29, 8-10 Oct. 2013.

[14] T.R. Da Rosa, R. Lemaire, F. Clermidy, "A Co-design Approach for Hardware Optimizations in Multicore Architectures Using MCAPI," 9th International Workshop on Interconnection Network Architectures: On-Chip, Multi-Chip (INA-OCMC), 2015, pp.17-20, 19-19 Jan. 2015.

[15] S. Miura, T. Hanawa, T. Boku, M. Sato, "XMCAPI: Inter-core Communication Interface on Multi-chip Embedded Systems," Conference on Embedded and Ubiquitous Computing, 2011 pp.397,402, 24-26 Oct. 2011.

[16] Multicore Association. Multicore Communications API specification version 2.015, March 2011, [online], Available: http://www.multicore-association.org/workgroup/mcapi.php.

[17] C. Ebert and C. Jones, "Embedded software: Facts, figures, and future," Computer, vol. 42, no. 4, pp. 42–52, Apr. 2009.

[18] M.R. Guthaus et al., "MiBench: A free, commercially representative embedded benchmark suite," in IEEE International Workshop on Workload Characterization, 2001, pp.3-14, Dec. 2001.