Multicore Communications API (MCAPI) implementation on an FPGA multiprocessor

Lauri Matilainen, Erno Salminen, Timo D. Hämäläinen, and Marko Hännikäinen
Tampere University of Technology,
Department of Computer Systems,
P.O. Box 553, 33101 Tampere, Finland
Email: lauri.matilainen@tut.fi, erno.salminen@tut.fi, timo.d.hamalainen@tut.fi, marko.hannikainen@tut.fi

Abstract—**The design and implementation of an application programming interface (API) is a trade-off between abstraction it provides and overheads it causes. This paper presents an implementation of Multicore Communications API (MCAPI) on a heterogeneous platform consisting of FPGA-based multiprocessor system-on-chip (MPSoC) connected via PCIe to an external CPU board. The purpose is to provide a unified programming API to different processor and OS types as well as hardware IP-blocks. MCAPI is shown to meet these requirements. We show the MCAPI transport implementation on three processors and two buses, measure the overhead cost, and analyze the effort required to port an application from a PC to the MPSoC. The measured library memory footprint is less than 25KB and the roundtrip communication latency is diminishing low - only few dozen clock cycles - compared to non-MCAPI implementation.**

Keywords-parallel programming, multiprocessor, programming interface, FPGA

I. INTRODUCTION

Modern embedded applications are becoming increasingly complex and more demanding with respect to code reuse as platforms develop and applications should be implemented rapidly. The implementation is combination of software and hardware and may even change several times during product's lifetime. Hence, efficient hardware alone is not enough but needs the support of programming models and tools.

Figure 1 depicts the goal of this paper. The overall functionality is specified using, for example model based techniques, and mapped to physical implementation consisting of various processors and fixed-function HW IP-blocks. In addition, efficient software platform services like operating systems, drivers and compatibility layers are needed.

The goal is to implement an interface layer that provides suitable abstraction for both SW- and HW-blocks and still offers feasible performance, small memory and area requirements, as well as fast code porting and re-mapping of application tasks.

There are already several programming models and tools for MPSoC programming [1] [2]. These tools help, for example, in partitioning and mapping the problem to the specific platform. Application portability is often provided by some compatibility layer API (Application Programming Interface). However, traditional multicomputer APIs are too heavy-weight and lack support for extreme heterogeneity like treating HW IP-blocks as "processors".

Multicore Communications API (MCAPI) [3] was developed for multi-core environment where inter-core communication is expected to be faster and more reliable than inter-computer. Thus, MCAPI aims to be more lightweight and efficient than other parallel computing APIs.

We adopted MCAPI for an FPGA-based MPSoC environment. Modern FPGA devices very flexible and can include up to tens of soft-core processors and IP-blocks. Main goal is a practical solution that is immediately usable in embedded product development. The main contributions of this paper are:

1. The first MCAPI implementation for FPGA MPSoC.
2. Analysis of memory footprint and performance overhead.
3. An example demonstrating code portability effort.

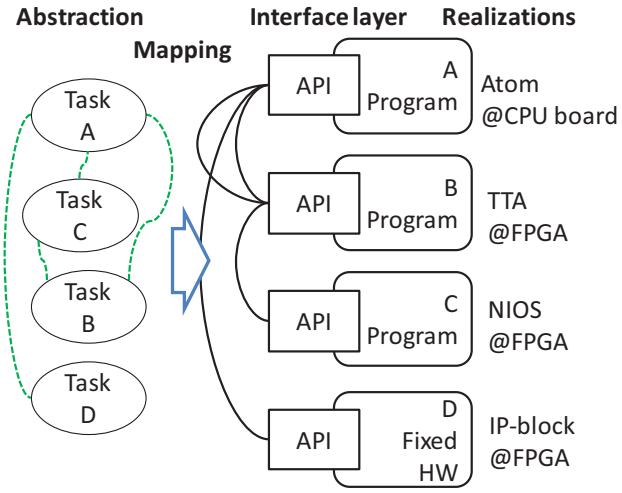

Figure 1 Application tasks are implemented in processor programs and fixed-function HW blocks using unified interface layer abstraction. Intel Atom, Customizable TTA-processors and Altera NIOS are example processors used in this work.

Rest of the paper is organized as follows: In section II we discuss related work. Section III presents an overview of MCAPI specification and Section IV illustrates our own implementation. Section V presents the obtained results and Section V the conclusions.

II. Related Work

An API is an abstraction that describes an *interface* for the interaction between system components. In practice, an API is a set of functions in a program, e.g. *send_data()* or *connect_channel()*. Such common interface is needed in order to develop efficiently complex portable applications. The same functions should be available to programmer regardless of API implementation that can be e.g. PC running Linux and NIOS without any OS. Consequently, most of the development and functional verification can be carried out on a workstation, and porting the tested software into target MPSoC should go smoothly. However, some APIs make certain assumption from the HW platform, such as utilization of shared memory.

A. OpenMP, MPI, and CORBA

OpenMP is an API for shared-memory multiprocessing programming in C/C++ and Fortran on many architectures ranging from desktops to supercomputers, e.g. Unix and Windows NT platforms [4]. OpenMP requires special support from the compiler which, on the other hand, makes it quite easy to adopt. The code can still be compiled for serial execution and parallelization can be added gradually.

MPI is a message-passing library interface specification [5]. Unlike OpenMP, MPI can be used on either shared or distributed memory architectures. However, MPI requires more changes to source codes than OpenMP but does not require any special compilers.

CORBA (Common Object Request Broker Architecture) is meant for inter-object communication between computers. It provides high abstraction and several services. Attempts have been made to use CORBA on embedded systems and even some parts have been implemented in VHDL for hardware [6]. CORBA is, however, too heavy-weight to our goals.

Unfortunately generic solutions originating from workstation environment may easily consume too many resources from an embedded system. Moreover, the sheer number of functions (e.g. ~300 in MPI) might prove overkill as all of them might not be needed.

Another challenge is memory footprint. For example, Cell Broadband Engine includes 8 slave cores (synergistic processing engine, SPE) each having 256KB of local memory. This is "far too small to host a full MPI library" [8], but some reduced MPI versions are available [7] [10].

B. Research proposals and experiments

Among several multiprocessing interface proposals we consider some cases close to our goals. Cell processor is one interesting platform since its satellite processors offer good performance but optimized software is often laborious. To develop. Khunjush [13], Abellan [10], Hung [8] have used and analyzed Cell SDK which offers low-level primitives to access parallel resources. Hung has also implemented a hybrid API combining parts of MPI and MCAPI and their work is closest to our MCAPI implementations.

Several MPI implementations have been presented for FPGA-based MPSoCs as well. Minhass [7] uses Altera NIOS as we do, whereas Saldaña [15] and Mahr [14] Xilinx MicroBlaze. Setälä et al. have presented distribution embedded applications modeled in UML 2.0 and the associated inter-process communication [17].

Paulin et al. [11] [16] have presented a custom MPSoC exploration environment called StepNP. It offers hardware support message-passing and multithreading to minimize latencies. We compare our results with the related work in later sections.

III. Multicore Communications API (MCAPI)

The MCAPI [3] is targeted primarily towards *inter-core* communication whereas MPI and sockets are mainly developed for *inter-computer* communication. Thus, a principal design goal of MCAPI was to specify a low latency API to enable efficient use network-on-chip. MCAPI's communications latencies and memory footprint are expected to be significantly lower than MPI or sockets, at the expense of less flexibility.

MCAPI's objective is to provide a limited number of calls with sufficient communication functionality while keeping it simple enough. Additional functionality can be layered on top of the API set. The calls in the specification serve as examples of functionality and are not mapped to any particular existing implementation [3].

A. Communicating entities

The MCAPI specification is both an API and communications semantic specification. It does not define which link management, device model or wire protocol is used underneath it. As such, by defining a standard API, it is intended to provide source code compatibility for application code to be ported from one environment to another (from PC to MPSoC, in our case). The implementation of MCAPI hides also the differences in memory architectures.

MCAPI communication is based on *node* and *endpoint* abstraction, as shown in Figure 2. An MCAPI node is a logical concept that can be denote to many entities, including a process, a thread, a hardware accelerator, or a processor core. Nodes are always unique and, in our case, statically defined at design time.

Each node can have multiple endpoints that are socket-like termination points. For example, an encryption node could be implemented as single thread receiving plain text from one endpoint, and sending encrypted data via another endpoint. Endpoints are defined with a tuple <*node_id, endpoint_id*>. Endpoints can be created at runtime.

MCAPI *channels* can be dynamically created between pairs of endpoints. However, channel type and direction cannot be changed without deleting and recreating the channel. Multicast and broadcast is not supported. Channels are normally set up once during initialization. MCAPI runtime library performs operations like name lookup, route determination, and buffer allocation between a specific pair of endpoints. After that the channel sends and receives are carried out with less overhead.

We use static endpoint names and channel definitions to explicitly embed the topology in the source code. This also enables the use of external tools to generate topologies automatically and facilitates simple initialization.

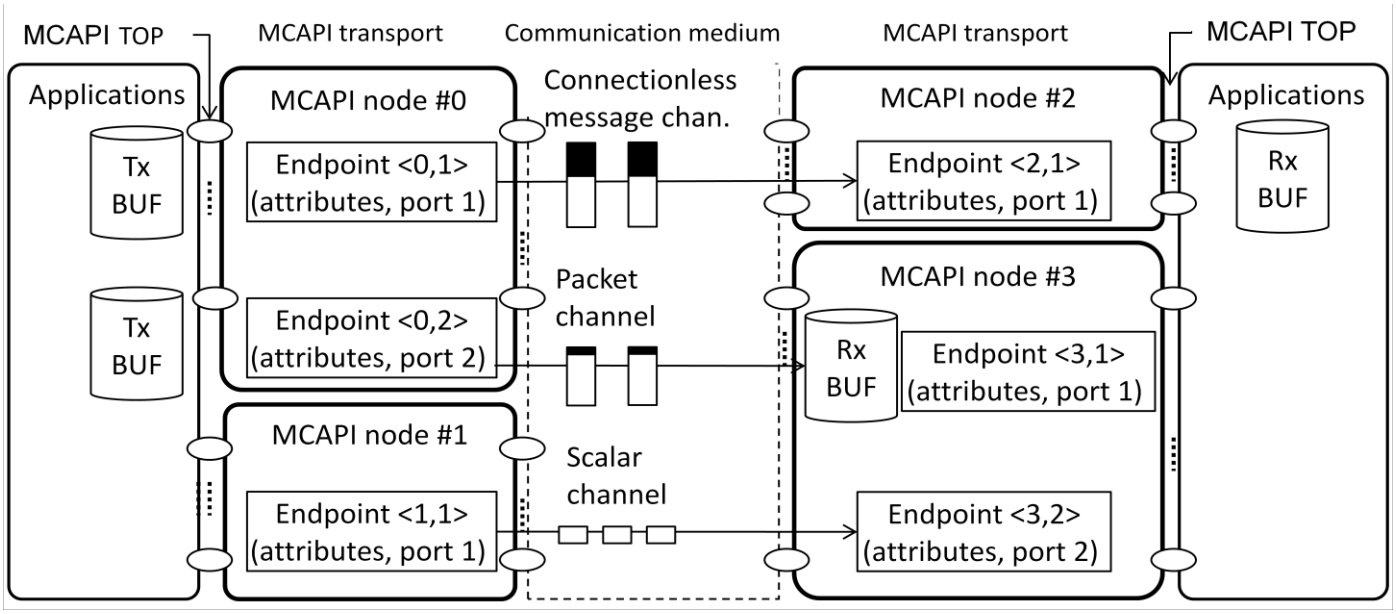

Figure 2. Example with 4 MCAPI nodes that are using the three available communication schemes. Communicating endpoints must have the same type, hence there are 6 of them.

B. *Communication types*

MCAPI defines three communications types:

1. Messages – connection-less datagrams.
2. Packet channels – connection-oriented, uni-directional, FIFO packet streams.
3. Scalar channels – connection-oriented single word uni-directional, FIFO packet streams.

MCAPI messages transmit data between endpoints without first establishing a connection. The memory buffers on both sender and receiver sides must be provided by the user application. MCAPI messages may be sent with different priorities. The topmost communication arc in Figure 2 depicts a message transfer.

MCAPI packet channels provide a method to transmit data between endpoints by first establishing connection, thus potentially removing the message header and route discovery overhead (marked with darker color in Figure 2). Packet channels are unidirectional and deliver data in a first in first out (FIFO) manner. The buffers are provided by the MCAPI implementation on the receive side, and by the user application on the send side, as depicted in the middle.

MCAPI scalar channels transmit 8-bit, 16-bit, 32-bit and 64-bit words between endpoints by first establishing a connection. Like packet channels, scalar channels are unidirectional and deliver data in a FIFO manner.

Each of these communications types has its own API calls, the total number being ca. 50. About one fourth are for initialization and setup, whereas the rest are for communication. Channel API calls are designed to be simple and statically typed, which minimizes overhead of dynamic software structures. This allows applications to access the underlying multicore hardware with low, preferably deterministic latency.

Messages are the most flexible form, and are useful when senders and receivers are dynamically changing and communicate infrequently. These are commonly used for synchronization and initialization.

Packet and scalar channels provide light-weight socket-like stream communication mechanisms for senders and receivers with static communication graphs. In a multicore, these channel APIs provide a low-overhead ASIC-like uni-directional FIFO communications capability.

IV. HARDWARE PLATFORM IN CASE STUDY

Figure 3 shows the MPSoC platform used in this study. Platform includes two processing elements (PEs), HIBI network, as well as direct memory access (DMA) interfaces for IP-blocks (HIBI PE DMA) and external memory (HIBI MEM DMA). Processors have local, private instruction and data memories, and inter-processor communication is implemented using message-passing. The architecture is in a FPGA on a development board that is connected via PCIe to a PC, or embedded CPU board. The platform was synthesized to Altera's Arria II GX FPGA development board [12] at 100MHz.

A. *Bus interfaces*

Blocks are connected to the bus by DMAs that take care of copying data to/from network and local dual port memory in order to speed up the communication and overlap it with computation. The difference is in their usage: HIBI PE DMAs are configured by the local PE (Nios) whereas HIBI MEM

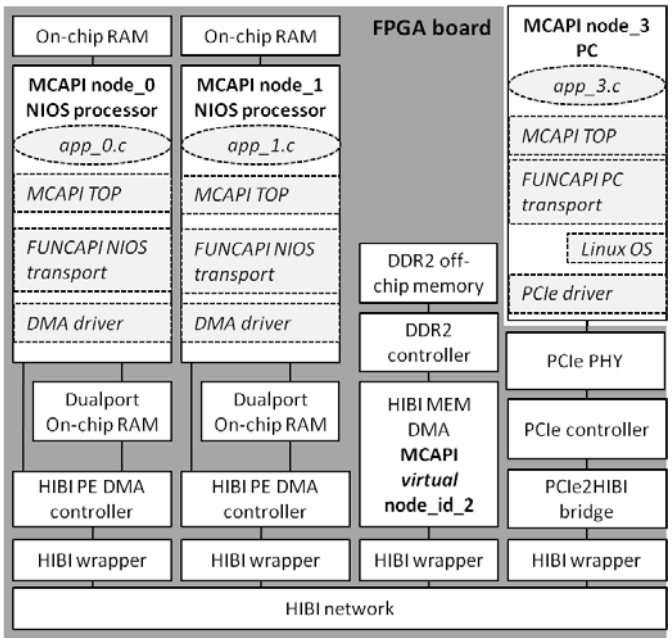

Figure 3 Case study architecture with PC and MP-SoC on FPGA.

DMA is configured by the remote host (Nios or PC) from the HIBI side. Configuration needs 4 parameters: *source address, target address, amount, and command*. Processors' DMA can have one ongoing TX to the network but can be waiting for multiple RX data streams from the network. PE is either interrupted upon the completion of reception, or it can actively poll the status of RX channels. Transfers to/from the local on-chip dualport data memory are controlled explicitly by the programmer. Here, these *scratch-pad memories* run with the same frequency as the processors. Similarly to SPE memories in Cell processor i.e. PEs cannot access each other's local memory space. Instead, each HIBI wrapper has a set of target addresses and PE DMA performs address translation between local memory and HIBI addresses.

Memory DMA offers multiple transfer channels to both directions. Memory can be accessed transparently but DMA controller allows using single bus address in the transfers. For example, reading a large block of data from memory using DMA is clearly more efficient than using one request per data word. This is beneficial in HIBI that utilizes multiplexed address and data lines, and also in packet-switched NoCs.

B. PC

A regular PC (2.4 GHz, 1024 MB) running Debian GNU /Linux 5.0.6, is used for application development and FPGA synthesis. The main purpose is here to demonstrate that an application code can easily run either on PC or Nios since MCAPI takes care of the inter-processor communication be it PCIe, HIBI, or any other. Hence, the applications can be developed in a workstation environment that allows simpler debugging. Then, application code can be transferred to FPGA or be distributed among PC and FPGA processors.

V. MCAPI IMPLEMENTATION

Figure 3 shows also the software layers implementing MCAPI on our platform. Nearest to hardware is a DMA driver that was previously presented in [17] and was used with eCos operating system. In this case study we have excluded the operating system. We name our MCAPI implementation as FUNCAPI according to a research project "Funbase" (Function-based platform).

A. Layered approach

The implementation of FUNCAPI is divided into two layers: *top* and *transport*. The top layer implements MCAPI specified endpoint abstraction for user application and does only simple error checking for function calls.

Transport layer implements interface between the top layer and the underlying HAL (Hardware Abstraction Layer) layer, namely the DMA drivers in this case. HAL layer abstracts the underlying communication media and protocol. In this case study, we use HIBI network inside FPGA and PCIe between FPGA and PC. Other responsibilities are to handle MCAPI databases and send/receive data and messages to/from other physical PEs. It transforms the used endpoint identifiers to target PE's HIBI addresses. Transport layer also performs error checking features concerning illegal endpoint accesses or splitting too long transmissions. Application cannot access non-existing nodes or endpoints which gives safety for the application developer point of view.

B. Using virtual nodes for fixed-function units

A new feature in our implementation is that also HW accelerators are seen as MCAPI nodes. Hence they can be accessed with regular MCAPI functions, although IP-blocks itself does not support MCAPI natively at all. This is done by *virtualizing* the IP-block's endpoint abstraction to transport layer in a processor. Hence, MCAPI acts also as a HW/SW interface and a node is not aware with which type of PE it is communicating. A major benefit is that this allows gradually accelerating the application by including more and more hardwired accelerators.

A simple pseudo code example below shows how the virtual node for a discrete cosine transform (DCT) HW IP-block is handled. The IP-block is given a unique node number at design time, "1" in this case. By default the MCAPI *msg_send* function calls the driver of DMA controller (a C-macro *HPD_send*), but for this virtual node number 1 it forwards execution to a DCT driver. This eventually calls HPD_SEND function, perhaps multiple times, to configure the DCT and send raw data to be transformed. The same procedure is easily repeated for other accelerators.

```
send_msg(…) {
...
        switch(node_id){
        case '1' :
                ...
                dct_drv(…);
                ...
        default :
                ...
                HPD_SEND(…);
                ...
        }
}
```

C. Functions

The connectionless message and packet channel API functions have both *blocking* and *non-blocking* variants. For example *mcapi_msg_recv(...)* function blocks the execution of the application node until all the message data has been received. The non-blocking variants will return immediately and therefore their names are denoted with "*_i*".

The 11 most essential MCAPI functions for this study are listed and described in Table 1. They are divided into 3 categories as follows:

1. Functions 1-5 are initialize the system and called only once at the beginning of application.
2. Functions 6-9 are used to pass and receive data between MCAPI nodes in different MCAPI communication formats.
3. With functions 10-11 user can query the status of the non-blocking operation. More specific descriptions about the MCAPI functions can be found from the MCAPI specification.

Required buffers are statically reserved at compile time just like endpoints and channels to increase overall reliability and ease of debugging. Applications must take care of data alignment, endian-ness and the internal structure of messages in general.

D. Example

An example execution sequence is described in Figure 4 for initialization and transfer of a single message. The 4 leftmost lifelines depict the software, namely application, the two layers of MCAPI, and the DMA driver. On the right, there is the HW network. The receiving PE is not shown for brevity.

Before the actual message passing can be done, node and outgoing endpoint initialization is needed. In the first phase, sender initializes itself with a unique tuple *<node_id, port_id>* and creates an endpoint for sending messages. Similarly, receiving side must execute the same initializations.

In the 5th phase, user application obtains a handle for the target endpoint from the (static) MCAPI database and uses that for sending messages. This phase could omitted in a completely static system. However, it ensures the validity of target endpoint and retains code portability to dynamically configured systems.

Phases 1-6 are executed only once in applications boot sequence. Note that no data is sent over the network during the initialization, which simplifies API a lot in the absence of shared memory. It should be noted that this is possible only when the node topology is static. In general, nodes can also be created dynamically as threads, and then initialization is done between a thread and MCAPI implementation.

After initialization, the message data is ready to send to network using HIBI_PE_DMA driver. The FUNCAPI top layer forwards message to transport layer which implements MCAPI endpoint abstraction transformation to HIBI address space. For example, sending to endpoint *<node=3, port=1>* is transformed to target address *0xF000_0000*. Once the HIBI PE DMA has been configured, MCAPI returns.

It is obvious that the actual communication functions are more important regarding the performance than the initialization. Therefore the implementation is focused more to transfer functions.

Table 1. The most essential MCAPI functions and their meaning.

Function	Meaning
initialize(…)	Initializes the MCAPI implementation
create_endpoint(…)	Creates an endpoint into local database
connect_pktchan_i(…)	Connects send & receive side endpoints with a channel (non-blocking function)
open_pktchan_recv_i(…)	Creates a typed, local representation of the reception channel. It also provides synchronization for channel creation between two endpoints (non-blocking)
open_pktchan_send_i(…)	Same as previous function for sender's endpoint
msg_send(…)	Sends a (connectionless) message
msg_recv(…)	Receives a (connectionless) message. Blocks until the whole message has arrived.
pktchan_send(…)	Sends a (connected) packet on a channel
pktchan_recv(…)	Receives a data packet on a (connected) channel
test(…)	Tests if non-blocking operation has completed
wait(…)	Waits for a non-blocking operation to complete

VI. Implementation

We implemented nearly all MCAPI functions but excluded certain tests for non-blocking transfers. As mentioned, the configuration of nodes, endpoints, and channels is static. Total lines of current C code in FUNCAPI transport for Nios is about 1.500 and low-level DMA driver 266 without comment lines. MCAPI transport for PC takes 1.450 lines of C and PCIe driver 248 lines. These give a rough estimate of the modest complexity and ease of portability of the MCAPI implementation itself.

It must be noted, that although transport layer is specific to each platform, large portions of it can be reused. For example, in our case 95% of the source code remains the same despite the differences between Nios without OS and PC with Linux. The latter communicates using PCIe that is accessed with Linux driver functions that are implemented as regular user-space applications in this study. Table 2 shows main software components for Nios implementation; a similar set is used in PC as well.

For remapping a MCAPI node from one processor to another we show a simple case with the test application in Figure 2. At first, the left part (nodes 0+1) was implemented on Nios processors Figure 2. On the second step, we remapped node 1 from Nios to PC. Example code snippet from *mcapi_mapping.h* below shows the address definition that is the only modification needed in addition to re-compilation to target processor.

```
// address_map[Nios_0_addr, Nios_1_addr];
address_map[Nios_0_addr, pc_addr];
```

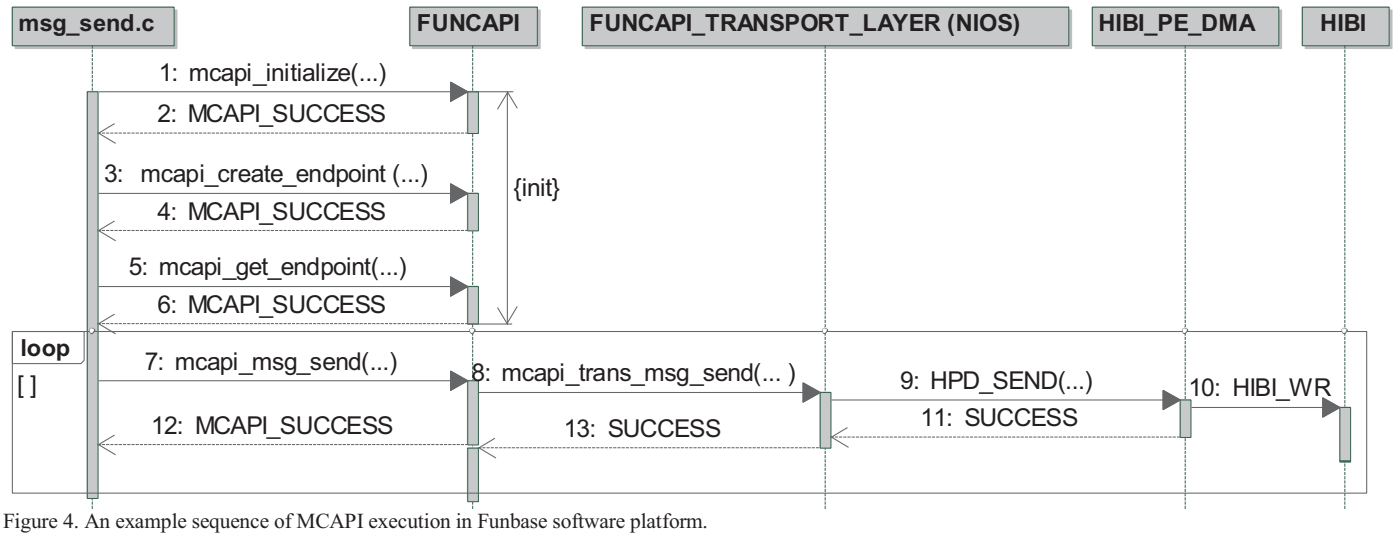

Figure 4. An example sequence of MCAPI execution in Funbase software platform.

Hence, the remapping node from other processor to another requires changing just one line in this example. Without standardized API, the developer has to study the underlying communication media driver usage and rewrite the application code. Consequently, MCAPI offers even hours save in the code reuse with different processors.

Table 2 Nios-related software components.

Code	Content
app_x.c,	Test application code
mcapi.c	MCAPI top layer implementation
funcapi_transport_Nios.c	MCAPI transport layer implementation
mcapi_config.h	Defines constants, e.g. MCAPI packet sizes
mcapi_mapping.h	Defines endpoints and mapping to physical addresses
mcapi_datatypes.h	Defines MCAPI data types
hibi_pe_dma_driver.c	Hardware abstraction for DMA controller
hpd_regs_and_macros.h	Low-level register and macro definitions

VII. PERFORMANCE MEASUREMENTS

We measured the memory footprint, communication latencies, and throughputs. Latency statistics are listed in Table 3 and Table 4. The latency measurements were repeated 100 times for each transfer size to obtain reliable results. Measurements are divided into two different parts: initialization and data transfer.

Latencies are measured by using different type of MCAPI communication methods between two NIOS processor: *node_0* and *node_1*. In transfer measurements, we assume that the *node_1* is ready to receive data from sending PE *node_0* and vice versa. Thereby, we are able to distinguish the different parts of the transfer and see the difference of MCAPI communication methods.

The first row (connection-less message, 21 clock cycles) in the Table 3 represents used time in initialization phases 1-6 in Figure 4. Packet and scalar channels are connection-oriented channels and their initialization takes more time than message channel. However, initializations are done only once in the beginning of the application execution and it is independent of the message or packet size. Thus, the latency caused by initialization functions is negligible with long term stream-like data transfers.

Actual data transfer translates into DMA driver function call *HPD_SEND()*, phase 9 in Figure 4. It initializes the DMA transfer with data size, data address, receive address and transfer size parameters. After this, HIBI_PE_DMA handles autonomously the rest of the transfer. Correspondingly, in the receive node, HIBI_PE_DMA is initialized to copy the received message to a desired address in local memory. Function *msg_recv()* just polls the status of the DMA driver whether the data is ready or not.

Packets size affects obtained throughput notably. For example, with 16 B packets Nios to Nios throughput is about 25 MB/s, whereas 1KB packets yield 120 MB/s. Without MCAPI throughputs are 8 % and 2% higher, respectively. Theoretical maximum is 4B × 100 MHz = 400 MB/s. Throughput between PC and Nios processor via PCIe is significantly lower, only 15 MB/s but this is caused by the general purpose, simple PCIe Linux driver that handles only single 32/64-bit transfers. Its optimization is not on the scope of this paper.

Table 3. Initialization latencies for message, packet and scalar channels.

Communication type	Initialization time [clock cycles]
Message	21
Packet	90
Scalar	90

Table 4. Data transfer roundtrip latencies for various packet/message size.

Words(32-Bit)	Data transfer [clock cycles]
1	63
2	63
4	67
8	71
16	79
32	105
64	127
128	191

A. Memory footprint

Program memory footprint of the FUNCAPI code library is about 25KB, from which 7KB is for top and 18 KB for transport layer. HIBI_PE_DMA driver library consumes only 1KB slice from the system memory.

The total data memory footprint depends on the used maximum packet/message size. With 2KB packet/message size, the memory buffer usage is *channel_count* × *max_packet_size*. The channel count differs between nodes. For example, system in Figure 2 has one channel of each type: message, packet and scalar.

Thus, the complete buffer memory usage is sum of message, packet and scalar buffers: 1 × 2KB + 1 × 2KB + 4B ≈ 4 KB. Program and data memory in total takes about 25+1+4= 30 KB.

B. Comparison to other API implementations

Table 5 shows results from literature. Columns are divided into different features: Used API; HW platform; memory footprint; inclusion of OS; system frequency; one-way communication latency; used transfer type. Many sources report latencies in microseconds and we converted them to clock cycles to ease comparison. Similarly, we converted the reported roundtrip times to one-way latencies. The roundtrip latency in our MCAPI implementation between two PE is from 63 to 191 cycles depending on the size of sent data, hence one-way minimum latency is about 32 cycles.

Three sources report results from MPI implementations. Saldana et al. report 340 clock cycles communication latency for zero-length message transfer. Minhass et al. measured clearly larger, ~1150 clock cycles latency for minimum size data transfer and they considered that as "bottleneck of the system" [7]. Implementation by Mahr, on the other hand, had even larger latency, ~9 000 clock cycles for 128B packet [14].

Overhead of MPI and OpenMP was recognized also by Hung et al. They presented MSG API which obtained at minimum ~1100 clock cycles latency for minimum size data transfer [8]]. Hence, MSG is faster than native CELL SDK communication functions of the platform but practically doubles the communication latency compared to memory mapped I/O (mmio) [8]. StepNP can perform message-passing in less than 40 cycles. Fast communication is absolutely necessary in network processing where the tasks are short, even less than 500 instructions [16]. In summary, our MCAPI implementation performs very well compared to these implementations.

As seen from the table, memory footprint of the API is rarely reported in the related works. Data and program memory footprint was classified even less frequently. Nevertheless, Saldana et al. presented 8.76 KB communication library size, which is one third of our implementation. However, Saldana et al. does not inform thoroughly which part of the MPI is supported in their function subset. Mahr et al. were able to implement MPI in 16 KB or less, depending on configuration. For comparison, eCos library memory footprint in NIOS processor was about 53 KB in our earlier prototypes. In [16], accelerator units for message-passing and object request broker required 27 kilogates of logic and 12.5 KB memory per system plus 3 kilogates/PE and 0.5 KB/PE.

C. Performance considerations

The factors contributing to API performance originate from the underlying HW and SW layers and also what operations are included in measurements. Buffer management was here done by the application and hence excluded from the measurements. For example, IPC mechanism presented by Setälä et al. had to use memory copy in receive/send side due to limited on-chip memory. This caused heavy overhead for the transfer latency (~5 clock cycles/Byte) in addition to context switch in OS.

Our MCAPI implementation has few restrictions which may cause the performance gap compared to many other API implementations. Firstly, all nodes, endpoints and channels are decided in design time. Secondly, MCAPI offers only 50 communication functions. Compared to MPI's over 200 functions, this is a huge difference [5]. And thirdly, buffer management is mostly done by user application.

We notice from Table 5 that OS causes major overheads; for example in Cell transfers to/from PPE running Linux are much slower than between SPEs that do not have OS. Normally the context switch in an embedded processor takes hundreds of clock cycles, whereas in another extreme StepNP's HW-accelerated context switch takes only one cycle and scheduling only around 40 clock cycles [9].

Other aspects that should be avoided are dynamic memory allocation (175 clock cycles in our platform) and inefficiently planned thread creation (e.g. 1.68 ms which equals over 5 million cycles [10]). Similarly, table look-aside buffer (TLB) is needed in virtual memory system but may incur notable runtime overheads, e.g. in the range 500-1500 cycles for TLB miss and 85K-500K cycles for replacement [13].

VIII. CONCLUSIONS

We have implemented, as far as we know, for the first time Multicore Association MCAPI for an FPGA-based MPSoC including virtual endpoint support for hardware IP-blocks. With layered structure, we separated the platform specific parts in a transport layer, thereby porting MCAPI conveniently to NIOS without OS and PC with Linux. Portability and performance evaluation was demonstrated in the case study. In addition, comparison to other API implementation from the literature was done.

It seems obvious that unified and standardized communication API is needed for embedded solutions, when the application parallelization is increasing all the time.

MCAPI is one solution to this dilemma and seems to be fairly promising one.

Our future work will be to improve the transport layer structure to decrease memory footprint and support to handle multiple low-level communications media drivers for a single MCAPI transport implementation. Furthermore, we will investigate more thoroughly the differences between APIs with respect to their overheads and ease of use.

REFERENCE

[1] W Wolf, A. A. Jerraya, G. Martin, "Multiprocessor System-on-Chip (MPSoC) Technology," Computer-Aided Design of Integrated Circuits and Systems, 2008, vol 27, pp.1701

[2] R. Leupers, L. Thiele, Xiaoning Nie, B. Kienhuis, M. Weiss, T. Isshiki, "Cool MPSoC programming," in Proc. DATE, 2010, pp. 1488

[3] Multicore Association. Multicore Communications API Specification Version 1, 2008. http://www.multicore-association.org.

[4] Woo-Chul Jeun, Soonhoi Ha, "OpenMP Effective OpenMP Implementation and Translation For Multiprocessor System-On-Chip without Using OS," in Proc. ASP-DAC, 2007, pp 44-49

[5] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard Version 2.2, http://www.mpi-forum.org

[6] Objective Interface Systems, Inc., http://www.ois.com/Products/orbexpress-fpga.html

[7] 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," in Proc. FPGAworld, 2009, pp 52-57

[8] Shih-Hao Hung, Wen-Long Yang, Chia-Heng Tu, "Designing and Implementing a Portable, Efficient Inter-core Communication Scheme for Embedded Multicore Platforms," in Proc. RTCSA, 2010, pp. 303-308

[9] J. Holt et al., "Software standards for the multicore era", IEEE Micro May/June 2009, pp. 40-51.

[10] J.L. Abellan, J. Fernandez, M.E Acacio, "CellStats: A Tool to Evaluate the Basic Synchronization and Communication Operations of the Cell BE," in Proc. PDP, pp. 261-268.

[11] P. G. Paulin, C. Pilkington, M. Langevin E. Bensoudane, and G. Nicolescu, Parallel Programming Models for a Multi-Processor SoC Platform Applied to High-Speed Traffic Management, in Proc. CODES+ISSS, 2004, pp. 48-53.

[12] Altera Arria II Device Handbook, Altera Corporation, 2010, http://www.altera.com/literature/hb/arria-ii-gx/arria-ii-gx_handbook.pdf

[13] F. Khunjush, N. J. Dimopoulos," Extended Characterization of DMA Transfers on the Cell BE Processor," in Proc. IPDPS, 2008, pp. 1 – 8.

[14] P. Mahr, C. Lörchner, H. Ishebabi and C. Bobda, "SoC-MPI: A flexible Message Passing Library for Multiprocessor Systems-on-Chips," Reconfigurable Computing and FPGAs, in Proc. ReConFig International Conference, 2008, p. 187 - 192

[15] M. Saldaña, P. Chow, "TMD-MPI: An MPI Implementation for Multiple Processors Across Multiple FPGAs," in Proc. Field Programmable Logic and Applications International Conference, 2006, p. 1 – 6

[16] P. G. Paulin et al., Parallel Programming Models for a Multiprocessor SoC Platform Applied to Networking and Multimedia, TVLSI, Vol 14, Iss. 7, 2008, pp. 667-680.

[17] M. Setälä, P. Kukkala, T. Arpinen, M. Hännikäinen, T. D. Hämäläinen, "Automated Distribution of UML 2.0 Designed Applications to a Configurable Multiprocessor Platform," in Proc. SAMOS VI, 2006, pp. 27-38.

Table 5 Performance and memory comparison.

1st Author	Ref	API	Platform	MEM footprint [code, data buffer] [KB]	OS	f [MHz]	One-way latency [us]	One-way latency [cycles]	Transfer type
Saldana	[15]	MPI	MicroBlaze	8.7KB library	No	40	8.5	*340*	on-chip TMD-MPI zero-length message
Minhass	[7]	MPI	NIOS	10KB / core,Rx 2KB, Tx 1KB	No	50	23	*1 150*	"MPI minimum size packet" to near neighbor in same FPGA
Mahr	[14]	MPI	MicroBlaze	11.5 - 16 KB	No	100	-	9 000	128 Bytes MPI message
Hung	[8]	*mmio()*	CELL BE	total <256KB	Yes (PPE)	3200	0.06	*198*	mailbox PPE->SPE
		Cell SDK			Yes (PPE)		2.96	*9 469*	mailbox PPE->SPE
		MSG			No		~0.35	*~1 100*	16 B SPE -> SPE
		MSG			Yes (PPE)		~1.5	*~4 800*	16 B PPE -> SPE
		MSG			No		~0.4	*~1 300*	2 KB SPE -> SPE
		MSG			Yes (PPE)		~1.8	*~5 800*	2 KB PPE -> SPE
Khunjush	[13]	Cell SDK	CELL BE	N/A	No	3200	0.07	*220*	16 B put/get SPE -> SPE
Abellan	[10]	Cell SDK	CELL BE	N/A	No	3200	0.08	*256*	mailbox SPE->SPE
					Yes (PPE)		0.18	*512*	mailbox PPE->SPE (direct)
					Yes (PPE)		3	*9 600*	mailbox PPE->SPE (sys call)
					No		~0.1-0.15	*~320-500*	16 B SPE DMA (priv. mem)
					No		~0.2-0.25	*~640-800*	2 KB SPE DMA (-"-)
					Yes		1680	*~5.4M*	Thread creation on PPE
Paulin	[16]	DSOC	StepNP	12.5 KB + n * 0.5KB	No	200	<0.2	*<40*	Message passing
Matilainen	**This**	MCAPI	NIOS, PC	24.8KB, 4KB	No	100	-	32	4 Bytes message