QoS-Driven Reconfigurable Parallel Computing for NoC-Based Clustered MPSoCs

Jaume Joven, Akash Bagdia, Federico Angiolini, *Member, IEEE*, Per Strid, David Castells-Rufas, Eduard Fernandez-Alonso, Jordi Carrabina, and Giovanni De Micheli, *Fellow, IEEE*

***Abstract*—Reconfigurable parallel computing is required to provide high-performance embedded computing, hide hardware complexity, boost software development, and manage multiple workloads when multiple applications are running simultaneously on the emerging network-on-chip (NoC)-based multiprocessor systems-on-chip (MPSoCs) platforms. In these type of systems, the overall system performance may be affected due to congestion, and therefore parallel programming stacks must be assisted by quality-of-service (QoS) support to meet application requirements and to deal with application dynamism.**

In this paper, we present a hardware-software QoS-driven reconfigurable parallel computing framework, i.e., the NoC services, the runtime QoS middleware API and our ocMPI library and its tracing support which has been tailored for a distributed-shared memory ARM clustered NoC-based MPSoC platform.

The experimental results show the efficiency of our software stack under a broad range of parallel kernels and benchmarks, in terms of low-latency interprocess communication, good application scalability, and most important, they demonstrate the ability to enable runtime reconfiguration to manage workloads in message-passing parallel applications.

***Index Terms*—Networks-on-chip (NoCs), NoC-based multiprocessor systems-on-chip (MPSoC), parallel computing, quality of service (QoS), runtime reconfiguration.**

I. Introduction

In the past, due to Moore's law the uniprocessor performance was continually improved by fabricating more and more transistors in the same die area. Nowadays, because of the complexity of the actual processors, and to face the increasing power consumption, the trend to integrate more but less complex processors with specialized hardware accelerators [1].

Thus, multiprocessor systems-on-chip (MPSoCs) [2], [3] and cluster-based SoCs with tens of cores such as the Intel SCC [4], Polaris [5], Tilera64 [6] and the recently announced 50-core Knights Corner processor, are emerging as the future generation of embedded computing platforms in order to deliver high-performance at certain power budgets. As a consequence, the importance of interconnects for system performance is growing, and networks-on-chip (NoCs) [7] and multilayer sockets-based fabrics [8], [9] have been integrated using regular or application-specific topologies efficiently in order to be the communication backbone for those systems depending on the application domain.

Nevertheless, when the number of processing elements increase and multiple software stacks are simultaneously running on each core, different application traffic can easily conflict on the interconnection and the memory subsystems. Thus, to mitigate and control the congestion, it is required to support certain level of quality-of-service (QoS) in the interconnection allowing to control and reconfigure at runtime the execution of prioritized or real-time tasks and applications.

From the software viewpoint, to boost software engineer productivity and to enable concurrency and parallel computing, it is necessary to provide parallel programming models and Application Programming Interface (API) libraries which exploit properly all the capabilities of these complex many-core platforms. The most common and viable programming languages and APIs are OpenMP [10] and Message-Passing Interface (MPI) [11] for shared-memory and distributed-memory multiprocessor programming, respectively. In addition, Open Computing Language (OpenCL) and Compute Unified Device Architecture (CUDA) have been proposed to program effortlessly exploiting the parallelism of GPGPU-based platforms [12], [13].

In summary, there is consensus that suitable software stacks and, system-level software in conjunction with QoS services integrated in the hardware platform will be crucial to achieve QoS-driven reconfigurable parallel computing for the upcoming many-core NoC-based platforms.

In this work, reconfiguration is achieved by means of hardware-software components, adjusting a set of NoC-based configurable parameters related to different QoS service levels available in the hardware architecture from the parallel programming model. Regarding the programming model, we believe that a customized MPI-like library can be a suitable candidate to hide hardware many-core complexity and to enable parallel programming on highly parallel and scalable NoC-based clustered MPSoCs 1) due to the inherent distributed nature of message-passing parallel programming model, 2) the low-latency NoC interconnections, 3) because of the easy portability and extensibility to be tailored in NoC-based MPSoC, and 4) since it is a very well-know API and efficient parallel programming model in supercomputers, and therefore,

Manuscript received September 20, 2011; revised April 15, 2012; accepted July 24, 2012. Date of publication October 02, 2012; date of current version August 16, 2013. This work was supported in part by the Catalan Government Grant Agency (Ref. 2009BPA00122), European Research Council (ERC) under Grant 2009-adG-246810, and a HiPEAC 2010 Industrial Ph.D. grant from the R&D Department, ARM Ltd., Cambridge, U.K. Paper no. TII-11-552.

J. Joven and G. De Micheli are with the Integrated Systems Laboratory (LSI), École Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland (e-mail: jaime.jovenmurillo@epfl.ch).

A. Bagdia and P. Strid are with the R&D Department, ARM Limited, Cambridge GB-CB1 9NJ, U.K.

D. Castells-Rufas, E. Fernandez-Alonso, and J. Carrabina are with CAIAC, Universitat Autònoma de Barcelona (UAB), Bellaterra 08193, Spain

F. Angiolini is with in iNoCs SaRL, Lausanne 1007, Switzerland.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TII.2012.2222035

experienced software engineers can create and reuse effortlessly message-passing parallel for the embedded domain, as well as many debugging and tracing tools.

Thus, the main objective is to design a QoS-driven reconfigurable parallel computing framework capable to manage the different workloads on the emerging distributed-shared memory clustered NoC-based MPSoCs. In this work, we present a customized on-chip Message Passing Interface (ocMPI) library, which is designed to support transparently runtime QoS services through a lightweight QoS middleware API enabling runtime adaptivity of the resources on the system. Thus, one major contribution of the proposed approach is the abstraction of the complexity from the provided QoS services in the reconfigurable NoC communication infrastructure. By simple annotations at application-level in the enhanced ocMPI programming model, the end user will reconfigure the NoC interconnect, adapting the execution of parallel application in the system and achieving QoS-driven parallel computing. This is a key challenge in order to achieve predictability and composability at system-level in embedded NoC-based MPSoCs [14], [15].

The ocMPI library has been extended and optimized from previous works [16]. It has been optimized for distributed-shared memory architectures removing useless copies, and most important, it has been instrumented in order to generate open trace format (OTF) compliant traces, which will help to debug and understand the traffic dynamism and the communication patterns, as well as to profile the time that a processor is executing a particular task or group of tasks.

This paper is organized as follows. Section II presents the related works on message-passing APIs for MPSoCs platforms, as well as support for system-level QoS management. Section III describes the designed distributed-shared memory Cortex-M1 clustered NoC-based MPSoC. Section IV presents the QoS hardware support and the middleware SW API to enable runtime QoS-driven adaptivity at system-level. Section V describes our proprietary ocMPI library tailored for our distributed-shared memory MPSoC platform. Section VII reports results of low-level benchmarks, message-passing parallel applications in the distributed-shared memory architecture. Section VIII presents the results about QoS-driven parallel computing benchmarks performed in our MPSoC platform. Section IX concludes the paper.

II. Related Work

QoS has been proposed in [14], [17], and [18] in order to combine best-effort (BE) and guaranteed throughput (GT) streams with time division multiple access (TDMA), to distinguish between traffic classes [19], [20], to map multiple use-cases in worst-case scenarios [21], [22], and to improve the access to shared resources [23], such as external memories [24], [25] in order to fulfill latency and bandwidth bounds.

On the other hand, the industry as well in the academy due to the necessity to enable parallel computing on many-core embedded systems, they provide custom OpenMP [26]–[30] and MPI-like libraries. In this work, we will focus on message-passing. In the industry, the main example of message-passing is the release of Intel RCCE library [31], [32] which provides message-passing on top of the SCC [6]. IBM also explored the possibility to integrate MPI on the Cell processor [33]. In the academy, a wide number of MPI libraries have been reported so far, such as rMPI [34], TDM-MPI [35], SoC-MPI [36], RAMPSoC-MPI [37] which is more focused on adaptive systems, and the work presented in [38] about MPI task migration.

Most of these libraries are lightweight running explicitly without any OS ("bare metal" mode) and they support a small subset of MPI functions. Unfortunately, some of them do not follow the MPI-2 standard, and none include runtime QoS support on top of the parallel programming model, which enable reconfigurable parallel computing in many-core systems.

This work is inspired on the idea proposed in [39], [40] in the ambit of high performance computing (HPC). However, in this work rather than focus on traditional supercomputing systems, we target the emerging embedded many-core MPSoC architectures.

Through our research, rather than focus exclusively on developing QoS services, the main target is to do step forward by means of a hardware-software codesign towards a QoS-driven reconfigurable message-passing parallel programming model. The aim is to design the runtime QoS services on the hardware platform, and expose them efficiently in the proposed ocMPI library through a set of QoS middleware API.

To the best of our knowledge, the approach detailed in this paper represents one of the first attempt together with our previous work [16] to have QoS management on our standard message-passing parallel programming for embedded systems. Rather than in our previous work, where the designed NoC-based MPSoC was a pure distributed-memory platform, this time the proposed ocMPI library have been redesigned, optimized and tailored to suit in the designed distribute-shared memory system.

The outcome of this research enables runtime QoS management of parallel programs at system-level, in order to keep cores busy, manage or speedup critical tasks, and in general, to deal with multiple traffic applications. Furthermore, on top of this, the ocMPI library have been extended in order to generate traces and dump through joint test action group (JTAG) to enable later a static performance analysis. This feature was not present in our previous work, and it is very useful to discover performance inefficiencies and optimize them, but also to debug and detect communication patterns in the platform.

III. Overview of the Proposed Clustered NoC-Based MPSoC Platform

The proposed many-core cluster-on-chip prototype consists of a template architecture of eight-core Cortex-M1s interconnected symmetrically by a pair of NoC switches including four Cortex-M1 processors attached on each side.

Each Cortex-M1 soft-core processor in the subcluster rather than including I/D caches, it includes a 32-kB instruction/data tightly coupled memory (ITCM/DTCM), 2×32-kB shared scratchpad memories, as well as the external interface for a 8-MB shared zero bus turnaround RAM (ZBTRAM) memory interconnected by a NoC backbone. Both scratchpads (also called in this work as message passing memory) are strictly local to each subcluster.

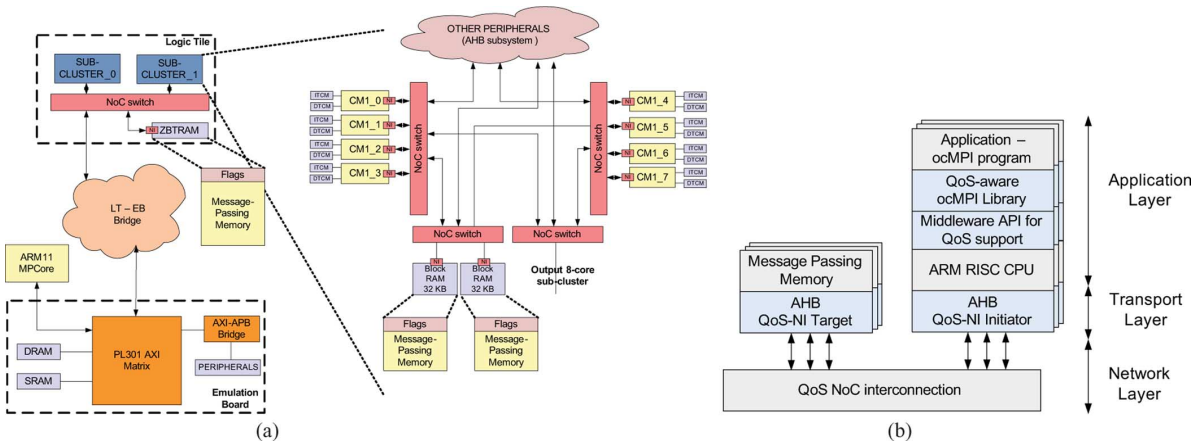

Fig. 1. Architectural and hardware-software overview of our cluster-based MPSoC architecture. (a) 16-core cluster-based Cortex-M1 MPSoC architecture supervised by ARM11MPCore host processor. (b) HW-SW view of the cluster-on-chip platform.

Additionally, each 8-core subcluster has a set of local ARM IP peripherals, such as the Interprocess Communication Module (IPCM), a Direct Memory Access (DMA), and the *Trickbox*, which enable interrupt-based communication to reset, hold and release the execution of applications in each individual Cortex-M1.

The memory map of each 8-core subsystem is the same with some offsets according to the cluster id, which helps to boost software development by executing multiple equal concurrent software stacks (e.g., the ocMPI library and the runtime QoS support), when multiple instances of the 8-core subcluster architecture are integrated in the platform.

For our experiments, as is shown in Fig. 1(a), we designed a 16-core NoC-based MPSoC including two 8-core subcluster instances supervised by an ARM11MPCore host processor. The system has been prototyped and synthesized in a LT-XC5VLX330 FPGA LogicTile (TL), and later, it has been plug-in together with the CT11MPCore CoreTile on the emulation baseboard (EB) from ARM Versatile Products [41] to focus on further software exploration.

As presented in [42], the system can optionally integrate an AHB-based decoupled *Floating Point Unit* (FPU) to support hardware-assisted floating point operations. In our case, the FPU must be connected through an AMBA AHB *Network Interface* (NI) instead of being connected directly to an AHB matrix.

The proposed 16-core clustered NoC-based MPSoC platform enable parallel computing at two levels, 1) intracluster and, 2) intercluster, leverage to exploit locality on message-passing applications. In this scheme, we assume that short-fast intracluster messages will be exchanged using the small size scratchpad memories taking profit of their low-latency access time. On the other hand, for intercluster communication larger messages can be exchanged between each subcluster due to the higher capacity of the ZBTRAM.[1]

This clustered NoC-based architecture instead of including like in a pure distributed-memory architecture, one scratchpad for each processor, each scratchpad is shared between all 4 cores in each side of each subcluster. Thus, this cluster-based architecture can be considered as noncache-coherent distributed-shared memory MPSoC.

To keep the execution model simple, each Cortex-M1 runs a single process at the same time that is a software image with the compiled message-passing parallel program and the ocMPI library. This software image is the same for each Cortex-M1 processor, and it is scattered and loaded in each ITCM/DTCM from one of the ARM11MPCore host processors.

Once the software stack is loaded, the ARM11MPCore through the *Trickbox* starts the execution of all the cores involved in the parallel program. The application will finish only after each Cortex-M1 has completed.

IV. Runtime QoS Support at System Level

As we state before, the QoS services on the platform must be raised up to the system-level to enable runtime traffic reconfiguration on the platform from the parallel application. As a consequence, two architectural changes at hardware level have been done on the NoC fabric. The first one is the extension of the best-effort allocator (either the fixed-priority or round-robin) on the *switch* IP from $\times$pipes library [43], [44] in order to support the following QoS services.

- Soft-QoS—Up to eight levels of priority traffic classes.
- Hard-QoS or GT—Support for end-to-end establishment/release of circuits.

The second structural modification is to tightly-coupled a set of configurable memory-mapped registers in the AMBA AHB NI to trigger the QoS services at transport level.

In Fig. 2, we show the area overhead and frequency degradation to include QoS support. At switch level, we varied the number of priority levels according to each type of best-effort allocator (fixed priority and round-robin).

As expected, the synthesis results[2] show that when eight priority levels are used either with fixed priorities or round-robin best-effort allocator, the increment in area is

[1]Nevertheless, if it is required, even for intracluster communication large messages can be exchanged using a simple fragmentation protocol implemented on top of the synchronous rendezvous protocol.

[2]The results have been extracted using Synplify Premier 9.4 to synthesize each component on different FPGAs. VirtexII (xc2v1000bg575-6) and Virtex4 (xc4vfx140ff1517-11) from Xilinx, and StratixII (EP2S180F1020C4) and StratixIII (EP3SE110F1152C2) from Altera.

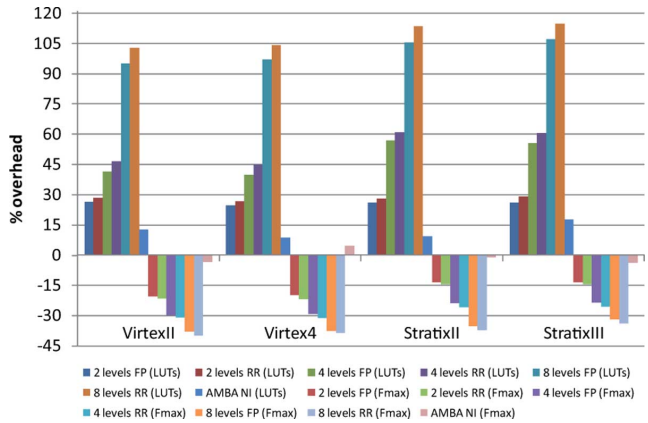

Fig. 2. QoS impact at switch level according to the priority levels, and in the AMBA AHB NI (LUTs, $f_{\max}$).

around 100%–110%, i.e., doubling the area of the switch without QoS. On the other hand, with two or four priority levels, the overhead ranges from 23% to 45% in Virtex, and 25% to 61% in Stratix FPGAs, respectively. The presented priority-based scheme is based on a single unified input/output queue, and therefore no extra buffering is required in the switch. The presented area overhead is the control logic in the allocator, with respect to the base case switch without QoS.

On the other hand, in terms of $f_{\max}$, as shown in Fig. 2, the circuit frequency drops between 32% and 39% in case to use eight priority levels. In the other extreme, if we use just two priority levels, the overhead is only between 13% and 19%, whereas an intermediate solution with four priority levels, the outcome frequency degradation ranges from 23% to 29% depending on the FPGA device and the selected best-effort allocator.

It is important to remark that the hardware required in each switch to establish end-to-end circuits or GT channels can be considered negligible because it is only required a flip-flop to hold/release the grant in each switch.

At the AMBA AHB NI level, as shown in the same Fig. 2, the overhead to include QoS extensions is only 10–15% depending on the FPGA device. Mainly, the overhead is due to the fact to extend the packet format and the redesign of the NI finite state machines. On the other hand, the frequency drop can be considered totally negligible (around 2% drop), and even in one case despited the fact that, the AMBA AHB NI is a bit more complex, the achieved $f_{\max}$ improves.

Even if, the area costs and frequency penalty are not negligible, the costs to include least two or four, and even eight priority level can be assumed depending on the application and taking into account the potential benefits to have the runtime NoC QoS services on the system.

According to each QoS services integrated in the proposed NoC-based clustered MPSoC, a set of lightweight middleware API QoS support functions have been implemented. In Listing 1, we show their functionality and prototypes.

Listing 1. Middleware API QoS support

```
// Set up an end-to-end circuit
// unidirectional or full duplex (i.e., write or R/W)
int ni_open_channel (uint32_t address, bool
full_duplex);

// Tear down a circuit
// unidirectional or full duplex (i.e., write or R/W)
int ni_close_channel (uint32_t address, bool
full_duplex);

// Set high-priority in all W/R packets between an
// arbitrary CPU and a memory on the system
int setPriority(int PROC_ID, int MEM_ID, int level);

// Reset priorities in all W/R packets between an
// arbitrary CPU and a memory on the system
int resetPriority(int PROC_ID, int MEM_ID);

// Reset all priorities in all W/R packets of
// a specific CPU on the system
int resetPriorities(int PROC_ID);

// Reset all priorities W/R packet on the system
int resetAllPriorities(void);
```

The execution of each middleware function will configure at runtime the NI according the selected QoS service. The activation or configuration overhead to enable priority traffic can be considered null since the priority level is embedded directly on the request packet header on the NoC backbone. However, the time to establish/release GT circuits is not negligible. Mainly, the latency depends on the time to transmit the request/response packets along several switches from the processor until the destination memory. In (1) and (2), we express the zero-load latency in clock cycles to establish and release unidirectional and full-duplex GT circuits, respectively. In any case, in large NoC-based systems, this means tens of clock cycles

$$\mathrm{GT\_uni}_{\mathrm{time}} = 2 \cdot \left(\frac{\mathrm{request\_packet}_{\mathrm{length}}}{FLIT_{width}} + \mathrm{Num}_{\mathrm{hops}} \right) \quad (1)$$

$$\mathrm{GT\_bi}_{\mathrm{time}} = 2 \cdot \left(\frac{\mathrm{request\_packet}_{\mathrm{length}}}{\mathrm{FLIT}_{\mathrm{width}}} + \mathrm{Num}_{\mathrm{hops}} \right) + 2 \cdot \left(\frac{\mathrm{response\_packet}_{\mathrm{length}}}{\mathrm{FLIT}_{\mathrm{width}}} + \mathrm{Num}_{\mathrm{hops}} \right). \quad (2)$$

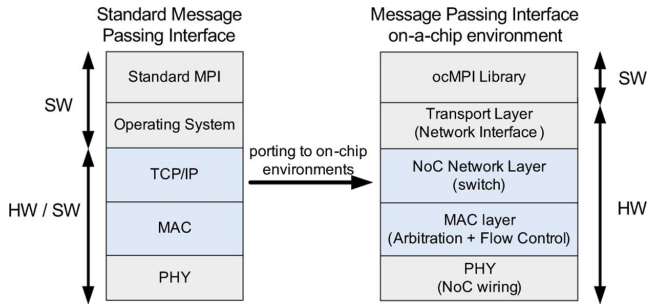

Fig. 3. MPI adaptation for NoC-based many-core systems.

V. On-Chip Message Passing Library

Message passing is a common parallel programming model, which in the form of a standard MPI API library [11], [45] can ported and optimized in many different platforms.

In this section, we show an overview of ocMPI that is our customized proprietary and MPI-compliant library targeted for the emerging MPSoCs and cluster-on-chip many-core architectures.

The ocMPI library has been implemented starting from scratch using a bottom-up approach as proposed in [46] taking as a reference the open source Open MPI project [47]. It does not rely on any operating system, and rather than use TCP/IP as the standard MPI-2 library, it uses a customized layer in order to enable message-passing on top of parallel embedded systems. Fig. 3 shows our MPI adaptation for embedded systems.

However, in contrast with our previous work [16], we have redesigned the transport layer of the ocMPI library to be tailored efficiently using the scratchpad memories for intracluster communication (i.e., each of the four Cortex-M1 processors on the left-side of each subcluster uses the first scratchpad memory, whereas the other processors in the right-side of each subcluster work with the second scratchpad memory), and the shared external ZBTRAM for intercluster communication, in the distributed-shared memory MPSoC.

The synchronization protocol to exchange data relies on a rendezvous protocol supported by means of flags/semaphores, which have been mapped on the upper address memory space of each scratchpad memory and the external memory. These flags are polled by each sender and receiver to synchronize. The lower address space is used by each processor as a message-passing buffer to exchange ocMPI messages in the proposed cluster-based MPSoC.

During the rendezvous protocol, one or more senders attempt to send data to a receiver, and then block. On the other side, the receivers are similarly requesting data, and block. Once a sender/receiver pair matches up, the data transfer occurs, and then both unblock. The rendezvous protocol itself provides a synchronization because either the sender and the receiver unblock, or neither does.

ocMPI is built-in upon a low-level interface API or transport layer which implements the rendezvous protocol. However, to hide hardware details, these functions are not directly exposed to the software programmers, and the software programmers can only see the standard `ocMPI_Send()` and `ocMPI_Recv()` functions.

The rendezvous protocol has some well-know performance inefficiencies, such as the synchronization overhead specially with small packets. However, as we show later, the efficiency of the protocol in the proposed ocMPI library running in fast on-chip interconnection, such NoCs, is acceptable even for small packets. Another problem that affects the overlapping between the communication and computation is the "early-sender" or "late-receiver" pattern. Nevertheless, as we demonstrate later, this issue can be mitigated reconfiguring and balance the workloads by means of runtime QoS services.

TABLE I
Supported Functions in the ocMPI Library

Types of MPI functions	Ported MPI functions
Management	`ocMPI_Init()`, `ocMPI_Finalize()`, `ocMPI_Initialized()`, `ocMPI_Finalized()`, `ocMPI_Comm_size()`, `ocMPI_Comm_rank()`, `ocMPI_Get_processor_name()`, `ocMPI_Get_version()`
Profiling	`ocMPI_Wtick()`, `ocMPI_Wtime()`
Point-to-point Communication	`ocMPI_Send()`, `ocMPI_Recv()`, `ocMPI_SendRecv()`
Advanced & Collective Communication	`ocMPI_Broadcast()`, `ocMPI_Barrier()` `ocMPI_Gather()`, `ocMPI_Scatter()`, `ocMPI_Reduce()`, `ocMPI_Scan()`, `ocMPI_Exscan()`, `ocMPI_Allgather()`, `ocMPI_Allreduce()`, `ocMPI_Alltoall()`

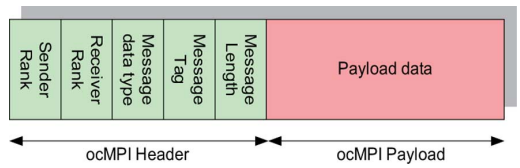

Fig. 4. ocMPI message layout.

To optimize the proposed ocMPI library, we improve the rendezvous protocol to do not require any intermediate copy and user-space buffer since the ocMPI message is stored directly on the message-passing memory. This leads to a very fast interprocess communication by means of a remote-write local-read transfers hiding the read latency on the system.

This implementation leads to a lightweight message-passing library that only uses ≈ 15 kB of memory footprint (using `armcc -O2`), which is suitable for distributed-memory embedded and clustered SoCs.

Table I shows the 23 standard MPI functions supported by ocMPI. To keep reuse and portability of legacy MPI code, the ocMPI library follows the standardized definition and prototypes of MPI-2 functions.

All ocMPI advanced collective communication routines (such as `ocMPI_Gather`, `ocMPI_Bcast()`, `ocMPI_Scatter()`, etc.) are implemented using simple point-point `ocMPI_Send()` and `ocMPI_Recv()`.

As shown in Fig. 4, each ocMPI message has the following layout: 1) source rank (4 B); 2) destination rank (4 B); 3) message tag (4 B); 4) packet datatype (4 B); 5) payload length (4 B); and, finally, 6) the payload data (a variable number of bytes). The ocMPI message packets are extremely slim to avoid big overhead for small and medium messages.

In this vertical hardware-software approach to support runtime QoS-driven reconfiguration at system-level and application-level, the next step is to expose the QoS hardware support and these middleware functions on top of the ocMPI library. In this work, rather than invoking manually the QoS middleware API, the programmer in a lightweight manner can explicitly define or annotate critical tasks according to a certain QoS level by means of using an extended API functionality of the ocMPI library [see Fig. 1(b)].

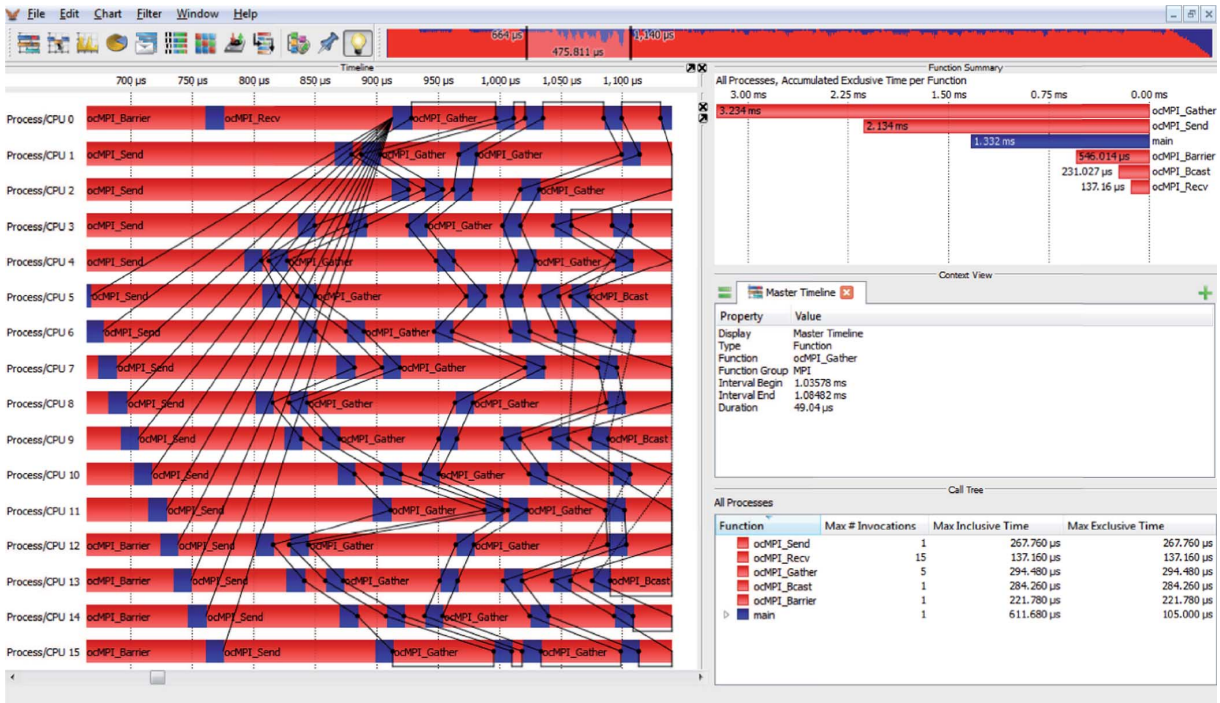

Fig. 5. Vampir view of the traces from an ocMPI program.

Thus, we extend the ocMPI library reusing part of the information on the ocMPI packet header (i.e., `ocMPI Tag`) in order to trigger specific QoS services on the system. Later, the library automatically will invoke in-lining the corresponding QoS middleware function(s) presented in Listing 1. This will enable prioritized traffic or end-to-end circuits reconfiguring the system during the execution of message-passing parallel programs for a particular tasks or group of tasks.

VI. Instrumentation and Tracing Support for Performance Analysis

The verification, the debugging, and the performance analysis of embedded MPSoCs running multiple software stacks with even runtime reconfiguration will become a hard problem when the number of cores increase. The HPC community has already faced this problem, however it has not been tackled properly in the embedded domain.

In this paper, we present a novel way to reuse some of the methods from the HPC world to be applied in the emerging many-core MPSoCs. In HPC, performance analysis and optimization specially in multicore systems is often based on the analysis of traces. In this work, we added support in the presented ocMPI library to produce OTF traces [48], [49].

OTF defines a format to represent traces which is use in large-scale parallel systems. The OTF specification describes three types of files: 1) a *.otf* file that defines the number of processors on the system; 2) a *.def* file which includes the different functions that are instrumented; and 3) a *.event* files containing the data traces of each specific event according to each processor.

We created a custom lightweight API to generate OTF events and dump them through JTAG in the proposed FPGA-based many-core MPSoC platform. Later, tools like Vampirtrace and Vampir [50], [51], Paraver [52], TAU [53] are used to view the traces and to perform, which is known as postmortem analysis, in order to evaluate the performance of the application, but also to detect bottlenecks, communication patterns, and even deadlocks.

To enable tracing, the original ocMPI library can be instrumented automatically by means of a precompiler directives (i.e., `-DTRACE_OTF`). This will inline, at the entry and the exit of each ocMPI function, the calls to generate OTF events. In addition, other user functions, can also be instrumented manually adding properly calls to the OTF trace support. Later, using the logs, we can analyze for instance, the time that the processor has been executing an `ocMPI_Bcast()`, `ocMPI_Barrier()`, ..., and/or to know how many times an ocMPI function is called. In Fig. 5 we show a trace and its associated information from a parallel program using Vampir.

Rather than a profiler, Vampir gives much more information adding at the same time dynamics and preserving the spatial and temporal behavior of the parallel program.

This is very useful, however there are several drawbacks due to the instrumentation of the original application. When the application is instrumented, a small number of instructions must be added to produce the trace and as a consequence an overhead is introduced. To reduce it, logs are stored in memory first to minimize the time spent to dump continuously the traces. Afterwards, when the execution finished or the memory buffers have been filled, the logs are flushed.

The outcome is full insight into the proposed many-core system, where we can analyze and control the execution of multiple SW stacks, or parallel applications with reconfigurability in order to improve the overall system performance.

VII. Message-Passing Evaluation in Our Clustered NoC-Based MPSoC

In this section, we investigate the performance of the proposed ocMPI library executing a broad range of benchmarks, low-level communication profiling tests, and the scalability and speedups of different message-passing parallel applications in

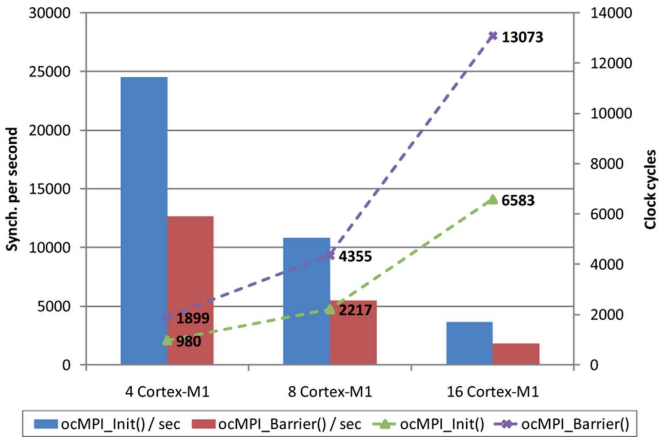

Fig. 6. Profiling of the ocMPI_Init() and ocMPI_Barrier() synchronization routines.

our distributed-shared memory ARM-based cluster-on-chip MPSoC architecture.

Apart from the tracing support presented in Section VI, in order to enable profiling in our cluster-based MPSoC, we used the Nested Vector Interrupt Controller (NVIC). The NVIC is a peripheral closely coupled with each Cortex-M1 soft-core processor. It has a very fast access which leverage a high-accuracy profiling support. The NVIC contains a memory-mapped control registers and hardware counters which can be configured to enable low-latency interrupt handling (in our case 1 ms with a reload mechanism) in order to get timestamps at runtime.

Later, this hardware infrastructure is used by `ocMPI_Wtime()` and `ocMPI_Wtick()` profiling functions. Thus, we can measure the wall-clock time of any software task running on each processor in the cluster in the same way as in traditional MPI programs, as well as to obtain the equivalent number of clock ticks consumed by the message-passing library.

A. Benchmarking the OCMPI Library

In this section, the first goal is the evaluation of the zero-load execution time of the most common ocMPI primitives to initialize and synchronize the process in message-passing parallel programs (i.e., `ocMPI_Init()` and `ocMPI_Barrier()`).

In the ocMPI library an initialization phase is used to assign dynamically the ocMPI rank to each core involved in the parallel program. In Fig. 6, we report the number of cycles of `ocMPI_Init()` to set up the `ocMPI_COMM_WORLD`. The plot shows that 980, 2217, and 6583 clock cycles are consumed to initialize the ocMPI stack in a 4, 8, and 16-core processor system, respectively. Moreover, running the MPSoC at 24 MHz, the outcome is that, for instance, we can reassign part of the ocMPI ranks within each communicator, performing up to ≈10 000 reconfigurations per second inside each eight-core subcluster, or ≈3500 in the entire 16-core system.

Similarly, in Fig. 6, we show the amount of clock cycles required to execute an `ocMPI_Barrier()` according to the number of processors involved. Barriers are often used in message-passing to synchronize all tasks involved in parallel workload. Thus, for instance, to synchronize all Cortex-M1s on a single-side of each subcluster, the barrier only takes 1899 clock cycles, whereas to execute it in the proposed 16-core cluster-on-chip, it consumes 13 073 clock cycles.

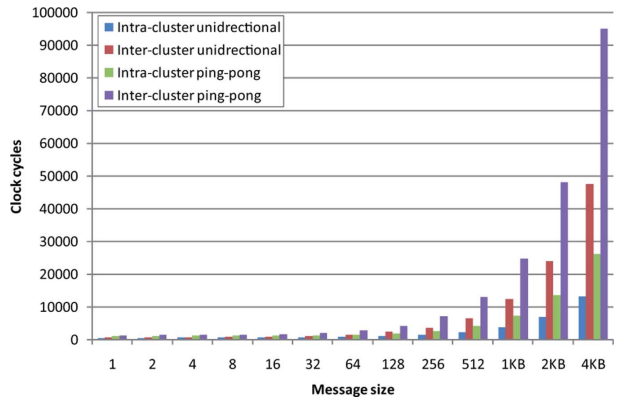

Fig. 7. Intra- and intercluster point-to-point latencies under unidirectional and ping-pong traffic.

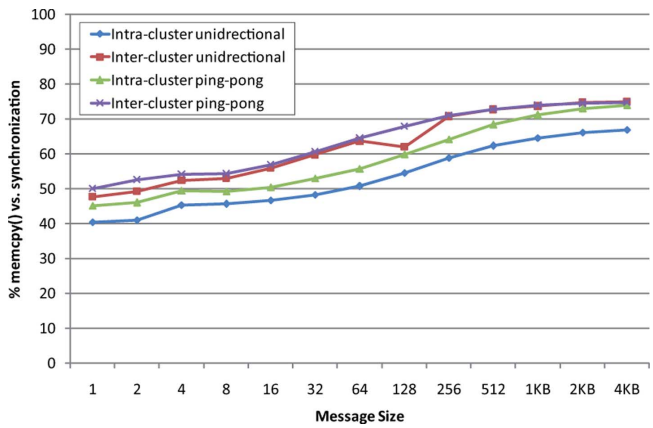

Fig. 8. Efficiency of the ocMPI synchronous rendezvous protocol under unidirectional and ping-pong traffic.

The second goal is to profile the `ocMPI_Send()` and `ocMPI_Recv()` functions using common low-level benchmarks presented MPIBench [54] in order to measure point-to-point latency.

In the proposed hierarchical clustered MPSoC platform, we can distinguish between two different types of communication: 1) intracluster communication, when the communication is between processes on the same 8-core subcluster; and 2) intercluster communication, if the communication is between tow processes on different subclusters.

Fig. 7 shows the trend of point-to-point latencies to execute unidirectional and ping-pong message-passing tests varying the payload of each ocMPI message from 1 B up to 4 kB. For instance, the latency to send a 32-bit intracluster ocMPI message is 604 and 1237 cycles, under unidirectional and ping-pong traffic, respectively. For intercluster communication, the transmission of unidirectional and ping-pong 32-bit messages takes 992 and 2021 clock cycles. Similarly, for larger message than 4 kB the peer-to-peer latencies are following the trend presented in Fig. 7.

The proposed rendezvous protocol implemented has the advantage of not requiring intermediate buffering. However, due to the synchronization between sender and receiver, it adds some latency overhead that can degrade the performance of ocMPI programs. An important metric is to show the efficiency of the rendezvous protocol for inter and intracluster communication under unidirectional and ping-pong ocMPI traffic.

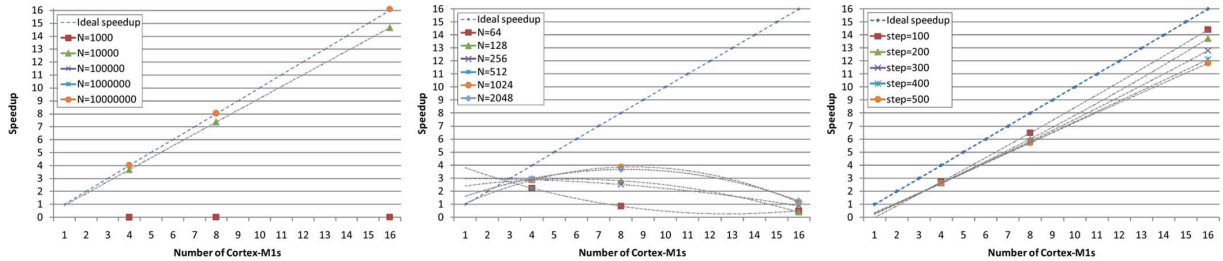

Fig. 9. Scalability of message passing applications in our ARM-based cluster-on-chip many-core platform (a) PI approximation (b) Dot product (c) Heat 2-D.

In Fig. 8, it is possible to observe that in our distributed-shared memory system, for very small messages, the efficiency of the protocol is around 40%–50%. In other words, the synchronization time is comparable to the time to copy ocMPI message payload. However, for messages of few kBs, still a small ocMPI message, the percentage rise up until about 67%–75%, which is an acceptable value for such small messages.

The efficiency of the protocol for intercluster communication is higher than for intracluster. Essentially this is because even if the time to poll the flags is a bit larger on the ZBTRAM, the overall number of pollings decreases. Besides, the overall time to copy the message data is larger than for intracluster, which makes the intercluster efficiency higher.

In the experiments presented in Fig. 8, we show that the efficiency of sending relatively small ocMPI messages (i.e., up to 4 kB) is at maximum 75% because of the synchronization during the rendezvous protocol. Nevertheless, preliminary tests with larger ocMPI messages achieve efficiencies over 80%.

B. Scalability of Parallel Applications Using OcMPI Library

In this section, we report results, in terms of runtime speedup, extracted from the execution of some scientific message-passing parallel applications in the proposed cluster-on-chip many-core MPSoC. The selected parallel applications show the tradeoffs in terms of scalability, varying the number of cores and the granularity of the problem playing with the computation and the communication ratio.

The first parallel application is the *approximation of number* π using (3). We parallelized this formula so that every processor generates a partial summation, and finally the root uses `ocMPI_Reduce()` to perform the last addition of the partial sums. This is possible because every point of (3) can be computed independently

$$\frac{\pi}{4} = \sum_{N=0}^{N=\infty} \frac{(-1)^N}{2N+1}. \tag{3}$$

In Fig. 9(a), we show that as the precision increases, then the computation to communication becomes higher and therefore the speedups are close to ideal growing linearly with to the number of processors. Even more, when $N \to \infty$ this application can be considered as an embarrassingly parallel having a coarse-grain parallelism.

As second parallel application, in Fig. 9(b), we report the results to parallelize the computation of the *dot product* between two vectors following (4)

$$a \cdot b = \sum_{i=1}^{N} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_N b_N$$
$$\text{where } a_i, b_i \in \Re. \tag{4}$$

The data is distributed using `ocMPI_Scatter()`. Once each processor receives the data, it computes the partial dot product, then the root gathers them, and it performs the last sum using `ocMPI_Reduce()`. We execute this parallel application varying N, the length of the vector, from 1 B to 2 kB.

In Fig. 9(b), it is easy to observe that, the application does not scale when more processors are used. This is because the overhead to scatter the data is not amortized during the computation phase for the selected data set.

In fact, we can highlight that in this fine-grained application, the best speedup point is when the data set is 2 kB, and the parallelization is performed in only 4-cores achieving a speedup of $2.97\times$. On the other hand, when the parallel program is executed on 16-cores the maximum speedup is only $1.25\times$.

As a final parallel application, we execute in the cluster-based MPSoC, the parallelization of *Heat 2-D grid* model in order to compute the temperature in a square surface. Equation (5) shows that the temperature of a point is dependent on the neighbor's temperature

$$U_{x,y} = U_{x,y} + C_x \cdot (U_{x+1,y} + U_{x-1,y} - 2 \cdot U_{x,y}) + C_y \cdot (U_{x,y+1} + U_{x,y-1} - 2 \cdot U_{x,y})$$
$$where\ U_{x,y} \in \Re. \tag{5}$$

We parallelize dividing the grid by columns with some points according to the number of ocMPI tasks. Thus, the temperature in the interior elements belonging to each task is independent, so that it can be computed in parallel without any communication with other tasks. On the other side, the elements on the border depend on points belonging to other tasks, and therefore, they need to exchange data with other.

In Fig. 9(c), we show the results when parallelizing a 40×40 2-D surface changing the number of steps to allow the (5) to converge. It is easy to realize that the application scales quite well with the number of processors. Thus, best-case speedup are $2.71\times$, $6.49\times$ and $14.42\times$ in our 4-, 8-, and 16-core architecture, respectively. This is a message-passing computation with medium computation to communication ratio for the selected data size.

However, an issue arises, when the number of steps increases. As shown in Fig. 9(c), the speedup decrease slightly according to the increment of the steps. This is because in between each iteration step, due to the blocking rendezvous protocol, the system blocks for a short time before to progress to the next iteration. As a consequence, at the end of the day after many iterations, it turns out in a small performance degradation.

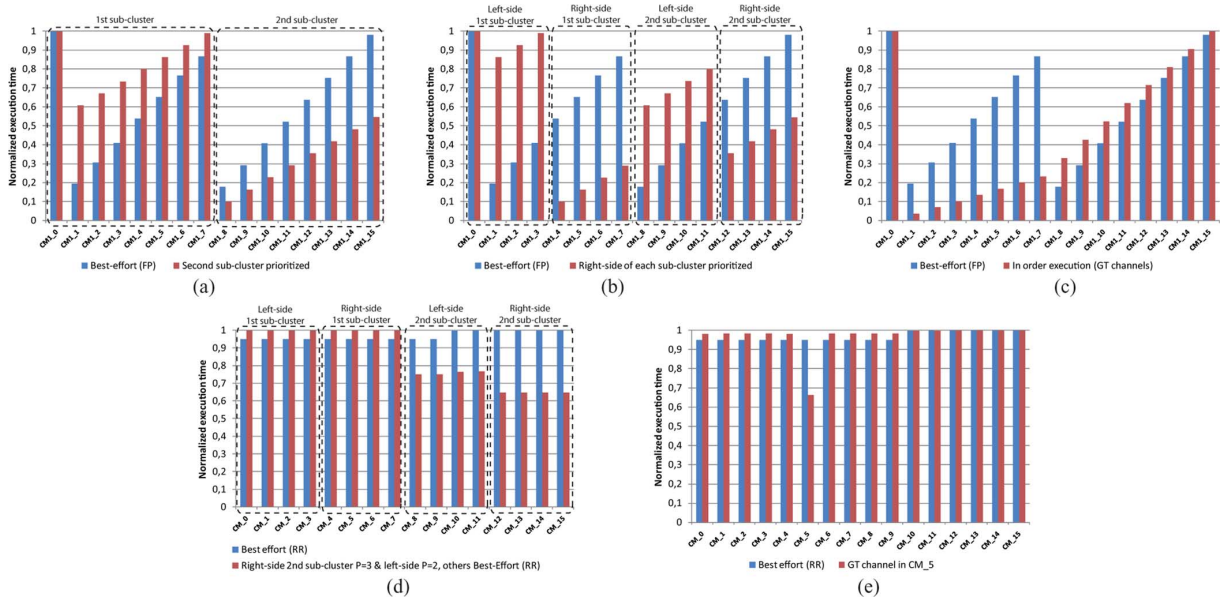

Fig. 10. QoS-driven reconfigurable parallel computing based on fixed priority (FP) and round-robin (RR) best-effort allocator. (a) Prioritized ocMPI tasks located on the second-cluster. (b) Prioritized ocMPI tasks on the right-side of each cluster. (c) Guaranteed in-order completion on ocMPI execution. (d) Second subcluster (right-side $P = 3$ and left-side $P = 2$), others RR. (e) Guaranteed throughput in Cortex-M1 with $\mathrm{rank} = 5$, others RR.

VIII. QoS-Driven Reconfigurable Parallel Computing in Our Clustered NoC-Based MPSoC

As final experiments, we explore the use of the presented runtime QoS services when multiple parallel applications are running simultaneously in the proposed ARM-based clustered MPSoC platform.

One of the big challenges in parallel programming is to manage the workloads in order to have performance improvements during the execution of multiple parallel kernels. Often, message-passing parallel programs do not achieve the desired balance even by allocating similar workload on each process. Even more, multiple applications running simultaneously in many-core system can degrade the overall execution time. This is due to different memory latencies and the access patterns to them, and the potential congestion that can occur in homogeneous and specially in heterogeneous NoC-based MPSoCs.

As a consequence, in this section, we show the benefits to reconfigure the NoC backbone using the QoS middleware API used by the ocMPI library. The target is to be able to reconfigure and to manage at runtime potential interapplication traffic from ocMPI workloads in the proposed hierarchical distributed-shared memory NoC-based MPSoC under different intra and intercluster nonzero-load latency communication patterns. In the proposed experiments, we explore the following:

- the effectiveness to assign multiple different priority-levels to a tasks or group of tasks which are executing simultaneously;
- to guarantee the throughput using end-to-end circuits, in a particular critical task or group of tasks.

In Fig. 10, we show the normalized execution time to execute a two similar benchmarks in each Cortex-M1 processor. The first benchmark is composed by three-equal subkernels and the second contains two subkernels. The benchmarks perform an intensive interprocess communication among all the 16 processors in the cluster-on-chip platform. At the end of each subkernel, a synchronization point is reached using a barrier. The idea is to set up and tear down priorities and GT channels between each `ocMPI_Barrier()` call in order to achieve different execution profiles.

In Fig. 10(a)–(c) (first row in Fig. 10) runtime QoS services are implemented on top of a fixed priority (FP) best-effort allocator, whereas in Fig. 10(d) and (e) (second row in Fig. 10), a round-robin best-effort allocator have been used. As a consequence, under no priority assignment, the tasks in each processor completes according to the corresponding best-effort scheme. However, once we use the proposed runtime QoS services, the execution behavior of the parallel program and each subkernel change radically depending on how the priorities and the GT channels are set up and torn down.

In Fig. 10(a), we show the execution of the first subkernel in a scenario when the tasks on the second subcluster, i.e., Tasks 8–15 on Cortex-M1 processors with rank 8–15, are prioritized over the first subcluster. The speedup of the prioritized tasks ranges between 7.77% and 43.52%. This is because all the tasks in the second subcluster are prioritized with the same priority level. Similarly, the average performance speedup of the prioritized subcluster is 25.64%, whereas Tasks 0–7 mapped on the nonprioritized subcluster have an average degradation of 26.56%.

In the second subkernel of the first benchmark, we explore a more complex priority scheme, triggering high-priority on each task on the right-side of each subcluster, and prioritizing at the same time, all tasks on the first subcluster over the second one. As shown in Fig. 10(b), on average Tasks 4–7 and Tasks 12–15 are sped up 51.07% and 35.93%, respectively. On the other hand, the tasks on the left-side of each subcluster which are nonprioritized are penalized 62.28% and 37.97% for the first and the second subcluster, respectively.

Finally, during the execution of the last subkernel of the first benchmark, we experiment with a completely different approach using GT channels. Often, MPI programs complete in unpredictable order due to the traffic and memory latencies

on the system. In this benchmark, the main target is to enforce a strict completion ordering by means of GT channels ensuring latency and bandwidth guarantees once the channel is established in each processor.

In Fig. 10(c), we show that in-order execution can effortlessly be achieved through GT channels triggered from ocMPI library, instead of rewriting the message-passing application to force in-order execution in software. On average, in the first subcluster, the average improvement over best-effort for Tasks 0–7 is 39.84%, but with a peak speedup in Task 7 of 63.45%. On the other hand, it is possible to observe that, the degradation in the second subcluster is not much, in fact it is only 8.69% on average.

On the other hand, in Fig. 10(d), we show the normalized execution of the first subkernel of the second benchmark when multiple priority levels are assigned in the same subcluster to a group of tasks. The setup is that, the right-side of the second subcluster is prioritized with $P = 3$ (i.e., Tasks 12–15), whereas the left-side (i.e., Tasks 8–11) is prioritized but with less priority, i.e., $P = 2$. The remaining tasks are not prioritized, and therefore they use the round-robin best-effort allocator.

The results show that all prioritized tasks with the same priority level are almost improving equally thank to the round-robin mechanism implemented on top of the runtime QoS services. Thus, Tasks 12–15 improve around 35.11%, whereas the speedup in Tasks 8–11 range between 19.99% and 23.32%. The remaining nonprioritized tasks also finish with almost perfect load balancing with a performance degradation of 0.05%.

Finally, in the second subkernel of the second benchmark, we explored a scheme where only one processor, i.e., the Cortex-M1 with $\mathrm{rank} = 5$, requires to execute a task with GT. As we can observe, in Fig. 10(e), the Task 5 finishes with a speedup of 28.74%, and the other tasks are perfectly balanced since they use again the best-effort round-robin allocator because no priorities were allocated.

In contrast, to the experiments presented in Fig. 10(a)–(c), in Fig. 10(d) and (e), under similar workloads executed in each processor, a load balancing is possible thanks to the implementation of the runtime QoS services within the round-robin allocator.

In this section, we have demonstrated that using the presented QoS-driven ocMPI library, we can effortlessly reconfigure the execution of all tasks and subkernels involved in a message passing parallel program under a fixed-priority or round-robin best-effort arbitration schemes. In addition, we can potentially deal with some performance inefficiencies, such as early-sender, late-receiver, simply by boosting this particular task of group of tasks with different priority-levels or using GT channels, reconfiguring the application traffic dynamism during the execution of generic parallel benchmarks.

IX. Conclusion and Future Work

Exposing and handling QoS services for traffic management and runtime reconfiguration on top of parallel programming models has not been tackled properly on the emerging cluster-based many-core MPSoCs. In this work, we presented a vertical hardware-software approach thanks to the well-defined NoC-based OSI-like stack in order to enable runtime QoS services on top of a lightweight message-passing library (ocMPI) for many-core on-chip systems.

We propose to abstract away the complexity of the NoC-based communication QoS services on the backbone at the hardware level, raising them up to system-level through an efficient lightweight QoS middleware API. This allows to build an infrastructure to assign different priority levels and guaranteed services during parallel computing.

In this work, both the embedded software stack, and the hardware components have been integrated in a hierarchical ARM-based distributed-memory clustered MPSoC prototype. Additionally, a set of benchmarks and parallel application have been executed showing good results, in terms protocol efficiency (i.e., 67%–75% with medium size ocMPI packets), fast interprocess communication (i.e., few hundred cycles to send/recv ocMPI small packets), and acceptable scalability in the proposed distributed-memory clustered NoC-based MPSoC.

Furthermore, using the presented lightweight software stack and running ocMPI parallel programs in clustered MPSoCs, we illustrate the potential benefits of QoS-driven reconfigurable parallel computing using a message-passing parallel programming model. For the tested communication-intensive benchmarks, an average improvement of around 45% can be achieved depending on the best-effort allocator, with a peak of speedup of 63.45% when GT end-to-end circuits are used.

The results encourage us to believe that the proposed QoS-aware ocMPI library even if is not the only possible solution to enable parallel computing and runtime reconfiguration, it is a viable solution to manage workloads in highly parallel NoC-based many-core systems with multiple running applications. Future work will focus on further exploration on how to select properly QoS services in more complex scenarios.

References

[1] S. Borkar, "Thousand core chips: A technology perspective," in *Proc. 44th Annu. Design Automation Conf. (DAC)*, 2007, pp. 746–749.

[2] A. Jerraya and W. Wolf, *Multiprocessor Systems-on-Chips*. San Mateo, CA: Morgan Kaufmann, 2005.

[3] R. Obermaisser, H. Kopetz, and C. Paukovits, "A cross-domain multiprocessor system-on-a-chip for embedded real-time systems," *IEEE Trans. Ind. Inf.*, vol. 6, no. 4, pp. 548–567, Nov. 2010.

[4] J. Howard *et al.*, "A 48-core IA-32 message-passing processor with DVFS in 45 nm CMOS," in *Proc. IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers (ISSCC)*, Feb. 2010, pp. 108–109.

[5] S. R. Vangal *et al.*, "An 80-tile sub-100-W TeraFLOPS processor in 65-nm CMOS," *IEEE J. Solid-State Circuits*, vol. 43, no. 1, pp. 29–41, Jan. 2008.

[6] S. Bell *et al.*, "TILE64—Processor: A 64-core SoC with mesh interconnect," in *Solid-State Circuits Conf., 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, Feb. 2008, pp. 88–598.

[7] L. Benini and G. D. Micheli, *Networks on Chips: Technology and Tools*. San Francisco, CA: Morgan Kaufmann, 2006.

[8] *AMBA 3 AXI overview* ARM Ltd., (2005). [Online]. Available: http://www.arm.com/products/system-ip/interconnect/axi/index.php

[9] *Open Core Protocol Standard* OCP International Partnership (OCP-IP), (2003). [Online]. Available: http://www.ocpip.org/home

[10] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, 1998.

[11] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming With the Message-Passing Interface*. Cambridge, MA: MIT Press, 1999.

[12] L. Seiler *et al.*, "Larrabee: A many-core x86 architecture for visual computing," *IEEE Micro*, vol. 29, no. 1, pp. 10–21, Jan. 2009.

[13] J. Nickolls and W. Dally, "The GPU computing era," *IEEE Micro*, vol. 30, no. 2, pp. 56–69, Mar./Apr. 2010.

[14] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken, "CoMPSoC: A template for composable and predictable multi-processor system on chips," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, no. 1, pp. 1–24, 2009.

[15] E. Carara, N. Calazans, and F. Moraes, "Managing QoS flows at task level in NoC-based MPSoCs," in *Proc. IFIP Int. Conf. Very Large Scale Integr. (VLSI-SoC)*, 2009, pp. 133–138.

[16] J. Joven, F. Angiolini, D. Castells-Rufas, G. De Micheli, and J. Carrabina, "QoS-ocMPI: QoS-aware on-chip message passing library for NoC-based many-core MPSoCs," presented at the 2nd Workshop Program. Models Emerging Archit. (PMEA), Viena, Austria, 2010.

[17] T. Marescaux and H. Corporaal, "Introducing the SuperGT network-on-chip; SuperGT QoS: More than just GT," in *Proc. 44th ACM/IEEE Design Automat. Conf. (DAC)*, Jun. 4–8, 2007, pp. 116–121.

[18] E. Rijpkema, K. G. W. Goossens, A. Radulescu, J. Dielissen, J.. van Meerbergen, P. Wielage, and E. Waterlander, "Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip," in *Proc. Design, Automat. Test Eur. Conf. Exhib.*, 2003, pp. 350–355.

[19] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny, "QNoC: QoS architecture and design process for network on chip," *J. Syst. Archit.*, vol. 50, pp. 105–128, 2004.

[20] B. Li *et al.*, "CoQoS: Coordinating QoS-aware shared resources in NoC-based SoCs," *J. Parallel Distrib. Comput.*, vol. 71, pp. 700–713, May 2011.

[21] S. Murali, M. Coenen, A. Radulescu, K. Goossens, and G. De Micheli, "A methodology for mapping multiple use-cases onto networks on chips," in *Proc. Design, Automat. Test Eur. (DATE)*, Mar. 6–10, 2006, vol. 1, pp. 1–6.

[22] A. Hansson and K. Goossens, "Trade-offs in the configuration of a network on chip for multiple use-cases," in *NOCS '07: Proc. First Int. Symp. Networks-on-Chip*, 2007, pp. 233–242.

[23] T. Cucinotta, L. Palopoli, L. Abeni, D. Faggioli, and G. Lipari, "On the integration of application level and resource level QoS control for real-time applications," *IEEE Trans. Ind. Inf.*, vol. 6, no. 4, pp. 479–491, Nov. 2010.

[24] S. Whitty and R. Ernst, "A bandwidth optimized SDRAM controller for the MORPHEUS reconfigurable architecture," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE Int. Symp.*, Apr. 2008, pp. 1–8.

[25] D. Göhringer, L. Meder, M. Hübner, and J. Becker, "Adaptive multi-client network-on-chip memory," in *Proc. Int. Conf. Reconfigurable Comput. FPGAs (ReConFig)*, Dec. 2011, pp. 7–12.

[26] F. Liu and V. Chaudhary, "Extending OpenMP for heterogeneous chip multiprocessors," in *Proc. Int. Conf. Parallel Process.*, 2003, pp. 161–168.

[27] A. Marongiu and L. Benini, "Efficient OpenMP support and extensions for MPSoCs with explicitly managed memory hierarchy," in *Proc. Design, Automat. Test Eur. Conf.. Exhib. (DATE)*, Apr. 20–24, 2009, pp. 809–814.

[28] J. Joven, A. Marongiu, F. Angiolini, L. Benini, and G. De Micheli, "Exploring programming model-driven QoS support for NoC-based platforms," in *Proc. 8th IEEE/ACM/IFIP Int. Conf. Hardw./Softw. Codesign Syst. Synth.*, 2010, pp. 65–74, CODES/ISSS '10.

[29] B. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, and A. Gatherer, "Implementing OpenMP on a high performance embedded multicore MPSoC," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. (IPDPS)*, 2009, pp. 1–8.

[30] W.-C. Jeun and S. Ha, "Effective OpenMP implementation and translation for multiprocessor system-on-chip without using OS," in *Proc. Asia South Pac. Design Automat. Conf. (ASP-DAC)*, Jan. 23–26, 2007, pp. 44–49.

[31] T. Mattson *et al.*, "The 48-core SCC processor: The programmer's view," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, 2010, pp. 1–11.

[32] R. F. van der Wijngaart, T. G. Mattson, and W. Haas, "Light-weight communications on Intel's single-chip cloud computer processor," *SIGOPS Oper. Syst. Rev.*, vol. 45, pp. 73–83, Feb. 2011.

[33] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani, "MPI microtask for programming the cell broadband engine processor," *IBM Syst. J.*, vol. 45, no. 1, pp. 85–102, 2006.

[34] J. Psota and A. Agarwal, "rMPI: Message passing on multicore processors with on-chip interconnect," *Lecture Notes Comput. Sci.*, vol. 4917, pp. 22–22, 2008.

[35] M. Saldaña and P. Chow, "TMD-MPI: An MPI implementation for multiple processors across multiple FPGAs," in *Proc. Int. Conf. Field Program. Logic Appl. (FPL)*, Aug. 2006, pp. 1–6.

[36] P. Mahr, C. Lorchner, H. Ishebabi, and C. Bobda, "SoC-MPI: A flexible message passing library for multiprocessor systems-on-chips," in *Proc. Int. Conf. Reconfigurable Comput. FPGAs (ReConFig)*, Dec. 3–5, 2008, pp. 187–192.

[37] D. Göhringer, M. Hübner, L. Hugot-Derville, and J. Becker, "Message passing interface support for the runtime adaptive multi-processor system-on-Chip RAMPSoC," in *Proc. Int. Conf. Embedded Comput. Syst. (SAMOS)*, Jul. 2010, pp. 357–364.

[38] N. Saint-Jean, P. Benoit, G. Sassatelli, L. Torres, and M. Robert, "MPI-based adaptive task migration support on the HS-scale system," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, 2008, pp. 105–110.

[39] A. J. Roy, I. Foster, W. Gropp, B. Toonen, N. Karonis, and V. Sander, "MPICH-GQ: Quality-of-service for message passing programs," in *Proc. ACM/IEEE Conf. Supercomput. (CDROM)*, 2000, pp. 19–19.

[40] R. Y. S. Kawasaki, L. A. H. G. Oliveira, C. R. L. Francê, D. L. Cardoso, M. M. Coutinho, and Á. Santana, "Towards the parallel computing based on quality of service," in *Proc. Int. Symp. Parallel Distrib. Comput.*, 2003, pp. 131–131.

[41] *ARM Versatile Product Family* ARM Ltd. [Online]. Available: http://www.arm.com/products/tools/development-boards/versatile/index.php

[42] J. Joven, P. Strid, D. Castells-Rufas, A. Bagdia, G. De Micheli, and J. Carrabina, "HW-SW implementation of a decoupled FPU for ARM-based Cortex-M1 SoCs in FPGAs," in *Proc. 6th IEEE Int. Symp. Ind. Embedded Syst. (SIES)*, Jun. 2011, pp. 1–8.

[43] D. Bertozzi and L. Benini, "Xpipes: A network-on-chip architecture for gigascale systems-on-chip," *IEEE Circuits Syst. Mag.*, vol. 4, no. 2, pp. 18–31, 2004.

[44] S. Stergiou, F. Angiolini, S. Carta, L. Raffo, D. Bertozzi, and G. De Micheli, "×pipes lite: A synthesis oriented design library for networks on chips," in *Proc. Design, Automat. Test Eur.*, 2005, pp. 1188–1193.

[45] D. W. Walker and J. J. Dongarra, "MPI: A standard message passing interface," *Supercomputer*, vol. 12, pp. 56–68, 1996.

[46] T. P. McMahon and A. Skjellum, "eMPI/eMPICH: Embedding MPI," in *Proc. 2nd MPI Develop. Conf.*, Jul. 1–2, 1996, pp. 180–184.

[47] Open MPI: Open Source High Performance Computing (2004). [Online]. Available: http://www.open-mpi.org/

[48] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. Nagel, "Introducing the open trace format (OTF)," in *Proc. Comput. Sci.—(ICCS)*, 2006, vol. 3992, pp. 526–533.

[49] A. D. Malony and W. E. Nagel, "The open trace format (OTF) and open tracing for HPC," in *Proc. ACM/IEEE Conf. Supercomput.*, 2006, p. 24.

[50] M. S. Müller, A. Knüpfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, and W. E. Nagel, "Developing scalable applications with vampir, vampirserver and vampirtrace," in *PARCO*, 2007, pp. 637–644.

[51] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and analysis of MPI resources," *Supercomputer*, vol. 12, pp. 69–80, 1996.

[52] V. Pillet, J. Labarta, T. Cortés, and S. Girona, "PARAVER: A tool to visualize and analyse parallel code," in *Proc. WoTUG-18: Transputer Occam Develop. Vol. 44—Transputer Occam Eng.*, 1995, pp. 17–31.

[53] R. Bell, A. Malony, and S. Shende, "ParaProf: A portable, extensible, and scalable tool for parallel performance profile analysis," *Euro–Par Parallel Process.*, vol. 2790, pp. 17–26, 2003.

[54] D. Grove and P. Coddington, "Precise MPI performance measurement using MPIBench," in *Proc. HPC Asia*, 2001.

Jaume Joven received the M.S. and Ph.D. degrees in computer science from the Universitat Autonòma de Barcelona (UAB), Bellaterra, Spain, in 2004 and 2009, respectively.

He is currently a postdoctoral Researcher in École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland. His main research interests are focused on the embedded NoC-based MPSoCs, ranging from circuit and system-level design of application-specific NoCs, up to system-level software development for runtime QoS resource allocation, as well as middleware and parallel programming models.

Dr. Joven received the Best Paper Award at the PDP Conference in 2008, and a Best Paper nomination in CODES + ISSS in 2010.

Akash Bagdia received the dual degree B.E.(Hons.) degree in electrical and electronics and the M.Sc.(Hons) degree in physics from the Birla Institute of Technology and Science, Pilani, India, in 2003, and the M.Sc. degree in microelectronics systems and telecommunication from Liverpool University, London, U.K., in 2007.

He is a Senior Engineer working with the System Research Group, ARM Limited, Cambridge, U.K. His research interests include design for high performance homogeneous and heterogeneous computing systems with focus on on-chip interconnects and memory controllers.

Federico Angiolini received the M.S. and Ph.D. degrees in electronic engineering from the University of Bologna, Bologna, Italy, in 2003 and 2008, respectively.

He is Vice President of Engineering and Cofounder of iNoCs SaRL, Switzerland, a company focused on NoC design and optimization. He has published more than 30 papers and book chapters on NoCs, MPSoC systems, multicore virtual platforms, and on-chip memory hierarchies. His current research interests include NoC architectures and NoC EDA tools.

Per Strid received the M.Sc. in electrical engineering from the Royal Institute of Technology, Stockholm, Sweden.

He is currently a Senior Principal Researcher working with the R&D Department, ARM Limited, Cambridge, U.K. Prior to this position, he was working as an ASIC Designer with Ericsson. His research interests include the design of MPSoC systems, processor microarchitecture, memory hierarchies and subsystems, and power characterization of AMBA systems.

David Castells-Rufas received the B.Sc. degree in computer science and the M.Sc. in research in microelectronics from the Universitat Autònoma de Barcelona, Bellaterra, Spain, where he is currently working toward the Ph.D. degree in microelectronics.

He is currently the Head of the Embedded Systems Unit at CAIAC Research Center, Universitat Autònoma de Barcelona. He is also Associate Lecturer in the Microelectronics Department of the same university. His primary research interests include parallel computing, NoC-based multiprocessor systems, and parallel programming models.

Eduard Fernandez-Alonso received the B.Sc. degree in computer science and the M.Sc. degree in micro- and nanoelectronics from the Universitat Autònoma de Barcelona, Bellaterra, Spain, in 2008 and 2009, respectively, where he is currently working toward the Ph.D. degree.

He is currently with the CaiaC (the center for research in ambient intelligence and accessibility in Catalonia), Research Center, Universitat Autònoma de Barcelona. His main research interests include parallel computing, NoC-based multiprocessor systems, and parallel programming models.

Jordi Carrabina graduated in physics from the University Autonoma of Barcelona (UAB), Bellaterra, Barcelona, Spain, in 1986, and received the M.S. and Ph.D. degrees in microelectronics from the Computer Science Program, UAB, in 1988 and 1991, respectively.

In 1986, he joined the National Center for Microelectronics (CNM-CSIC), Madrid, Spain, where he was collaborating until 1996. Since 1990, he has been an Associate Professor with the Department of Computer Science, UAB. In 2005, he joined the new Microelectronics and Electronic Systems Department, heading the research group Embedded Computation in HW/SW Platforms and Systems Laboratory and CEPHIS, technology transfer node from the Catalan IT Network. Since 2010, he has been heading the new Center for Ambient Intelligence and Accessibility of Catalonia. He is teaching electronics engineering and computer science at the Engineering School, UAB, and in the Masters of micro- and nanoelectronics engineering and multimedia technologies at UAB, and embedded systems at UPV-EHU. He has given courses in several universities in Spain, Europe, and South America. He has been a consultant for different international and small and medium enterprises (SMEs) companies. During last five years, he has coauthored more 30 papers in journals and conferences. He also led the UAB contribution to many R&D projects and contracts with partners in the ICT domain. His main interests are microelectronic systems oriented to embedded platform-based design using system-level design methodologies using SoC/NoC architectures, and printed microelectronics technologies in the ambient intelligence domain.

Giovanni De Micheli (S'79–M'83–SM'89–F'94) received the nuclear engineer degree from Politecnico di Milano, Italy, in 1979, the M.Sc. and Ph.D. degree in electrical engineering and computer science from University of California, Berkeley, in 1983 and 1983, respectively.

He is currently a Professor and the Director of the Institute of Electrical Engineering and of the Integrated Systems Center, EPFL, Lausanne, Switzerland. He is the Program Leader of the Nano-Tera.ch Program. He was a Professor with the Electrical Engineering Department, Stanford University, Stanford, CA. He is the author of "Synthesis and Optimization of Digital Circuits" (New York: McGraw-Hill, 1994), and a coauthor and/or a coeditor of eight other books and over 400 technical articles. His current research interests include several aspects of design technologies for integrated circuits and systems, such as synthesis for emerging technologies, NoCs, and 3-D integration. He is also interested in heterogeneous platform designs including electrical components and biosensors, as well as in data processing of biomedical information.

Prof. Micheli is a Fellow of ACM. He has been serving IEEE in several capacities, including Division 1 Director from 2008 to 2009, Cofounder and President Elect of the IEEE Council on EDA from 2005 to 2007, President of the IEEE CAS Society, in 2003, and the Editor-in-Chief of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS from 1987 to 2001. He has been the Chair of several conferences, including DATE in 2010, pHealth in 2006, VLSI SOC in 2006, DAC in 2000, and ICCD in 1989. He received the D. Pederson Award for the Best Paper on the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS in 1987, two Best Paper Awards at the Design Automation Conference, in 1983 and 1993, and the Golden Jubilee Medal for outstanding contributions to the IEEE CAS Society in 2000. He was the recipient of the 2003 IEEE Emanuel Piore Award for contributions to computer-aided synthesis of digital systems and a Best Paper Award at the DATE Conference in 2005.