QoS-ocMPI: QoS-aware on-chip Message Passing Library for NoC-based Many-Core MPSoCs

Jaume Joven*†, Federico Angiolini†, David Castells-Rufas*, Giovanni De Micheli†, Jordi Carrabina-Bordoll*
*CEPHIS-MISE, Campus UAB, Building Q (ETSE), 08193 Bellaterra, Spain
Email: {jaume.joven, david.castells, jordi.carrabina}@uab.es
†LSI-EPFL, Station 14, 1015 Lausanne, Switzerland
Email: {federico.angiolini, giovanni.demicheli}@epfl.ch

***Abstract*—Homogeneous and heterogeneous NoC-based many-core MPSoCs are becoming widespread in many application areas. The diversity and spare network traffic characteristics generated by the IPs makes mandatory to provide certain Quality of Service (QoS) support for critical traffic streams on the system at application level even from the parallel programming model.**

In this paper, we present a hardware-software approach to enable QoS from the parallel programming model on the emerging NoC-based many-core MPSoCs. We designed NoC hardware QoS support, and the associated middleware API which enables runtime QoS on parallel programs. Additionally, a QoS-aware on-chip Message Passing Interface (ocMPI) stack is presented where QoS streams can be handled automatically on the system by means of task annotation on the ocMPI library, in order to distribute and balance workload under congestion, guaranteed throughput and latency bounds of critical processes and, in general to boost and meet QoS application requirements.

Our experimental results during the execution of message passing parallel programs using prioritized and guaranteed services extensions on the QoS-aware ocMPI library show an average speedup of ≈15% and ≈35%, respectivelly.

I. Introduction

Nowadays, the current trends in homogeneous or heterogeneous many-core systems [1] require easy programming and SoC design to deliver and improve the system performance under very constrained power budgets. At architecture level, NoCs (Networks-on-Chips) [2][3][4] have been proposed as the fabric to interconnect the IP blocks to create highly parallel scalable systems at reasonable hardware cost. These systems can be either homogeneous (e.g. many-cores) [5][6][7] or heterogeneous (e.g. embedded SoCs for mobile applications), featuring a mix of general purpose processors, memories, DSPs, multimedia accelerators, etc.

In both cases, different levels of Quality-of-Service (QoS), such as Best-Effort (BE) and Guaranteed-Throughput (GT) or low latency traffic classes, should be available to applications, allowing designers to carefully allocate communication resources in a prioritized manner according to the application requirements.

In addition, these facilities must be controllable by the software stacks (i.e. firmware and middleware) to tolerate software updates over the lifetime of the chip and to make incremental optimizations. Additionally, these chips can potentially be used in various application scenarios, also called *use cases*, even after tape-out, and therefore the various IP blocks may operate at different performance points as the *use cases* alternate at runtime. Each of these effects implies additional unpredictability of the on-chip traffic patterns, possibly rendering their static characterization impractical or overly conservative.

Thus, applications and/or the programming environment chosen for their development/execution should have some degree of control over the available NoC services. Furthermore, the access to the available NoC services should be mediated by an easily usable API, which must offer low-overhead and full compatibility with mainstream multi-core programming approaches.

In this way, the application programmer or the software execution layer (be it an OS, a custom support middleware or runtime environment) by means of a simple annotation upon critical flows can exploit the hardware resources and meet the performance requirements of the application.

In traditional multiprocessor architectures different parallel programming models API libraries have been proposed according how the memory hierarchy is organized. Most widespread are OpenMP [8] for shared-memory architectures and Message Passing Interface (MPI) [9][10] for distributed memory systems. MPI emerged as a widely used standard for writing message-passing programs in many-core systems.

In this work, we propose a QoS-aware lightweight on-chip MPI (ocMPI) software stack targeted to enhance performance of the emerging NoC-based many-core MPSoC application by provisioning runtime QoS on critical application tasks. With this customized stack, we enhance the programmability of these systems providing a well-known message passing programming model to enable parallel programming, and potentially we can deal with QoS requirement imposed at task level.

Because of its complexity, we adopt a layered approach that incrementally abstracts away hardware-specific details from QoS hardware support present at NoC level, and vertically exposes at higher levels on the ocMPI library. By means of annotation we can create create privileged streams during ocMPI program execution, where critical tasks are mapped with different priorities or using guaranteed services during synchronization and message passing, which leverage to important application-level benefits.

Thus, this work has two folds: *(i)* we describe the hardware modules (i.e. memory buffer and synchronization modules), and the QoS support necessary to set up QoS-aware message passing, and *(ii)* we expose the QoS through low-level and middleware API by extending our ocMPI library.

Under this framework, we explore multiple options to effectively utilize the QoS concepts to enhance the cluster on chip performance. Experimental results on a set of representative ocMPI parallel parallel programs show that under different traffic patterns according to ocMPI calls, the use of prioritized transactions boosts the overall execution time of each prioritized task up to ≈15%. On the other hand, the use of guaranteed services on critical threads ensure latency bounds, and speedup the critical tasks by ≈35%.

This paper is organized as follows. Section II presents related work on parallel programming models and QoS support management on NoC-based MPSoCs. Section III describes the hardware support to enable runtime QoS at the NoC level. Section IV describes a comprehensive layered view of our hardware and software platform. Section V describe briefly the developed NoC-based many-core platform. Section VI explains the QoS middleware API to enable runtime QoS on the NoC-based system. Section VII show in detail the ocMPI library and the extensions implemented to support runtime QoS on parallel applications. Section VIII describes our experimental framework, the selected benchmarks and the results obtained. Finally, Section IX concludes the paper.

II. Related Work

In traditional networks [11][12] many techniques can be applied to provide QoS, but this scenario changes radically when designing NoC-based MPSoCs due to largely different objectives (e.g. nanosecond-level latencies), opportunities (e.g. more control on the software stack) and constraints (e.g. minimum buffering to minimize area and power costs). QoS for NoCs is impractically implemented in hardware only, due to the large and hard-to-determine number of possible use cases at runtime; a proper mix of hardware facilities and software-controlled management policies is vital to achieving efficient results.

In QoS, the target as proposed in [13][14][15] is to combine BE with GT streams by handling them for instance with a time-slot allocation mechanism in the NoC switches. A more generic approach following the idea presented in QNoC in [16] to split the NoC traffic in different QoS classes is reported in [17][18] by shwowing a methodology to map multiple use-cases or traffic patterns onto a NoC architecture, satisfying the constraints of each use-case. However, none of these works tackles the problem on how to expose QoS features across the software stack.

On the other hand, due to the necessity to maximize performance/power figures on the new emerging NoC-based MPSoCs, recently a wide number of specific OpenMP [19][20][21][22] and MPI libraries [23][24][25][26] have been customized to enable application level parallelism.

In addition, The Multicore Associations offers MCAPI [27] a powerful alternative API for multicore architectures that is optimized for low-latency inter-core networks, and boasts a small memory footprint that can fit into in-core memory

Through our research the main contribution is to explore and shed some light by extending previous works by means of providing BE-GT QoS features on the NoC backbone but exposing them through the software stack all the way up until the parallel programming model (i.e our ocMPI library).

Even if this work is inspired in [28][29][30] where the authors present QoS at task level and on parallel programming models, to the best of our knowledge, the approach detailed in this paper represents the first attempt to have a complete QoS-aware software stack on the new emerging NoC-based many-core MPSoCs. This work leads to the strong challenge to program and exploit efficiently at higher levels of abstraction all the potential present on the execution platform, as well as to deal with the inherent application dynamism.

III. QoS Support at NoC Level

To enable runtime QoS at the task or application level, we extend the Network Interfaces (NIs) to include a set of configuration registers, memory-mapped in the address space of the system, to program QoS features on the fly. These registers can be programmed in different ways to provide different types of QoS. When the application demands a given QoS level, the NI correspondingly tags the packet with 4-bit QoS field. The possible QoS tags are shown in Listing 1.

Listing 1. QoS encodings

```
// Priority levels
#define ENC_QOS_HIGH_PRIORITY_PACKET_0   4'b0000
#define ENC_QOS_HIGH_PRIORITY_PACKET_1   4'b0001
#define ENC_QOS_HIGH_PRIORITY_PACKET_2   4'b0010
#define ENC_QOS_HIGH_PRIORITY_PACKET_3   4'b0011
#define ENC_QOS_HIGH_PRIORITY_PACKET_4   4'b0100
#define ENC_QOS_HIGH_PRIORITY_PACKET_5   4'b0101
#define ENC_QOS_HIGH_PRIORITY_PACKET_6   4'b0110
#define ENC_QOS_HIGH_PRIORITY_PACKET_7   4'b0111

// Open/close circuits
#define ENC_QOS_OPEN_CHANNEL             4'b1000
#define ENC_QOS_CLOSE_CHANNEL            4'b1001
```

As shown in Figure 1, at switch level, we design a configurable (up to 8-levels) priority scheme to support QoS which relies on *Fixed Priority and Round Robin* BE priority mechanisms based on [11] already implemented in the ×pipes library [31][32].

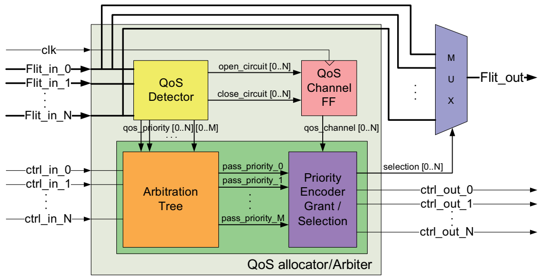

Fig. 1. QoS support extension on allocator/arbiter

On the other hand, the QoS support to establish/release circuits in order to guaranteed throughput is based on applying circuit switching over the wormhole NoC. We extend the arbitration and grant generation in the switch adding hardware support to store whether an end-to-end connection (circuit) is established or not.

IV. Layered View of our QoS-aware Software Stack

Usually HW-SW networked systems are organized along different abstraction layers in order to hide the complexity of the whole system, and expose the transparent interactions between components. In particular, in [3][33][34][35][36] the use of the micro-network stack is proposed for NoC-based systems based on the well-know ISO/OSI model [37]. Figure 2 shows our HW-SW components and the layered view our NoC-based MPSoCs platform.

- *Application layer:* At the topmost level of the software stack there is the parallel application, i.e. the ocMPI program. Parallel execution is supported by the underlying architecture and hardware support as well as the ocMPI library. QoS features are integrated within the library, and are implemented as a wrapper around the middleware API.
- *Transport layer:* is in charge of injecting/receiving packets using NIs initiator and target over the on-chip network between two end-points (i.e. processors and memories), respectively.
- *Network layer:* is responsible for the transmission/reception of packets using BE or the QoS features.

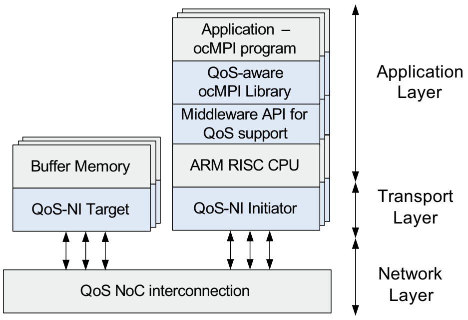

Fig. 2. HW-SW layered view of our NoC-based MPSoC architecture

V. Overview of NoC-based MPSoC Architecture

Our architecture shown in Figure 3, is an instance 4x4 2D Mesh many-core MPSoC platform. The NoC is developed using ×pipes library [31][32]. Later, the hardware components in conjunction with the QoS-aware software stack have been integrated in MPARM [38], a full-featured SystemC full system simulator based on ARM processor.

As shown in Figure 3, each tile of our NoC-based MPSoC architecture includes an ARM (with L1 I/D cache). On the whole system there is only one master ARM tile (usually ARM_0), and it is in charge of supervision the execution, whereas the remaining nodes act as ARM slaves. Each ARM-based tile also includes on the same switch the hardware support to support message passing: *(i)* a buffer memory that is used as an internal buffer for the ocMPI library and *(ii)* a synchronization module, which is in charge to notify (as a fast interrupt-like device) whether is an ongoing packet, they are packets to be received, etc.

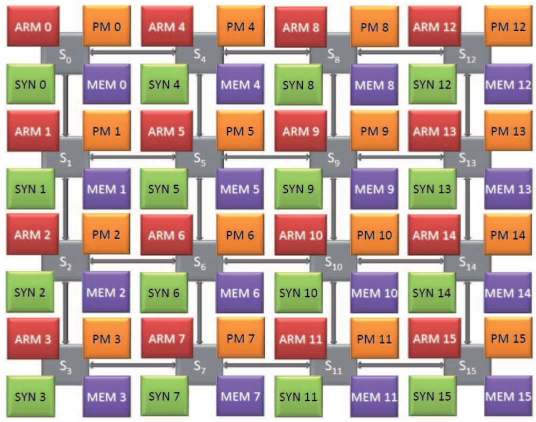

Fig. 3. NoC-based MPSoC platform

In order to get an efficient message-passing, we choose a distributed synchronization scheme including on each tile a synchronization module. Thus, the poll from the ARM processors on the synchronization module can be performed locally on each tile, and as a consequence, no additional traffic is injected across the NoC, unless a remote lock/unlock notification is performed.

VI. Low-level QoS Support and Middleware APIs - Interfacing NIs with Application Level

In order to exploit QoS features in an efficient way, a set of low-level GT QoS API have been implemented in order to establish/release channels using the programmable NI. In Listing 2, we describe the functions of which the QoS middleware API is made from.

Listing 2. Low-level 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);
```

In order to make a synergy with the programming model layer with functions that closely resemble the ocMPI semantics, we provide three middleware API primitives to set/release priority transactions, and two more to send/receive streams of data with GT channels, respectively. These functions are described in Listing 3.

The activation overhead of the QoS support above mentioned in Section III is null since the QoS field is directly embedded in the packet header as a 4-bit field. On the other side, the QoS middleware API software to program the NI and the system is based on the memory-mapped store transactions, i.e. few clock cycles depending on the NoC fabric.

Listing 3. Middleware API QoS support

```
// Set high-priority in all W/R packets between an
// arbitrary CPU and a Shared 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 Shared Memory on the system
int resetPriority(int PROC_ID, int MEM_ID);

// Reset all priorities W/R packet on system
int resetPriorities(void);

// Functions to send/receive stream of data with QoS
int sendStreamQoS(byte *buffer, int length, int MEM_ID);
int recvStreamQoS(byte *buffer, int length, int MEM_ID);
```

VII. QoS-AWARE ocMPI SOFTWARE LIBRARY

In this section, we explain in detail the ocMPI software library targeted for next generation NoC-based systems. The ocMPI library have been implemented starting from scratch taking as starting point the source code of Open MPI initiative [10] and doing a bottom-up approach as proposed in [39] with several refinement phases to get a lightweight stack.

Thus, we selected a minimal working and subset of standard MPI functions. This task is needed because MPI contains more than one hundred functions, most of them not useful in NoC-based MPSoC scenarios (e.g. the generation of virtual topologies is useless we can design real application-specific topologies [40][41][42]). As shown in Figure 4, the the lower layers of the selected MPI standard functions have been modified in order to send/receive packets by creating transactions through the NI to the NoC, and vice versa.

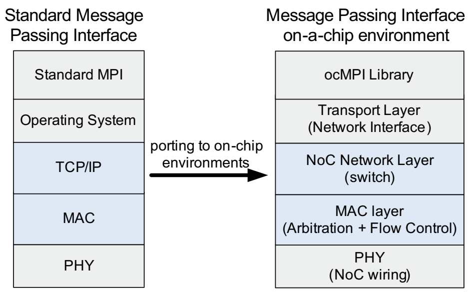

Fig. 4. MPI adaptation for NoC-based many-core systems

The ocMPI implementations is completely layered and advanced communication routines (such as `ocMPI_Gather`, `ocMPI_Scatter()`, `ocMPI_Bcast()`, etc.) are implemented using simple point-to-point routines, such as `ocMPI_Send()` and `ocMPI_Receive()`.

Furthermore, our ocMPI implementation does not depend of an OS. We define the number of processors involved in the NoC-based MPSoC using a configuration file, and at compilation time the master is defined by means of a precompiler directive `-DMASTER`. The assignment of the rank to each processor is performed at runtime when we call `ocMPI_Init()` function.

As shown in Figure 5, we defined a slim and extensible ocMPI packet format which is divided in two parts: *(i)* an ocMPI header of 20 bytes with contains the message passing protocol information, and *(ii)* a variable length payload which essentially includes the payload of the data to be sent.

Furthermore, to identify a packet in the NoC-based system, each ocMPI message has the following envelope: *(i)* Source rank (4 bytes), *(ii)* Destination rank (4 bytes), *(iii)* Message tag (4 bytes), *(iv)* Packet datatype (4 bytes), *(v)* Payload length (4 bytes), and finally *(vi)* The payload data (a variable number of bytes).

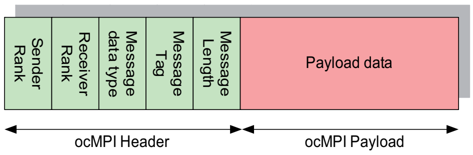

Fig. 5. ocMPI message layout

Later, this ocMPI message will be split in transactions or stream of flits according to the width of the NoC channel.

A. Implemented Functions and Software Stack Configurations

Table I shows the 20 standard MPI functions[1] ported to our NoC-based MPSoC platform.

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()`

TABLE I
SUPPORTED FUNCTIONS IN OUR ocMPI LIBRARY

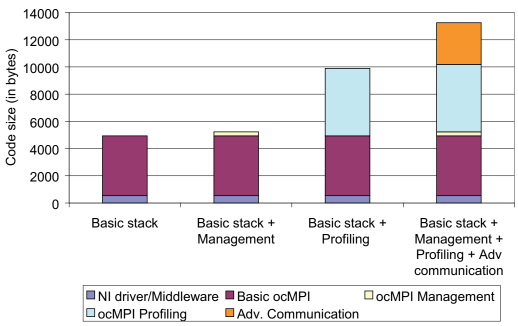

Fig. 6. Different ocMPI software stack configurations

Depending on the application requirements, we can select different software configuration stacks. Figure 6 lists four typical configurations of the ocMPI library and its stripped code memory footprint size in bytes.

[1]To keep reuse and portability of MPI code our ocMPI library follow the standardized definition and prototypes of MPI-2 functions.

B. *Exposing QoS Support on the ocMPI Library*

There are several ways in which the QoS features mentioned above in the NoC backbone can be exploited on top our ocMPI library. The first possibility is to allow the programmer by adding new parameters on the ocMPI API functions in order to trigger prioritized or GT channels during the execution of a particular tasks (or group of tasks). Even if this possibility can be useful to give the knowledgeable programmer the possibility of specifying appropriate prioritization patterns at different program points as needed, we discarded since can compromise the ease of programming since it requires insights on both program behavior and architectural details, and it breaks the standardized prototype of each MPI functions.

In this work, we focus on express QoS in very lightweight manner, rather than invoking manually the QoS middleware API, the idea is to simply annotate the critical tasks at high level by means of using an extended functionality of the ocMPI library. The extension consists on reuse part of the information on the ocMPI packet header (i.e. `ocMPI Tag`) in order to trigger the specific QoS features present at NoC level.

In the above mentioned approach, the ocMPI library is in charge of automatically invoking either low-level QoS or directly middleware AMPI calls without further programmer involvement. Thus, we extended our ocMPI library to embedded appropriate calls to the `setPriority()` and `resetPriority()` and `sendStreamQoS()` and `recvStreamQoS()` middleware functions. This has the effect of establishing prioritized or GT streams between two end-points on the NoC-based system. More specifically, based on the annotation, priorities or GT channels are automatically set/re-set when necessary.

The outcome is a lightweight QoS-aware ocMPI library tailored to many-core on-chip systems since the minimal working subset library (i.e. `ocMPI_Init()`, `ocMPI_Finalize()`, `ocMPI_Comm_size()`, `ocMPI_Comm_rank()`, `ocMPI_Send()`, `ocMPI_Recv()` only takes 4.942 bytes of memory footprint.

VIII. Evaluation of QoS-aware ocMPI Library

In this section we describe the experimental setup that we considered to evaluate the proposed message passing framework with and without runtime QoS on 4, 9, 16 ARM NoC-based MPSoCs.

A. *ocMPI Library Profiling*

To test the effectiveness of ocMPI library, this section presents the scalability of ocMPI synchronization (i.e. `ocMPI_Init()` and `ocMPI_Barrier()` and the profiling of the management functions. The results have been obtained using either `ocMPI_Wtime()` or an external performance monitoring during the execution of parallel programs in different NoC-based MPSoC systems.

Within ocMPI library an initialization phase is required to assign dynamically the rank of each CPU involved in the system. In Figure 7, we report the evolution of the synchronization time in different 2D-Mesh NoC-based MPSoCs.

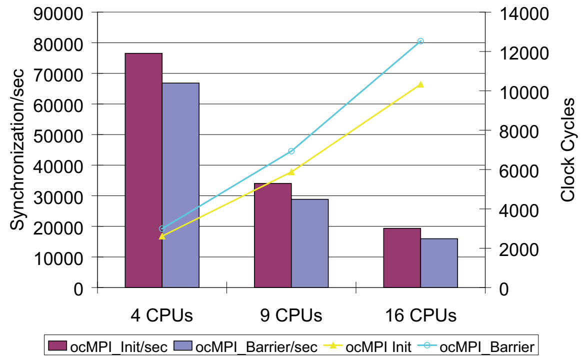

Fig. 7. Profiling of `ocMPI_Init()` and `ocMPI_Barrier()` function

The plot shows the number of `ocMPI_Init()` per second to give and idea which is the reconfiguration time of the system. In our 4 core system, we can perform ≈75.000 reconfigurations in a second, whereas on a large network it decreases until ≈20.000 at 200 MHz. On absolute clock cycles, to setup our 4-core system are necessary 2.613 cycles, since in the 16-core NoC-based MPSoC, `ocMPI_Init()` time raises until 10.331 cycles.

Quite often, MPI programs requires barriers to synchronize all CPUs involved in a parallel workload. As before, in Figure 7 we show the scalability of the synchronization time but now during the execution of barriers. Thus, for instance, in our 9 ARM system, we can perform less than ≈30.000 `ocMPI_Barrier()` per second. However, the time to execute a barrier is of 2.993, 6.935 and 12.527 cycles in our 4, 9 and 16 core systems, which is acceptable since in our measurements we include both, the software overhead and the end-to-end NoC latency[2].

Figure 8 presents the results acquired by performing a monitoring on different management functions demonstrating its minimum execution time.

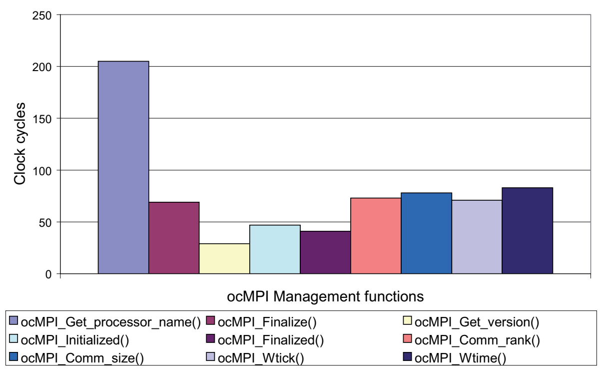

Fig. 8. Profiling of the management ocMPI functions

The results shown that our ocMPI management functions execute really fast, taking ≈75 cycles, with the exception of

[2]Each NoC component, i.e. NIs and switches have been configured to use one cycle of delay.

ocMPI_Get_processor_name() function spends ≈210 clock cycles.

Additionally, unidirectional and ping-pong benchmarks have been evaluated in terms of end-to-end latency using the point-to-point ocMPI library functions. Between adjacent pairs from transferring 1 word from a memory buffer from the source to the destination core, the ocMPI library takes 930 cycles, whereas a ping-pong using pairs of ocMPI_SendRecv() the execution time raise until 2.595 clock cycles. This demonstrates that for short messages, the ocMPI library is not really effective. However, the same benchmark sending 256 words the results are much better in terms of efficiency. Thus, the benchmark takes 8.425 and 15.945 clock cycles, for unidirectional and ping-pong traffic respectively, which represents ≈4x and ≈7x times more time from the previous one, but injecting 256 times more data.

B. Guaranteeing QoS Services on ocMPI Parallel Programs

Typically, MPI and as well ocMPI parallel programs under master-slave distribution does not achieve the desired balance during their execution even by allocating or dividing similar workload on each process on a homogeneous system. Thus, when we map the application onto the hardware resources, many issues could potentially arise.

High contention on the buffer memories and/or on the NoC during message passing specially during narrowcast traffic patterns may cause one (or more) processes to be delayed during its execution. As a consequence, late-sender, early-receiver performance issues can easily arise. Furthermore, critical tasks often requires hard-QoS in order to meet its deadlines.

Our aim is to explore the effectiveness of *(i)* different levels of priorities priority packets and *(ii)* GT channels in solving and mitigating this issue whenever it is required.

One of the techniques considered in our benchmarks is to change the balance of traffic during narrowcasts when typically ocMPI_Gather() function is executed in order to avoid late-senders.

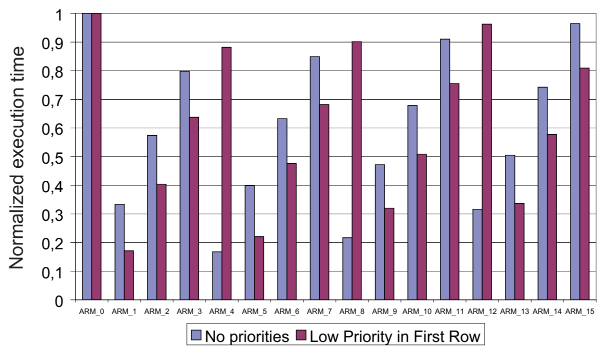

Fig. 9. Effect of priorities during ocMPI gather-narrowcast programs

In Figure 9 we show the normalized execution time to run this benchmark on the 16-core NoC-based MPSoC with and without QoS prioritized traffic. In this experiment, we evaluate the effect to prioritize ocMPI tasks hosted in all the rows far from ARM_0 which is the root to gather the data. In other words, the processes executed from processors on the first row (i.e. ARM_4, ARM_8, ARM_12) will be executed with less priority since they have less critical nature on the system.

Figure 9 shows the parallel program in absence of QoS (only fixed priorities acts on the switch if multiple requests), and same parallel program with a priority assignation according to the annotated tasks. It is easy to notice that the overall execution time on each prioritized process improve by ≈15%, whereas as expected, the processors on the first row due to their low-priority are delayed between ≈65-70%.

Another representative experiment due to its massive traffic generation is to evaluate the system under broadcasting and data gathering. Besides, in this benchmark we also perform computation on each data set before gathering the data. As shown in Figure 10 in absence of priorities, the results are quite similar from previous plot, but increasing the overall normalized execution time of each task because of it is computational part.

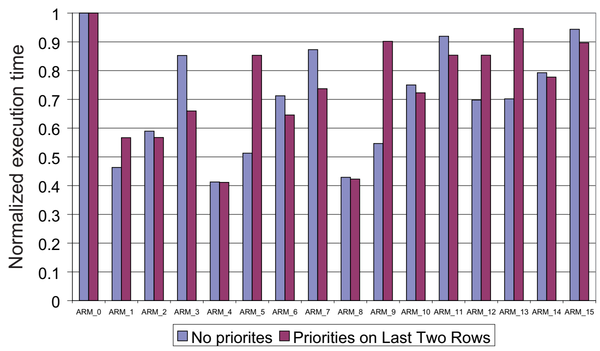

Fig. 10. Effect of priorities on ocMPI_Bcast() and ocMPI_Gather() with an additional computation on data sets

In this kernel/benchmark, we explore the prioritization of half of the system, concretely, the last two bottom rows of the system (i.e. tasks hosted on processors ARM_2, ARM_6, ARM_10, ARM_14 and ARM_3, ARM_7, ARM_11, ARM_15). In figure 10, we can notice different improvement in all processors under prioritization, ranging from ≈3-20%, and the consequent speed-down of the non-prioritized task between ≈10-34% depending on the processor.

Due to the unpredictability of data cache, and due to the reduced computation to communication ratio the improvement results are quite broad and dispare. However, even in this adverse experiment with very unbalanced and non-deterministic traffic, we still demonstrate that using our QoS extension on top of the ocMPI library, we can speedup the prioritized annotated tasks.

Finally, we experiment a completely different approach of leveraging on GT by means of allocate/release channels on ocMPI processes, with really critical requirements (for instance video encoding/decoding). To model this system behavior, we use an ocMPI parallel program which performs this functionality.

We define two scenarios according to the proposed approach: *(i)* none of the ocMPI process has been annotated as GT, and *(ii)* the application annotates a required GT channel from `ARM_14` to `ARM_0` (where video management is performed).

Other processors (i.e. `ARM_1-ARM_13` and `ARM_15`) execute generic workload with the only purpose of generating potentially interfering traffic with the critical transactions issued by `ARM_14` to the `ARM_0`.

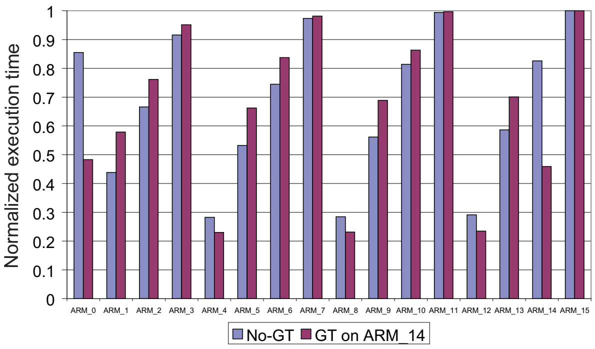

Fig. 11. Effect of GT traffic on ocMPI parallel programs

Looking at Figure 11, it is clear to observe that `ARM_14` is quite delayed when it requires to exchange data by passing a message with `ARM_0`, and therefore, the task has no guarantees to meet its deadlines which potentially can lead to some frames-dropping. As mentioned above, we establish a GT channel to mitigate this potential problem. The results are shown in Figure 11 (see GT on ARM_14 case study). The outcome is a speedup improvement of ≈35% with respect to the previous scenario with no guarantees, even when other processes were potentially generating interfering traffic.

However, it is important to remark that the guarantees are given once the GT channel is open. Thus, looking the plot in Figure 11, it is easy to observe that `ARM_4`, `ARM_8`, `ARM_12` really interferes the execution of `ARM_14`, even finishing their assigned workload. Nevertheless, once the GT channel is open, the end-to-end average latency of the packets is fixed and the throughput is ensured.

Furthermore, under this GT scenario, it is easy to realize that `ARM_0` also go to completion ≈37% before with respect to the non-GT scenario. Thanks to the guarantees imposed on `ARM_14` makes a mitigation of late sender or blocking early receiver performed by `ARM_14` on the ocMPI program.

IX. Conclusion

Handling QoS support on parallel programming has not been tackled properly on the new emerging NoC-based many-core MPSoCs since it is a difficult tasks because of the complex interaction of hardware and software components during the execution of parallel applications. In this work we have proposed a QoS-aware message-passing software stack (i.e. the low-level functions, the middleware and our extended ocMPI library) and we explore performance issues on homogeneous NoC-based MPSoCs.

Our QoS-aware ocMPI was designed as an improved and extended MPI alternative to enable parallel programming through message passing on the novel many-core MPSoCs providing by means of a lightweight middleware API direct access to QoS hardware resources present on the NoC-backbone.

Thus, mixing novel architectures as the emerging NoC-based many-core MPSoCs, together with an adapted well-known message-passing programming model (i.e. ocMPI) produces a reusable and robust way to program highly parallel many-core on-chip systems. In addition, thanks to the layered micro-network stack where NoC-based systems are sustained, we can give some degree to control the available QoS-related NoC services on the parallel programming model in order to boost, balance, and optimize the performance of parallel applications.

A set of representative benchmarks which can potentially lead to unbalancing and memory contention have been simulated on a 16-core NoC-based MPSoC. Our experimental results in this platform using prioritized and GT extensions on the ocMPI library show an average improvement of ≈15% and ≈35% of speedup during the execution of message passing parallel programms, respectively.

We believe that our lightweight QoS-aware software stack (i.e. our middleware API and our extended ocMPI library) is a viable solution to program efficiently high-performance highly parallel NoC-based many-core in a similar way a traditional cluster of supercomputers, giving more freedom on the parallel program in order to speedup applications, balance workload and guaranteed services in critical processes.

Acknowledgements

This work was partly supported by the European ITEA2 ParMA (Parallel programming for Multicore Architectures) Project, the Spanish Ministerio de Industria, Turismo y Comercio project TSI-020400-2009-26 and Ministerio de Ciencia y Innovacion project TEC2008-03835/TEC, the Catalan Government Grant Agency Ref. 2009SGR700 / Ref. 2009BPA00122.

References

[1] S. Borkar, "Thousand Core Chips: A Technology Perspective," in *DAC '07: Proceedings of the 44th annual Design Automation Conference*, 2007, pp. 746–749.

[2] W. J. Dally and B. Towles, "Route Packets, Not Wires: On-Chip Interconnection Networks," in *Proceedings of the 38th Design Automation Conference*, June 2001, pp. 684–689.

[3] L. Benini and G. De Micheli, "Networks on Chips: A new SoC Paradigm," *IEEE Computer*, vol. 35, no. 1, pp. 70 – 78, January 2002.

[4] L. Benini and G. D. Micheli, *Networks on chips: Technology and Tools*. San Francisco, CA, USA: Morgan Kaufmann Publishers, 2006.

[5] S. R. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar, "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] "Tilera Corporation," Tilera, http://www.tilera.com.

[7] "Single-chip Cloud Computing," Intel, http://techresearch.intel.com/articles/Tera-Scale/1826.htm.

[8] "The OpenMP API Specification for Parallel Programming," http://www.openmp.org/.

[9] "The Message Passing Interface (MPI) standard," http://www.mcs.anl.gov/mpi/.

[10] "Open MPI: Open Source High Performance Computing," http://www.open-mpi.org/.

[11] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., 2004.

[12] J. Duato, S. Yalamanchili, and L. Ni, *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers, 2003.

[13] T. Marescaux and H. Corporaal, "Introducing the SuperGT Network-on-Chip; SuperGT QoS: more than just GT," in *Proc. 44th ACM/IEEE Design Automation Conference DAC '07*, Jun. 4–8, 2007, pp. 116–121.

[14] 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, Automation and Test in Europe Conference and Exhibition*, 2003, pp. 350–355.

[15] D. Andreasson and S. Kumar, "On Improving Best-Effort throughput by better utilization of Guaranteed-Throughput channels in an on-chip communication system," in *Proceeding of 22th IEEE Norchip Conference*, 2004.

[16] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny, "QNoC: QoS Architecture and Design Process for Network on Chip," *Journal of System Architecture*, vol. 50, pp. 105–128, 2004.

[17] A. Hansson and K. Goossens, "Trade-offs in the Configuration of a Network on Chip for Multiple Use-Cases," in *NOCS '07: Proceedings of the First International Symposium on Networks-on-Chip*, 2007, pp. 233–242.

[18] S. Murali, P. Meloni, F. Angiolini, D. Atienza, S. Carta, L. Benini, G. De Micheli, and L. Raffo, "Designing Message-Dependent Deadlock Free Networks on Chips for Application-Specific Systems on Chips," in *Proc. IFIP International Conference on Very Large Scale Integration*, Oct. 16–18, 2006, pp. 158–163.

[19] F. Liu and V. Chaudhary, "Extending OpenMP for Heterogeneous Chip Multiprocessors," in *Proc. International Conference on Parallel Processing*, 2003, pp. 161–168.

[20] A. Marongiu and L. Benini, "Efficient OpenMP Support and Extensions for MPSoCs with Explicitly Managed Memory Hierarchy," in *Proc. DATE '09. Design, Automation. Test in Europe Conference. Exhibition*, Apr. 20–24, 2009, pp. 809–814.

[21] B. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, and A. Gatherer, "Implementing OpenMP on a High Performance Embedded Multicore MPSoC," in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, 2009, pp. 1–8.

[22] W.-C. Jeun and S. Ha, "Effective OpenMP Implementation and Translation For Multiprocessor System-On-Chip without Using OS," in *Proc. Asia and South Pacific Design Automation Conference ASP-DAC '07*, Jan. 23–26, 2007, pp. 44–49.

[23] J. Psota and A. Agarwal, "rMPI: Message Passing on Multicore Processors with on-chip Interconnect," *Lecture Notes in Computer Science*, vol. 4917, p. 22, 2008.

[24] M. Saldaña and P. Chow, "TMD-MPI: An MPI Implementation for Multiple Processors Across Multiple FPGAs," in *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, Aug. 2006, pp. 1–6.

[25] J. Joven, O. Font-Bach, D. Castells-Rufas, R. Martinez, L. Teres, and J. Carrabina, "xENoC - An eXperimental Network-On-Chip Environment for Parallel Distributed Computing on NoC-based MPSoC Architectures," in *Proc. 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing PDP 2008*, Feb. 13–15, 2008, pp. 141–148.

[26] J. Joven, D. Castells-Rufas, and J. Carrabina, "An efficient MPI Microtask Communication Stack for NoC-based MPSoC Architectures," in *4th Advanced Computer Architecture and Compilation for Embedded Systems*, 2008.

[27] "The Multicore Association – The Multicore Communication API (MCAPI)," http://www.multicore-association.org/home.php.

[28] A. J. Roy, I. Foster, W. Gropp, B. Toonen, N. Karonis, and V. Sander, "MPICH-GQ: Quality-of-Service for Message Passing Programs," in *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, 2000, p. 19.

[29] R. Y. S. Kawasaki, L. A. H. G. Oliveira, C. R. L. Francê, D. L. Cardoso, M. M. Coutinho, and Ádamo Santana, "Towards the Parallel Computing Based on Quality of Service," *Parallel and Distributed Computing, International Symposium on*, vol. 0, p. 131, 2003.

[30] E. Carara, N. Calazans, and F. Moraes, "Managing QoS Flows at Task Level in NoC-Based MPSoCs," in *Proc. IFIP International Conference on Very Large Scale Integration (VLSI-SoC '09)*, 2009.

[31] D. Bertozzi and L. Benini, "Xpipes: A Network-on-Chip Architecture for Gigascale Systems-on-Chip," vol. 4, no. 2, pp. 18–31, 2004.

[32] 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, Automation and Test in Europe*, 2005, pp. 1188–1193.

[33] M. Millberg, E. Nilsson, R. Thid, S. Kumar, and A. Jantsch, "The Nostrum backbone-a communication protocol stack for Networks on Chip," in *Proc. 17th International Conference on VLSI Design*, 2004, pp. 693–696.

[34] M. Dehyadgari, M. Nickray, A. Afzali-kusha, and Z. Navabi, "A New Protocol Stack Model for Network on Chip," in *Proc. IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, vol. 00, Mar. 2–3, 2006, p. 3pp.

[35] K. Goossens, J. van Meerbergen, A. Peeters, and R. Wielage, "Networks on Silicon: Combining Best-Effort and Guaranteed Services," in *Proc. Design, Automation and Test in Europe Conference and Exhibition*, Mar. 4–8, 2002, pp. 423–425.

[36] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli, "Addressing the System-on-a-Chip Interconnect Woes Through Communication-Based Design," in *Proc. Design Automation Conference*, 2001, pp. 667–672.

[37] H. Zimmermann, "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection," *IEEE Trans. Commun.*, vol. 28, no. 4, pp. 425–432, Apr. 1980.

[38] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, "MPARM: Exploring the Multi-Processor SoC Design Space with SystemC," *The Journal of VLSI Signal Processing*, vol. 41, no. 2, pp. 169–182, September 2005.

[39] T. P. McMahon and A. Skjellum, "eMPI/eMPICH: embedding MPI," in *Proc. Second MPI Developer's Conference*, Jul. 1–2, 1996, pp. 180–184.

[40] D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. De Micheli, "NoC Synthesis Flow for Customized Domain Specific Multiprocessor Systems-on-Chip," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 2, pp. 113–129, Feb. 2005.

[41] S. Murali, P. Meloni, F. Angiolini, D. Atienza, S. Carta, L. Benini, G. De Micheli, and L. Raffo, "Designing Application-Specific Networks on Chips with Floorplan Information," in *ICCAD '06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, 2006, pp. 355–362.

[42] D. Castells-Rufas, J. Joven, S. Risueo, E. Fernandez, and J. Carrabina, "NocMaker: A Cross-Platform Open-Source Design Space Exploration Tool for Networks on Chip," in *3th Workshop on Interconnection Network Architectures: On-Chip, Multi-Chip*, 2009.