Parallel Computing 39 (2013) 549–566

Contents lists available at SciVerse ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

An integrated, programming model-driven framework for NoC–QoS support in cluster-based embedded many-cores

J. Joven [a,*], A. Marongiu [b], F. Angiolini [c], L. Benini [b], G. De Micheli [a]

[a] *Integrated Systems Laboratory (LSI), Ecole Polytechnique Fédérale de Lausanne (EPFL), 1015 Lausanne, Switzerland*
[b] *Dipartimento di Energia Elettrica e Informazione (DEI), University of Bologna (UNIBO), Bologna 40136, Italy*
[c] *iNoCs SaRL, Lausanne, Switzerland*

ARTICLE INFO

Article history:
Received 11 June 2012
Received in revised form 22 April 2013
Accepted 30 June 2013
Available online 6 July 2013

Keywords:
Quality of Service
Network-on-chip
OpenMP
MPI
Cluster-based MPSoC
Parallel computing

ABSTRACT

Embedded SoC designs are embracing the many-core paradigm to deliver the required performance to run an ever-increasing number of applications in parallel. Networks-on-Chip (NoC) are considered as a convenient technology to implement many-core embedded platforms. The complex and non-uniform nature of the traffic flows generated when multiple parallel applications are running simultaneously calls for Quality-of-Service (QoS) extensions in the NoC, but to efficiently exploit similar services it is necessary to expose them to the software in a easy-to-use yet efficient manner. In this paper we present an integrated hardware/software approach for delivering QoS on top of an hybrid OpenMP-MPI parallel programming model. Our experimental results show the effectiveness of our proposal over a broad range of benchmarks and application mappings, demonstrating the ability to manage parallelism under QoS requirements effortlessly from the programming model.

1. Introduction

The ever-increasing complexity of embedded applications requires SoC designs to deliver very high performance while fitting tight power constraints. The current trend to achieve both goals is the *Multi-Processor System on-a-Chip* (MPSoC) design paradigm [1]. The capability of integrating a large number of low-power processing elements within a single chip is foreseen to be a dominant trend nowadays and in the future, as witnessed by many recent experimental platforms such as the Intel SCC [2], Polaris [3,4] and the commercial release of the 50-core Knights Corner processor [5].

In fact, MPSoC designs are rapidly transitioning to the many-core paradigm, where hundreds of cores are integrated on the same chip. To enable system scalability to such a large number of processing elements, another design solution that is being increasingly adopted is core *clusterization*. Several recently proposed many-core architectures leverage tightly-coupled *clusters* as a building block. Examples include the HyperCore Architecture Line (HAL) processors from Plurality [6], ST Microelectronics Platform 2012 [7], or even GPUs like NVIDIA Fermi [8]. In a shared memory paradigm, these designs try to overcome the scalability limitations encountered when increasing the number of processing elements (PEs) that share a unique interconnection and memory system [9] by creating a hierarchical design where PEs are clustered into small/medium-sized subsystems. The small number of PEs enables a high-performance design of the on-cluster interconnection and memory system, while scaling to larger system sizes is achieved by replicating clusters and interconnecting them with a scalable medium.

* Corresponding author. Tel.: +41 (0) 21 693 0916; fax: +41 (0) 21 693 09 09.
E-mail address: jaime.jovenmurillo@epfl.ch (J. Joven).

0167-8191/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.parco.2013.06.002

Indeed, as the number of integrated IP blocks increases, traditional interconnection fabrics show their limitations. Shared buses quickly encounter scalability bottlenecks, whereas the complexity of cross-bars becomes too important for a high number of cores. In this respect, *Networks-on-Chips* (NoCs) [10–12] have proven an effective solution to overcome these limitations and interconnect the many on-chip IP blocks present in a modern MPSoC in a scalable manner.

These emerging NoC-based many-core SoCs provide the potential to run several complex applications concurrently. However, the heterogeneous nature of the concurrent applications will make the application traffic patterns unpredictable, as they can easily conflict within the interconnection and/or memory sub-system. This issue may affect the performance of individual or a group of applications running in parallel in the MPSoCs. To mitigate this issue, the NoC should provide efficient transport and *Quality-of-Service* (QoS) support in order to manage the workloads at runtime while not resorting to resource overprovisioning, as is common in large networks. Hardware-only approaches to the problem are unpractical for two main reasons. First, sophisticated hardware policies are likely to require significant power and area costs. Second, the hardware is agnostic of QoS requirements of individual applications, as it works at the granularity of single transactions.

Many approaches have been proposed in the past to augment NoCs with mechanisms to provide the desired QoS on the individual transaction, but they typically assume that traffic flows can be somehow categorized within a few *classes* with predetermined priorities [13,14]. In short, they do not face the problem of how this information can be forwarded from the programming model to physical packets.

To boost software developers' productivity and to enable efficient exploitation of parallelism, programming models such as OpenMP [15] and MPI [16] have been historically and successfully adopted for large-scale shared-memory and distributed-memory systems, respectively. Programming models provide the necessary abstractions to create and manage parallelism without being too involved in handling low-level details. Thus, in our view, QoS should also be controlled at this level, with appropriate abstractions providing a high-level means to negotiate with the hardware in terms of application prioritization.

In this work, we aim at investigating the effectiveness of an integrated HW–SW approach to the problem. We present a NoC architecture providing QoS facilities and a software stack capable of leveraging such hardware provisions. This architecture is organized in tightly-coupled *clusters*, where a small-medium number of processors communicate through a local interconnect and a local shared memory. Overall, the platform consists of several *clusters*, interconnected with a top-level NoC. We assume a *Distributed Shared Memory model* (DSM) [17], where all processors in the system can directly access all the shared memory modules. However, due to the presence of the hierarchical interconnection system, the access latencies are subject to *Non-Uniform Memory Access* (NUMA) effects, depending on the physical distance of the end points. Similar to traditional NUMA systems from the HPC domain, to efficiently map applications on top of this clustered architecture we consider a hybrid OpenMP-MPI programming model [18,19]. Coarse grained tasks can be mapped onto different clusters, communicating through MPI primitives, while locally to each cluster OpenMP can be used to exploit fine-grained (loop-level) parallelism. The QoS services are exposed to the OpenMP programming model to create parallel threads with a given priority. The outcome is a software stack capable to deliver the desired QoS when multiple applications are running simultaneously in the system.

We provide custom directives to associate the notion of priority to OpenMP constructs. This allows to map the annotated tasks to high-priority threads, which are responsible for forwarding the priority information to the *Network Interfaces* (NI). Such threads are insensitive to the effects of conflicting transactions from non prioritized threads contending for the same interconnection and memory resources, as we make it possible to guarantee that their packets are delivered with higher priority. In fact, simple code annotations convey information about the required QoS for a given high-level task to an underlying runtime environment, which implements the requirement on the available NoC services.

We evaluate our proposal on a cycle-accurate virtual platform modeling the target multi-cluster architectures, on top of which we execute concurrently several applications extracted from two representative embedded benchmark suites, EEMBC [20] and MiBench [21]. Experimental results show an average overall improvement (in terms of execution time speedup) for the annotated tasks of around 70–80%, as well as the clear capability to meet soft deadlines for streaming applications.

This paper is organized as follows. Section 2 describes our cluster-based architectural template and its memory hierarchy. Section 3 describes our vertically integrated HW–SW support to QoS management. Here we provide detailed discussion of QoS support at various levels (NoC, middleware, programming model). Section 4 describes our experimental setup, the selected benchmarks and the results obtained. Section 5 presents related work on QoS support management in MPSoCs at various levels, as well as in parallel programming models. Section 6 concludes the paper and discusses future work directions.

2. Architectural template

In this paper, we consider a cluster-based MPSoC, with a hierarchical NoC interconnection to ensure system scalability. Each cluster includes a configurable number N of RISC32 cores (ARM-based), featuring private L1 instruction and data caches. Processors are interconnected through a $M \times M$ quasi-mesh network, where a central switch is attached to L2 memory banks ($N = (M \times M) - 1$). Fig. 1 shows a zoom of the block diagram of a *cluster* instance with $N = 8$, $M = 3$.

L2 memory consists of explicitly managed SRAM banks (scratchpad memory), which are mapped in the shared address space globally seen by the processors. The software is thus responsible for mapping or moving shared data onto this memory block for inter-processor (intra-cluster) communication. The L2 shared memory also hosts message passing buffers, used for

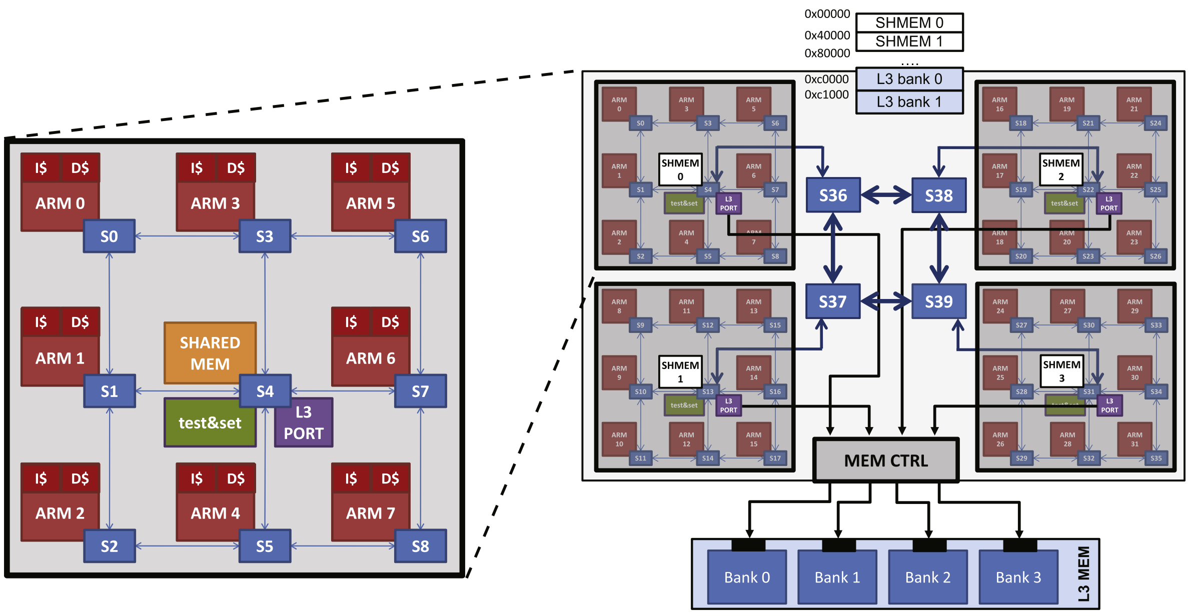


Fig. 1. 32-core ARM clustered NoC-based MPSoC architecture.

inter-cluster communication. Addresses belonging to this shared range cannot be cached. The central switch in each cluster is also connected to a small *test-and-set* memory (hardware semaphores), on top of which we implement synchronization primitives.

Finally, the central switch is also attached to a port towards L3 memory. L3 hosts program code, global and static data as well as program heap and stack storage. Addresses belonging to the L3 memory range can be cached into processors' L1 instruction and data caches. However, if shared data from the L3 memory is allowed in cache, the software is responsible for ensuring data consistency by means of explicit coherency operations (e.g., software flushes), since hardware cache coherency is not supported. However, in this work shared data is explicitly allocated in L2 shared memory. This prevents coherency issues, since only L3 can be cached (program code, read-only and private data), while multiple copies of a same shared datum are disallowed.

The multi-cluster platform that we target in this work consists of a number C of clusters, interconnected through a top-level NoC. Fig. 1 depicts a platform instance with C = 4. Overall, each of the C shared memory banks can be directly accessed from any processor in the system. Indeed, L2 memory is organized as a *Partitioned Global Address Space* (PGAS) system, where each bank is mapped onto a unique address range, visible to every processor. However, due to the physical distance between end-points and the variable number of network hops, accessing a remote cluster memory is subject to strong NUMA effects. L3 ports from each cluster are connected to an on-chip memory controller, where address routing towards L3 memory takes place. L3 memory resides off-chip and consists of four DRAM banks also mapped in the global address space as contiguous memory ranges.

3. Vertically-integrated HW/SW QoS support

In this section we describe our vertically integrated approach to providing programming model-driven QoS. Fig. 2 shows a layered view of this approach.

At the topmost level our framework there are *parallel applications*. Within this layer, we provide two different and complementary programming models, OpenMP [22] and MPI [23]. The reason for these two programming paradigms lies in the nature of the memory subsystem of the target MPSoC, which is, in fact, a DSM platform. Hybrid OpenMP-MPI programming has proven beneficial for DSM systems in the HPC domain [17], and more in general to implement nested (multi-level) parallelism on NUMA systems [18]. Here, coarse-grained (task-level) parallelism can be mapped over different clusters communicating through MPI primitives, whereas fine-grained (loop-level) parallelism can be easily distributed within a cluster with OpenMP.

Besides easing the task of mapping parallelism in an effective manner on the target MPSoC, in our proposal the programming model should also provide the means to specify QoS requirements at the application level. We thus propose simple extensions to OpenMP and MPI programming interfaces – and underlying compiler/runtime support – which allow programmers to negotiate with the hardware in terms of QoS. Clearly, the programmer should not be required to deal with low-level details of HW support for QoS. What we propose is a high-level, abstract notion of priority that can be attached to parallel

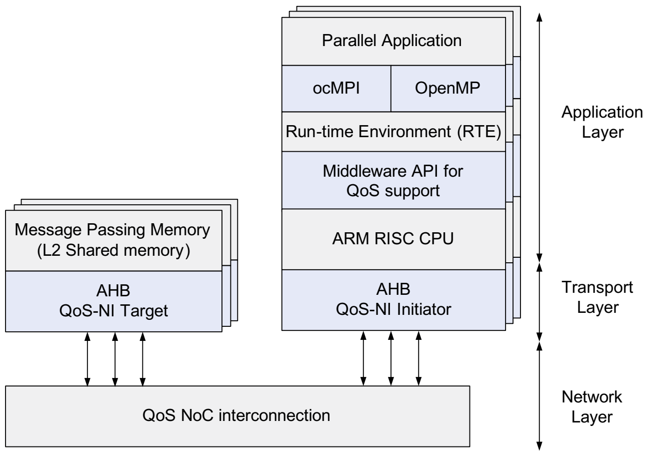


Fig. 2. HW–SW view of our MPSoC platform.

tasks, as exposed to the programming model. Thus, we provide additional clauses to OpenMP directives for parallelism specification, which allow us to forward to the compiler information about the priority of a given task. The compiler is then in charge of transforming this high-level information into appropriate interactions with the *runtime environment* (RTE). The latter is finally responsible for determining the physical priority of traffic flows generated by parallel threads, and properly propagating this information to NIs, which will eventually implement QoS requirements over the NoC. Indeed, from a hardware perspective, these software routines run on top of processors attached to the NIs. Thus, at the *transport layer*, each NI initiator (or target) can inject (or receive) the target transaction using QoS services present in our NoC. In the *network layer* all the application traffic is accordingly routed, arbitrated and allocated following the requested QoS services.

In the following sections we describe in detail the various levels at which QoS is supported in our framework.

3.1. QoS support at the NoC-level

QoS services have been proposed in NoC-based systems aiming to combine *Best-Effort* (BE) and *Guaranteed Throughput* (GT) streams with *Time Division Multiple Access* TDMA [24–26], to distinguish between traffic classes [13,14] and to map multiple use-cases or traffic patterns taking into account the required resource reservation to satisfy latency and bandwidth constraints in worst-case scenarios [27,28]. However, not a lot of effort has been put in studying how to integrate in an effective manner common QoS services at the NoC level into the software stack for emerging many-core SoCs. In this work we emphasize this aspect, providing an integrated approach to exposing QoS control to an OpenMP + MPI-based programming model. Clearly, at the lowest level of the stack, we must provide a streamlined implementation of QoS support in the NoC.

To expose the QoS features at higher levels as NoC services, and to enable runtime reconfiguration in the NoC backbone, we tackle a design at NoC level by extending the basic elements of the ×pipes NoC library [29,30]: *(i)* on the NIs and *(ii)* within the switches. We support QoS using a priority-based scheme (to fit multiple use cases and traffic classes) which classifies several types of traffic according to the constraints imposed on data delivery. This QoS feature is also known as soft QoS because still no guarantees are made for an individual traffic class. However, we also support GT QoS by means of end-to-end channels. This feature is designed by implementing circuit switching on top of our wormhole packet switching NoC architecture.

The aim of designing QoS features in the NIs is to expose them towards the software stack. In NIs, concretely in the NI initiator, the main target is to identify which type of QoS is requested by the processor, accelerator, etc., but also its programmability and reconfiguration at runtime. Furthermore, in the initiator NIs a set of configuration registers, memory-mapped in the address space of the system, are used to program the different levels of priority. These registers can be programmed to assign different levels of priority to each individual packet at runtime. Later, the QoS service level will be embedded in the NoC packet in order to classify different types of priority traffics within the system. Thus, we offer our QoS priority services on top of our BE scheduling – which is based on a *Round Robin* (RR) arbitration scheme – in each switch as in ×pipes [29,30].

Reserving a channel (GT QoS) requires a "fake" transaction (i.e. *open* and *close*), sent from the initiator NI to the target NI, with the purpose of configuring every switch along the path in order to deliver application packets with a guaranteed throughput from the source to the destination. For priority-based QoS service, it is sufficient to write into the configuration register in the initiator NI the priority level required, which will be attached to the head of all the following transactions/packets containing the "real" data. The actual priority scheme implementation is done in each switch depending on the channel request and the priority level embedded in the packet.

Fig. 3 shows a block diagram of a general allocator/arbiter with N ports and M levels of priority. The original ×pipes allocator has been extended with:

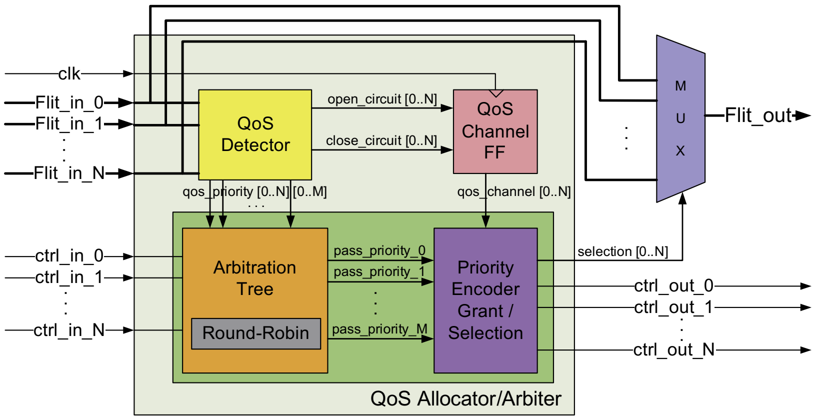


Fig. 3. QoS runtime support on the allocator/arbiter of each switch.

- A QoS tag detector logic block.
- A flip-flop (i.e. `qos_channel`) to store information about whether a GT channel is established or not.
- An extended arbitration tree to enable multiple levels of priority on top of our best-effort service (i.e., round robin).
- A modified grant generator using a priority encoder based on the selected priority level, or the `qos_channel` flip-flop.

In the *QoS detector* block, it is identified which QoS is requested for a specific packet by parsing properly the range of bits embedded on the first flit of the request header packet. If the QoS bits are related to circuit reservation, the circuit is set up or tear down by updating the value stored in the *QoS channel FF* ('1' if established, '0' if not). If the QoS bits store a priority level different from the default value, our extended arbitration tree will override round-robin scheduling decisions. As a consequence, in both cases (priority-based QoS or end-to-end channels) the grant generation will be dictated depending on the QoS service level request through our QoS middleware API within our RTE.

3.2. QoS support at the middleware level

To efficiently exploit our NoC QoS services, we implemented a middleware API to allow the software to interact with the NIs at a higher level of abstraction than register programming. The functions enclosed in this API are summarized in Listing 1.

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

/* Tear down a circuit ,
   unidirectional or full duplex (i.e. write or R/W) */
int ni_close_channel(int MEM_ID, 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);
```

Listing 1. Middleware API for QoS support.

Basically, this API provides functions to easily (i) set-up and tear-down channels for GT QoS (`ni_open_channel` and `ni_close_channel`) and (ii) append priority level information to a transaction between two end-points (`setPriority`) or reset part or all of the previous priority settings (`resetPriority`, `resetPriorities`, `resetAllPriorities`). We optimized the implementation of these functions to be extremely lightweight (only few assembly instructions, executing in few processor cycles). This enables a fine-grain QoS-driven reconfiguration of any type of traffic running in our many-core system (e.g. message-passing, OpenMP barrier synchronization, instruction fetching, shared data access, etc.).

This API is exposed to OpenMP and MPI runtime systems and closely matches the semantics of the programming model, allowing QoS services to be triggered naturally from thread-management or (point-to-point) communication primitives.

3.3. Qos Support at the Programming Model Level

In this section, we show an overview of our customized OpenMP + MPI-based parallel programming models targeted to clustered MPSoCs.

3.3.1. OpenMP

OpenMP is a widely adopted shared memory programming model. It allows to incrementally specify parallelism in sequential C (or C++, or Fortran) code through the insertion of compiler directives. Due to the ease of building parallel programs with OpenMP, several implementations for MPSoCs [31–34] have been proposed.

An OpenMP implementation consists of a code translator and a runtime support library. The framework presented in this paper is based on the GCC compiler, and its OpenMP translator (GOMP). The runtime library leverages a full-custom design, where the support to all OpenMP parallel constructs has been tailored to the target platform hardware. The interested reader can find more details on this OpenMP implementation in [33].

The focus in this paper is on providing extensions to the OpenMP programming interface, compiler and runtime to allow programmers to negotiate in terms of QoS with the underlying hardware. In our proposal this can be done at a very high level by providing a custom `prioritized` clause, that can be coupled with standard OpenMP directives for the creation of parallel thread teams. Listing 2 shows how a parallel region of code can be annotated for high-priority execution.

```
#pragma omp parallel num_threads(4) prioritized
{
  /* Parallel workload */
}
```

Listing 2. OpenMP prioritized clause.

The OpenMP directive translation pass in the compiler outlines the parallel workload into a new function (*parfun*). The `parallel` directive is then replaced with a call to a RTE function, to which the address of *parfun* is passed – as well as shared data addresses, and the number of threads requested by the user – to fork a parallel team. We have extended this mechanism to pass an additional parameter to the RTE, specifying if the `prioritized` clause is present. Before the parallel function is actually executed by each thread, our modified RTE calls the QoS middleware API to appropriately trigger programming of the underlying HW QoS support. This is done in three steps, as shown in Listing 3.

```
/* RTE code executed by prioritized threads */
{
  /* Call QoS API to determine priority settings */
  int PROC_ID = omp_get_proc_id ();
  int MEM_ID = omp_get_mem_id (<shared data address>)
  int level = get_pri_level (PROC_ID, MEM_ID);

  /* Call QoS API to set desired priority level */
  setPriority (PROC_ID, MEM_ID, level);

  /* EXECUTE PARALLEL FUNCTION */
  **parfun (*shdata)

  /* Call QoS API to reset priority level */
  resetPriority (PROC_ID, MEM_ID);
}
```

Listing 3. RTE code for prioritized threads.

First, the calling thread queries the RTE to retrieve end points for its traffic flow. More precisely, it resolves the physical ID of the processor on which it is hosted, and the physical ID of the memory on which its data are placed. This is done by invoking the custom RTE functions `omp_get_proc_id` and `omp_get_mem_id`. Second, a call to the RTE function `omp_get_pri_level` is inserted to determine the physical priority level for the calling thread. Third, the `setPriority` QoS middleware API is finally invoked to set the desired QoS level. After the execution of the parallel region previous QoS settings are reset through a call to `resetPriority`.

Being the RTE aware of the architectural topology (physical position of the cores in the NoC, and how they are interconnected with memories), once end-points for communication flows have been determined it is capable of setting appropriate QoS levels for each thread. It has to be pointed out that the `prioritized` clause only allows to logically outline high-priority *activities* in the system. However, when associated with a `parallel` directive, this *activity* gets parallelized over multiple threads, which should be guaranteed same QoS level.

For example, if the *activity* is a parallel loop, maintaining an identical QoS for all the processors involved may lead to load imbalance, due to different physical paths to which corresponding transactions are subject. It is thus necessary to exploit the

Table 1
Additional OpenMP runtime functions to manage QoS.

Function prototype	Brief description
`int omp_get_proc_id (void)`	Retrieves the physical ID of the processor on which the calling thread is running
`int omp_get_mem_id (int addr)`	Retrieves the physical ID of the memory on which datum at address *addr* is placed
`int omp_get_pri_level (void)`	Returns the physical priority level assigned to the calling thread

knowledge of the architectural topology to assign different threads in a parallel team distinct priorities, with the final goal of achieving the QoS requirement specified for the *activity* as a whole.

Within the `omp_get_pri_level` function each prioritized thread annotates in a specific data structure in the RTE the ID of its hosting processor and the ID of the memory containing the targeted data. If the end-point (i.e., the target memory) is the same for all the threads, it is possible to implement policies which assign them different priority levels depending on the physical path that each of them has to traverse, and taking into account the effect of the routing algorithm on the network. At the moment, we do not assign distinct priorities to threads belonging to a same team, because our QoS services are currently implemented on top of a wormhole network, where uninterruptible transactions from higher level threads would always be serviced first, thus leading to imbalance. However, if a budget scheduler is used (e.g., weighted fair queuing [14]), smarter policies can be implemented in `omp_get_pri_level`, where priorities are dynamically set at a finer granularity.

The `prioritized` clause can also be associated with OpenMP worksharing constructs, to prioritize only a subset of the parallel threads. For example, OpenMP v2.5 allows to model task parallelism with the `sections` directive.[1] Each section identifies a distinct parallel task, and we may want to annotate only a few of them as being high-priority ones. In Listing 4 we trigger parallel execution of tasks `task_A`, `task_B` and `task_C`. Let us suppose that `task_B` requires more bandwidth to satisfy certain constraints (e.g. soft deadlines). We can grant high-priority transactions with the custom `prioritized` clause as shown in Listing 4.

```
#pragma omp parallel sections
{
  #pragma omp section
  task_A();

  #pragma omp section prioritized
  task_B();

  #pragma omp section channel(data)
  task_C(data);
}
```

Listing 4. Prioritized and channel clause in OpenMP programs.

If a task has hard constraints and requires an exclusive communication channel towards a given memory, we allow the programmer to set such a communication link through the use of the custom `channel` (⟨*var*⟩) clause as shown in Listing 4 for `task_C`. When the `channel` clause is encountered, the compiler inserts a call to `omp_get_mem_id` to retrieve the ID of the memory where datum `var` is hosted, then establishes an exclusive communication channel to this memory by invoking the `ni_open_channel` QoS middleware function.

Table 1 summarizes the custom functions that we added to the OpenMP runtime to manage QoS services.

3.3.2. On-chip Message Passing Library

Message passing is a wide-spread parallel programming model, which in the form of a standard API library [35–37] can be ported and optimized on many different platforms. In this section, we show an overview of our proprietary, MPI-compliant *on-chip Message Passing Library* (ocMPI).

The ocMPI library has been designed using a bottom-up approach as proposed in [38], taking as a reference the open source Open MPI project [39]. It does not rely on any OS, and rather than relying on the TCP/IP protocol (as the standard MPI-2 library), it uses a customized transport layer to enable message-passing on top of many-core SoC hardware. Fig. 4 shows our MPI adaptation for embedded systems.

Each ocMPI message has the following format: *(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). The ocMPI message packets are extremely slim to avoid big overhead for small- and medium-sized messages.

The synchronization protocol to exchange data relies on a rendezvous protocol supported by means of flags/semaphores, which have been mapped onto the local shared memory on the cluster. These flags are polled by each sender and receiver to synchronize with the message-passing memory (L2 shared memories) to exchange ocMPI messages in our cluster-based

[1] Since specification version 3.0, OpenMP has introduced *tasks* as a more flexible abstraction for task-level parallelism. While we are currently working on efficient support for tasking in embedded MPSoCs, in this paper we only comply to the OpenMP specification version 2.5.

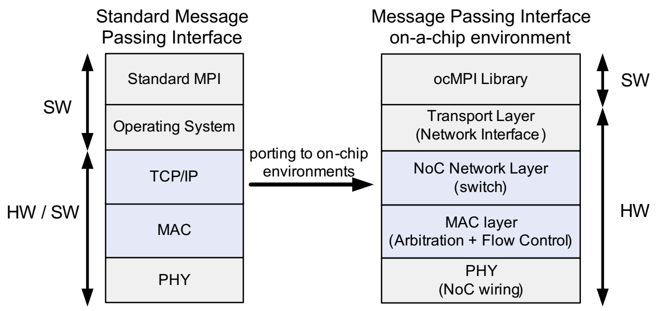

Standard Message Passing Interface
Standard MPI
Operating System
TCP/IP
MAC
PHY
SW
HW / SW
porting to on-chip environments
Message Passing Interface on-a-chip environment
ocMPI Library
Transport Layer (Network Interface)
NoC Network Layer (switch)
MAC layer (Arbitration + Flow Control)
PHY (NoC wiring)
SW
HW

Fig. 4. Layered view of our ocMPI software stack.

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 synchronization because either the sender and the receiver unblock, or neither does.

ocMPI is implemented on top of 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 programmers, who only need to deal with the standard `ocMPI_Send()` and `ocMPI_Recv()` functions. Our ocMPI library does not require any intermediate copies and user-space buffers, since the ocMPI message is stored directly on the message-passing memory. This leads to a very fast inter-process communication by means of a remote-write, local-read transfer hiding the read latency on the system.

The actual implementation of ocMPI includes 23 standard MPI functions (see Table 2). In this work, we have extended 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 MPSoC. Later, the library will automatically invoke the corresponding QoS middleware function(s) presented in Listing 1 to enable prioritized traffic or end-to-end circuits during the execution of message-passing parallel programs. To ensure reusability and portability of legacy MPI code, our ocMPI library even with QoS support 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()`.

3.3.3. *QoS in OpenMP + MPI programs*

OpenMP and MPI are typically used in a synergistic manner in traditional cluster-based systems [18]. Nested (multi-level) parallelism is typically used in these systems, where coarse-grained parallel tasks are assigned to different clusters and communicate through MPI primitives, while additional inner-level loop parallelism is distributed among cores within the cluster through OpenMP directives. Let us consider as an example the last two stages of JPEG decoding – *dequantization* and *IDCT*.

Fig. 5 shows how these can be parallelizes as a two-stage pipeline over time, iterating over all the *macro-blocks* (MB) composing the image. However, since macro-blocks can be independently processed, we can create coarse-grained tasks (i.e., pipeline stages) containing each as many macro-blocks as cores in a cluster, to keep the whole system fully busy with parallel processing. For example, considering the multi-cluster MPSoC platform previously introduced in Fig. 1, we would create tasks with eight macro-blocks each.

Listing 5 shows an example of how such an application should be written with our OpenMP + ocMPI approach.

Table 2
Supported functions in our 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 and collective communication	`ocMPI_Broadcast ()`, `ocMPI_Barrier ()` `ocMPI_Gather ()`, `ocMPI_Scatter ()`, `ocMPI_Reduce ()`, `ocMPI_Scan ()`, `ocMPI_Exscan ()`, `ocMPI_Allgather ()`, `ocMPI_Allreduce ()`, `ocMPI_Alltoall ()`

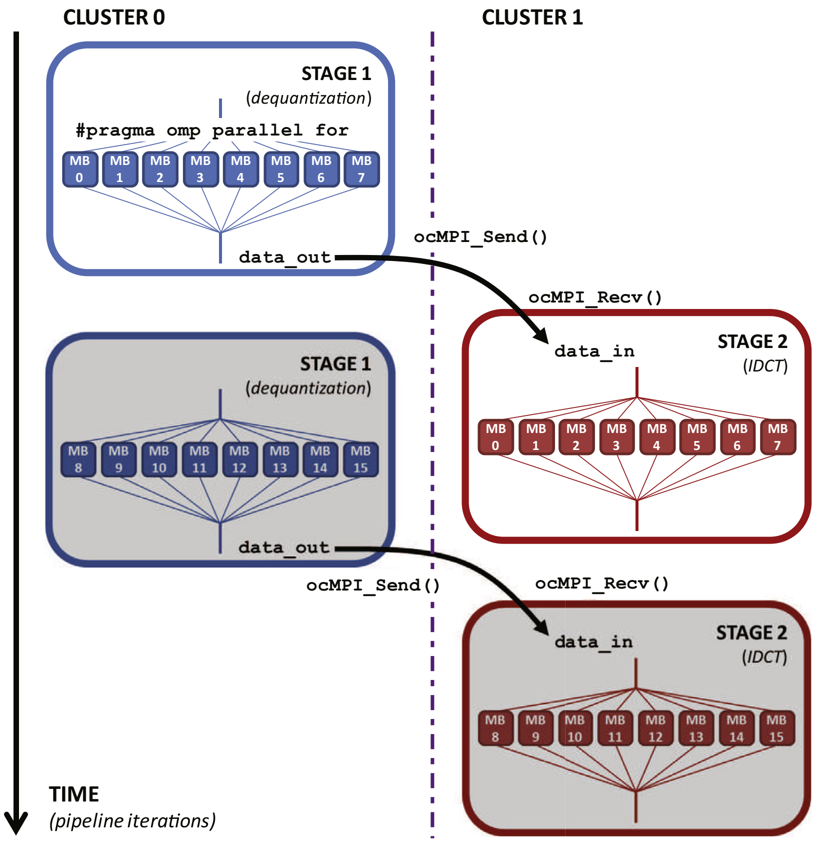


Fig. 5. Task graph for JPEG parallelized with OpenMP + MPI.

```
int input[];     /* The input image  */
int output[];    /* The output image */

#pragma omp parallel num_threads(2) prioritized
for (i=0; i<MBLOCKS; i+=8)
#pragma omp sections nowait
{
  #pragma omp section
  {
    int data_out[8*MB_SIZE];

    /* Dequantization loop */
    #pragma omp parallel for num_threads(8)
    for (j=i; j<i+8; j++)
      dequantization(&input[j], &data_out[j]);

    ocMPI_Send((void*)&data_out, ...);
  }

  #pragma omp section
  {
    int data_in[8*MB_SIZE];

    ocMPI_Recv((void*)&data_in, ...);

    /* IDCT loop */
    #pragma omp parallel for num_threads(8)
    for (j=i; j<i+8; j++)
      idct(&data_in[j], &output[j]);
  }
}
```

Listing 5. Prioritized OpenMP-ocMPI application code.

When designing our framework we faced a fundamental problem: How to design custom extensions for QoS control in presence of two distinct programming models? In our view allowing to independently control QoS from both programming models would make it tougher for the programmer to write and debug its QoS-based programs. Thus, in our proposal, QoS actions are taken from within OpenMP, using the previously described extensions. MPI primitives found within an OpenMP

```
/* Code executed by prioritized threads */
for (i=0; i<MBLOCKS; i+=8)
{
  int id = omp_get_proc_id ();
  if (id == ARM_0)
  {
    int data_out[8*MB_SIZE];

    dequantization(&input[i], &data_out[i]);
    _barrier (cluster_0);
    int mem_id = omp_get_mem_id (&data_out);
    ni_open_channel(mem_id, 0);
    ocMPI_Send((void*)&data_out, dataSize, ocMPI_BYTE, ARM_8, tagValue, ocMPI_COMM_WORLD);
    ni_close_channel(mem_id, 0);
  }
  else if (id == ARM_8)
  {
    int data_in[8*MB_SIZE];
    int mem_id = omp_get_mem_id (&data_in);
    ni_open_channel(mem_id, 0);
    ocMPI_Recv((void*)&data_in, dataSize, ocMPI_BYTE, ARM_0, tagValue, ocMPI_COMM_WORLD, &status);
    ni_close_channel(mem_id, 0);
    idct(&data_in[i], &output[i]);
    _barrier (cluster_1);
  }
  else if (<belongs to cluster 0 (id)>)
  {
      dequantization(&input[i+id], &data_out[i+id]);
      _barrier (cluster_0);
  }
  else
  {
      idct(&data_in[i+(id%8)], &output[i+(id%8)]);
      _barrier (cluster_1);
  }
}
```

Listing 6. OpenMP-ocMPI translated code.

program which uses prioritization are automatically treated by our compiler so as to maintain their behavior consistent with the decision specified through the OpenMP constructs.

In the JPEG example we are modeling a pipeline with two stages, which should be overall executed as a high-priority *activity* in our system. We thus create two prioritized threads with the top-level `parallel` directive coupled to our `prioritized` clause. The outermost loop sweeps through all the macro-blocks (pipeline iterations) with a stride of 8 (number of cores per cluster).

Later, we create two different tasks (OpenMP sections), each executing a stage of the pipeline (*dequantization* and *IDCT*), and further parallelize it among the cores in each cluster. We use MPI primitives to synchronize producer (first task/stage) and consumer (second task/stage).

Since the two coarse-grained tasks are mapped onto two different clusters, the sender will have to direct its message to a remote shared memory bank. In this case, the high-priority requirement attached to the OpenMP parallel team may be ignored by message passing primitives, which are agnostic about this information. To prevent this, the OpenMP compiler extends the high-priority semantics of the coarse-grained tasks to the `ocMPI_Send()` and `ocMPI_Recv()` operations by enhancing them with the prioritization mechanisms discussed in Section 3.3.2.

In particular, in the prototype implementation presented here we wrap `ocMPI_Send()` and `ocMPI_Recv()` with calls to `ni_open_channel` and `ni_close_channel`. It is important to remark that other implementations based on the use of priorities are possible. The code excerpt in Listing 6 shows how this example gets translated by the compiler.

4. Experimental results

To validate our QoS framework we model a 32-core, 4-cluster instance of the platform described in Section 2 within MPARM, a SystemC-based full-system simulator [40]. We integrate all the layered QoS-support facilities, as well as our software stack components, in this platform. The NoC backbone has been designed using ×pipescompiler and ×pipes library [29,30]. We designed two separate NoCs, one for request and another for responses, with XY routing in order to avoid routing and message-level deadlocks [41]. In Table 3 we provide detailed architectural parameter for the setup of our hardware platform.

To evaluate the proposed framework, we conduct different sets of experiments aimed at assessing the effectiveness of our programming model extensions both locally, in a single-cluster (i.e. inter-cluster), and globally, in the entire multi-cluster MPSoC (i.e. intra-cluster). Specifically, a first set of experiments focuses on a single cluster as a computational domain, and our custom OpenMP directives are used to test the behavior of parallelized applications in terms of QoS under different application mappings. A second set of experiments leverages a mix of OpenMP and MPI to partition and distribute applications over the whole multi-cluster platform. For both inter-cluster and intra-cluster scenarios, we studied the effect of our QoS control over the application workloads.

Table 3
Overview of the architectural parameters.

Parameter	Configuration
Topology	Dual 2-D quasi-mesh
Routing algorithm	XY routing
Arbitration policy	Round-robin
Switching scheme	Wormhole packet switching
Flow control	On/off
Flit size	48 bits
Queue scheme	Input–output queue
Queue size	2-flits input buffer
	6-flits output buffer
QoS services	Soft-QoS using up to 8-priority levels
	End-to-end virtual channels
Processor core	32-bit ARM RIS
	4 KB D-Cache/4 KB I-Cache
L2 Shared memory	256 KBytes/bank (10 cycles)
L3 Main memory	64 MBytes/bank (+100 cycles)

Table 4
Benchmarks.

Benchmark	Description	Source	Subkernels	Data Parallel
djpeg	JPEG Decoder	EEMBC	Huffman DC	NO
			Huffman AC	NO
			dequantization	YES
			IDCT	YES
rgbhpg01	Image filter	EEMBC		YES
rotate01	Image filter	EEMBC		YES
CRC32	Cyclic redundancy check	MiBench		NO
adpcm	Adaptive pulse code modulation	MiBench		NO

Table 4 summarizes the benchmarks considered in our experiments. Results related to the single- and multi-cluster explorations are presented in Sections 4.1 and 4.2 respectively. These benchmarks were extracted from two of the most representative program suites in the embedded domain, namely EEMBC [20] and MiBench [21]. We selected our test programs from different domains (image coding and filtering, security, telecommunication) to build a heterogeneous workload representative of a real system.

Some of the benchmarks contain data parallelism (i.e., parallel loops), which we outlined using our enhanced OpenMP programming model. In particular, two subkernels from the *djpeg* benchmark, namely *dequantization* and *IDCT*, are data-parallel routines. Similarly, *rotate01* and *rgbhpg01* are largely data-parallel after some transformations (privatization of counter variables which create false loop-carried dependencies). The remaining two subkernels from *djpeg*, namely *Huffman DC* and *Huffman AC* decoding, are not parallelizable, similar to benchmarks *adpcm* and *CRC32*.

4.1. Single-cluster results

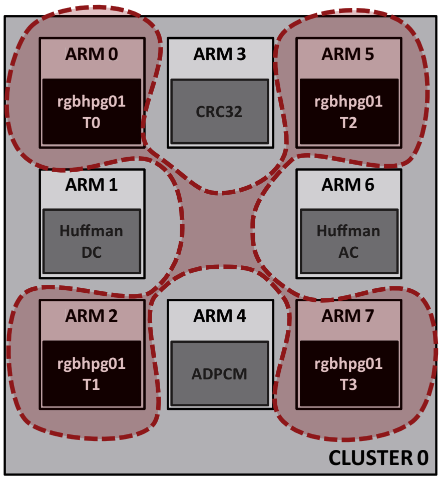
In our experiments with a single cluster, we use sequential benchmarks to generate background, interfering traffic flows to our prioritized traffic, which is originated from parallel benchmarks. More precisely, we allocate 4 processors in the cluster system to permanently host these sequential benchmarks, and we use the remaining 4 to host each of the 4 data-parallelized benchmarks in turn.

Fig. 6(a) shows an example of the described single-cluster application mapping, considering *rgbhpg01* as a parallel benchmark. An analogous mapping is considered for *rotate01*, *dequantization* and *IDCT*.

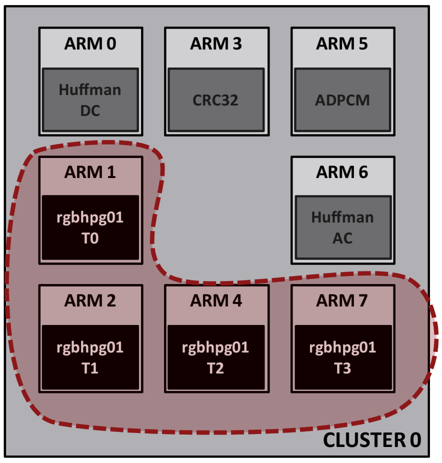
In this mapping all the parallel threads are hosted on physical processors which have a symmetrical layout in terms of communication towards the memory (which is kept in a central position). The traffic generated by these threads will thus have near-identical latency, which is a desirable feature to ensure load balancing.

However, we want to study the effects of a different mapping, where threads belonging to a same parallel team are hosted on processors having non-homogeneous paths to memory in terms of NoC hops. To evaluate this case and assess the benefits of our QoS framework, we also consider the application mapping shown in Fig. 6(b).

Results for this experiment are shown in Figs. 7 and 8 for mapping 1 and mapping 2, respectively. In both figures and subfigures, the bars labeled “*no QoS*” represent the execution cycles taken when no prioritization is applied, and all the transactions generated by the various applications are serviced through standard arbitration policies in the NoC. The bars labeled

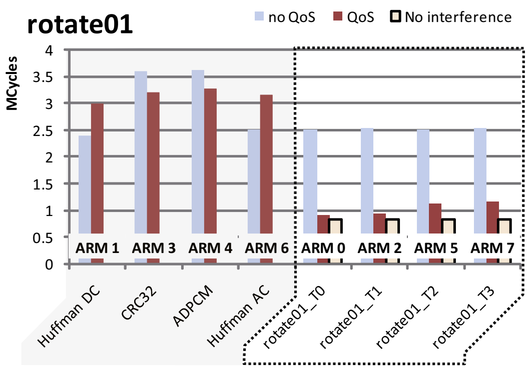


(a) Application mapping 1

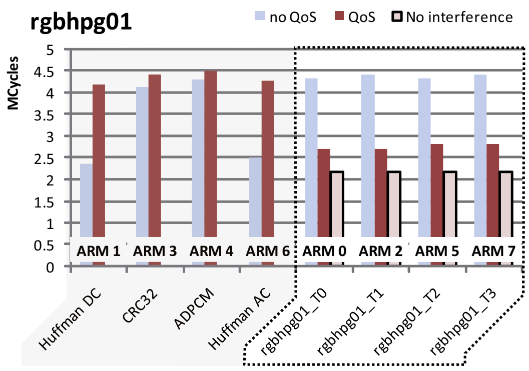


(b) Application mapping 2

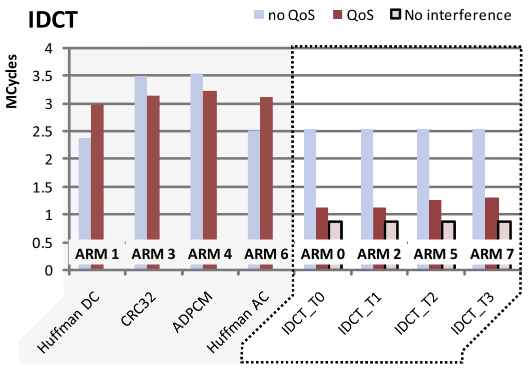
Fig. 6. Single-cluster application mappings.



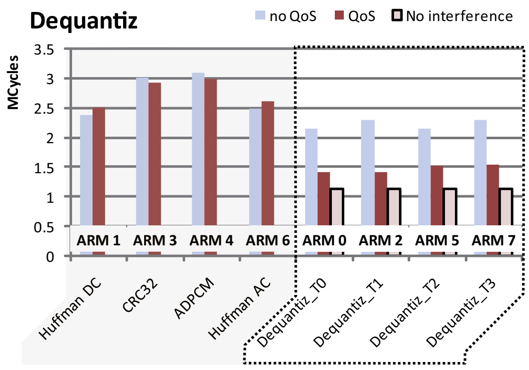
(a) Rotation



(b) RGB



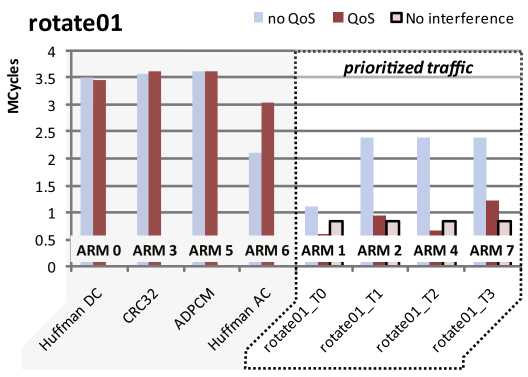
(c) IDCT



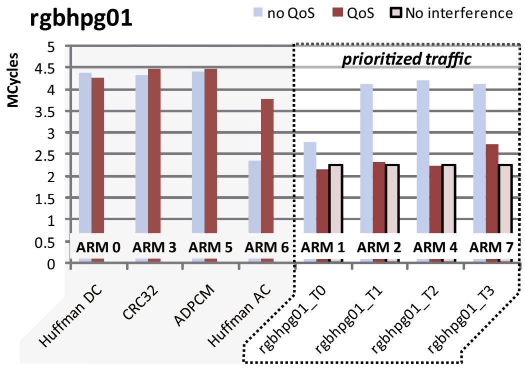
(d) Dequantization

Fig. 7. Single-cluster benchmarks using application mapping 1.

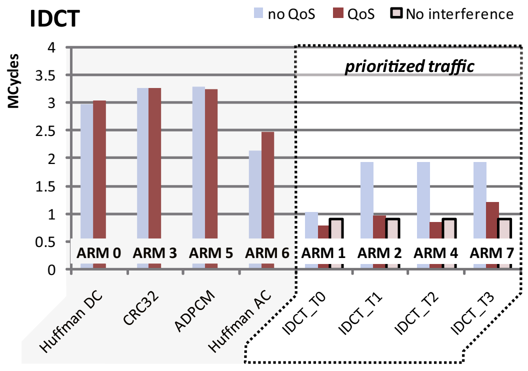
"*QoS*" represent the execution cycles taken when prioritization is applied to the parallel application considered in the specific test. Finally, the bars labeled "*No interference*" represent the duration of parallel threads in absence of interference. Specifically, the duration of the parallel benchmark is measured when its threads run in isolation in the platform. Since OpenMP



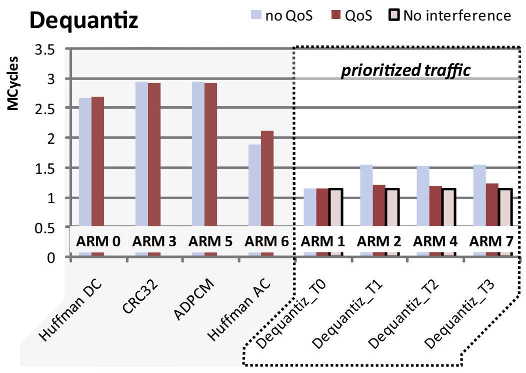
(a) Rotation



(b) RGB



(c) IDCT



(d) Dequantization

Fig. 8. Single-cluster benchmarks using application mapping 2.

implies a barrier synchronization step at the end of each parallel region, we plot the time taken by the slowest thread (the others are forced to wait for it to finish). Each of the benchmarks is also marked (at the foot of the bars) with the ID of the processor onto which it is mapped.

For application mapping 1 (see Fig. 7), it is possible to see that parallel threads have balanced execution times even in absence of prioritization, due to the symmetric communication paths and the round-robin allocator. However, the parallel block takes respectively 203.27%, 102.81%, 193.46% and 101.17% longer for *rotate01*, *rgbhpg01*, *IDCT* and *dequantization*, compared to the "*No interference*" use case.

It is possible to observe that annotating the same parallel regions with our custom directives for prioritization leads to a much smaller performance loss of 38.74%, 29.30%, 50.46% and 34.87% respectively, with an overall average improvement of 115.72%. As opposed to the prioritized parallel regions, the tasks mapped in processors ARM_1 (i.e., Huffman DC) and ARM_6 (i.e., Huffman AC) suffer a performance degradation on average of 32.29%.

On the other side, mapping 2 (see Fig. 8) provides an interesting scenario where parallel threads are unbalanced due to architectural non-homogeneity in the communication paths. Focusing on the "*QoS*" bars, it can be seen that thread 0 is always faster than the rest of the parallel threads, since it is mapped to the processor ARM_1, which is only 1-hop away from the shared memory. Thread 2 is also mapped to a processor (i.e., ARM_4) which has the same distance from the shared memory, however its communication path is shared with processors ARM_2 and ARM_7. In this case, the effect of interfering applications increases the duration of parallel benchmarks by 182.87%, 87.95%, 112.52 and 35.00% with respect to "*No interference*". Applying our QoS policies reduces this performance degradation to only 46.47%, 21.92%, 32.17%, 7.90% respectively. In this mapping 2, the overall average improvement for all the tested applications is on average 104.58%. Moreover, the unbalanced behavior of the parallel applications is reduced on average by 71.77%. In contrast, the task mapped in ARM_6 (i.e., Huffman AC) experience a performance degradation on average of 33.17%, whereas the other tasks maintain similar performance.

4.2. Multi-cluster results

In this section we describe our results with the entire platform consisting of multiple clusters. The setup for the experiments changes slightly.

First, as a priority application, we consider a variant of the *djpeg* benchmark where we repeat continuously the four decoding stages within a loop, to mimic the behavior of a motion jpeg decoder (*MJPEG*).

Second, we consider multi-level parallelization for the prioritized application, to assess the effectiveness of the hybrid MPI + OpenMP approach. More specifically, we adopt pipeline parallelism where each of the four stages of the decoder is assigned to a different cluster. Communication through the stages is implemented through our QoS-augmented MPI primitives. Then, we use OpenMP to extract data parallelism from the last two stages (*Dequantization* and *IDCT*). By profiling the serial duration of each single stage, we determine the number of parallel threads for parallel regions so as to roughly equalize the length of the stages. This allows us to minimize loss of parallelism due to unbalanced pipeline stages.

Third, we want to assess the capabilities of our framework to guarantee QoS on more than a single application. To this aim, we consider two instances of the MJPEG decoder (hereafter called *MJPEG_A* and *MJPEG_B*), both of which can be annotated as high-priority. The rest of the benchmarks, parallel or sequential, is used as interfering traffic. These benchmarks are also repeated several times to ensure persistent interfering activity with the high-priority programs. Moreover, we also insert fake inter-cluster communication (unnecessary send-receive pairs) to test the effectiveness of QoS support at this level. The overall application mapping for this experiment is shown in Fig. 9.

Results for this experiment are shown in Fig. 10. On the *X*-axis three different QoS policies, namely *no QoS*, *QoS on MJPEG_A* and *QoS on MJPEG_A* & *B*, whereas on the *Y*-axis we report parallelization speedups, normalized to the speedup obtained in the *Ideal* case (i.e., when the target application runs in isolation in the platform, without interference).

It is possible to see that in presence of interference, i.e. when no QoS is set, all the applications are largely delayed with respect to the ideal case, but this effect is particularly pronounced for the MJPEG programs, since their physical mapping spans multiple clusters, and the sources of delay are more numerous. Indeed, the speed achieved with these programs is only ≈20% of the ideal value. Using end-to-end channels within the MPI communication between the stages of *MJPEG_A* and annotating parallel stages with the custom `prioritized` OpenMP clause brings its execution time almost at 90% of the ideal value. Our QoS support facilities show good capacities also at managing two concurrent and independent high-priority applications. When prioritizing both *MJPEG_A* and *MJPEG_B*, their execution time reaches 83% and 87% of the ideal value, respectively.

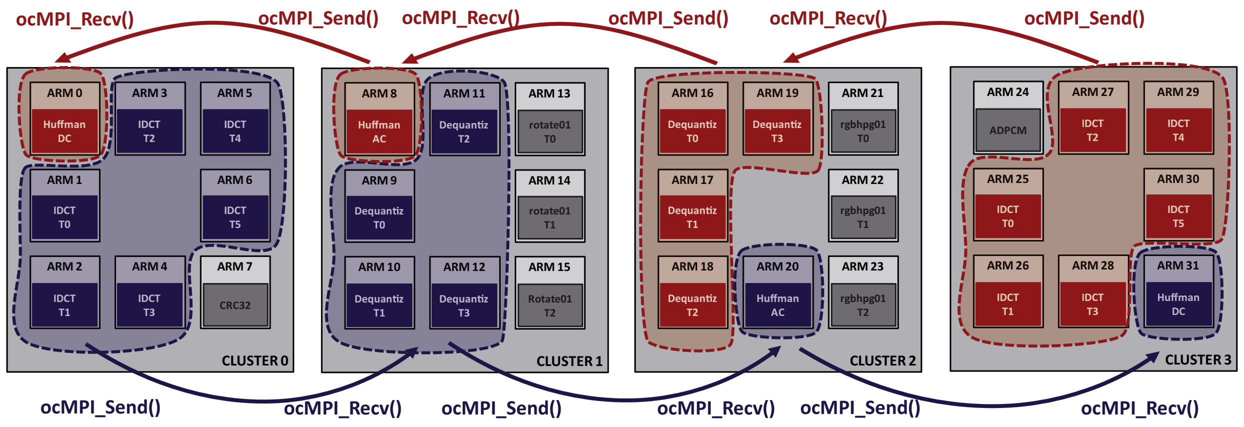


Fig. 9. Multi-cluster application mapping.

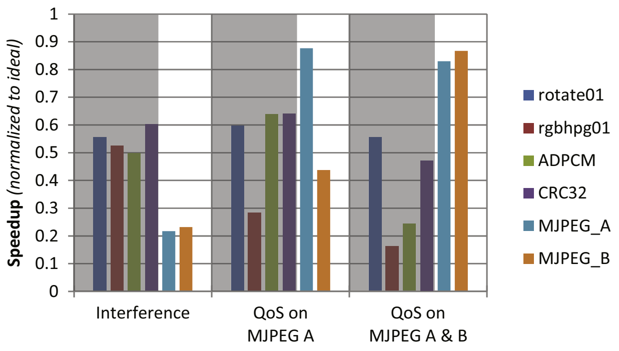
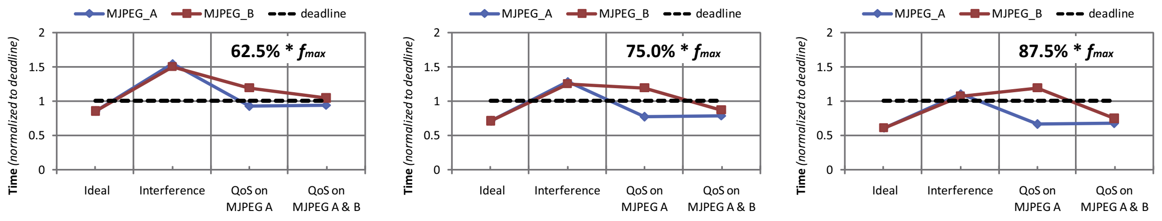


Fig. 10. Effect of prioritization for MJPEG_A and MJPEG_B on multi-cluster parallelization.



(a) 62.5% * fmax (b) 75.0% * fmax (c) 87.5% * fmax

Fig. 11. Effect of priorities on two instances of MJPEG for different processor frequencies.

As a further test of the capabilities of our approach to support multiple high-priority applications, we consider a specific case where a soft real-time constraint has to be met by two applications co-existing in the system. Thus, we consider a deadline for *MJPEG_A* and *MJPEG_B* for decoding a single frame, dictated by the 25 frames-per second rate constraint. This deadline is clearly a function of the system frequency, which may vary because of architectural reasons (e.g., different processing elements) or due to frequency scaling being applied.

In Fig. 11 we show the result of an experiment where the system frequency is scaled to 87.5%, 75% and 62.5% of the maximum. In our particular platform configuration these values constitute an operating range where QoS can be helpful to meet the deadlines. In the plots the deadline is drawn as a dashed black line. The curves with rhomboidal and square markers in Fig. 11 describe the behavior of *MJPEG_A* and *MJPEG_B*, respectively, when different QoS policies are applied. The leftmost point represents the *Ideal* case where the target application is running in isolation in the platform, without interference. The second point shows the behavior of the same application when all the interfering traffic is applied (all applications running at the same time in the platform). The third point refers to the case where *MJPEG_A* is marked for high-priority execution. The fourth point depicts results for a scenario where both *MJPEG_A* and *MJPEG_B* are marked for high-priority execution. It is possible to observe that within the considered frequency range our QoS support is capable of guaranteeing soft real-time constraints even in presence of heavy interfering traffic.

5. Related work

Different QoS support approaches have been widely studied for NoC-based systems considering the area-performance trade-offs. A first approach, as presented in [25,26], is to combine best-effort services with guaranteed throughput by means of TDMA, where channels are globally scheduled in time slots. In [24], a similar work is presented, introducing a new QoS level, SuperGT, where time slots are allocated to provide guarantees, but in case of traffic peaks, the traffic potentially can be transmitted using best-effort resources to inject data during free slots.

A second approach, as introduced in [27,28], is based on mapping at design-time multiple use-cases or traffic patterns taking into account the required resource reservation to satisfy latency and bandwidth constraints in worst-case scenarios. On the other hand, an alternative analytical models based on network calculus [42] have been also proposed to provide real-time latency and bandwidth bounds for different flows in best-effort NoCs without the need to include any special hardware support for QoS [43,44]. Nevertheless, these approach assume statistical traffic injection and regulation characterized for worst-case scenarios.

A third approach, as presented in [13], is to provide soft-QoS by means of distinguishing between several class of services. A similar idea is reviewed in [14], but rather than focusing on individual components, such as the NoC, it presents a coordinated management of multiple QoS-aware shared resources (i.e. cache, NoC and memory).

Providing hardware QoS support that efficiently deals with numerous traffic scenarios often results in over-provisioned resources at design-time. As a consequence, for the emerging NoC-based many-core systems, run-time QoS services must be provided together with streamline software specially to accommodate a broad range of many different simultaneous applications.

In this work, rather than focus on the development of advanced QoS HW support, we extended the ×pipes library to support at runtime the state-of-the-art techniques, such as soft-QoS using a priority-based scheme and hard-QoS by means of simple end-to-end channels. Our main focus in this paper is on maximizing the benefits of these existing NoC QoS techniques by (i) implementing them in the most lightweight and efficient manner for a resource-constrained MPSoC across the software stack (ii) exposing these runtime QoS services to the programming model by means of simple extensions.

On the other hand, the increasing complexity of the emerging high-performance many-core embedded SoCs has recently stimulated a lot of research to provide parallel programming models and runtime environments aimed at facilitating application development and enabling efficient exploitation of heterogeneous hardware.

Both OpenMP [31–33,45] and MPI [46–50] have received a lot of attention in the embedded domain, due to the fact that they are well-known and provide easy-to-use and familiar means of specifying parallelism. Regarding MPI, in the industry Intel released its own MPI-based library called RCCE [51,52] which provides message-passing on top of the SCC [2], and IBM explore the possibility to use both, OpenMP [53] and MPI-based [54] on top of the Cell processor.

Most of the optimizations proposed in these works are aimed at providing extremely lightweight support for the programming model services, as the target embedded hardware is always constrained in terms of the allowed memory and power consumption, the lack of native operating system services etc. This often leads to limited support of the programming model services, or to non-compliant implementations.

Furthermore, none of these works explicitly focuses on QoS support in the parallel application. How to combine runtime hardware QoS services with well-know parallel programming models remains an open problem.

In this paper, we believe that keeping the software stack agnostic about QoS issues and, demanding all the required support actions to the hardware does not afford an efficient solution to the problem. Therefore, in this work, we focus on a streamlined support to enable QoS, and how to effectively exploit it, when it is required in place by the application and programming model.

To the best of our knowledge, the approach detailed in this paper is one of the first attempts to tackle this issue (originated from our previous individual research works [50,55]) and the work presented in [56]. However, in this work we developed an integrated HW–SW approach to QoS support by tightly coupling our runtime QoS services with our combined OpenMP + MPI parallel programming model.

The conducted experiments and the results show the capabilities of our integrated framework to deliver the desired QoS services directly from the parallelized application by means of simple programming model directives avoiding to deal with very low-level NoC service programming.

6. Conclusion

Embedded software and parallel programming models are becoming more and more the centerpiece to exploit highly parallel architectures. In emerging NoC-based MPSoCs, multiple concurrent software stacks and applications will run in parallel, with markedly different timing constraints. All these parallel applications will share the interconnection and they will access the memory sub-system in an unpredictable and non-uniform way. As a consequence, depending on the application mapping, the frequency scaling mode of the system, the degree of congestion in the network and memory hierarchy may affect the overall system performance causing application delays or even missing application deadlines.

To mitigate these effects, it is mandatory to expose and handle QoS services on top of well-know parallel programming models, so as to manage QoS requirements for parallel workloads from the software stack. This issue has not be tackled properly in the emerging NoC-based many-core systems. However, we believe that this is a key challenge to boost embedded software development, giving at the same time the possibility to deliver certain QoS guarantees to a particular task or group of tasks.

In this paper, we provide QoS services on top of an OpenMP + MPI programming model, abstracting all the complexities of the hardware support, and raising their control up to the application level by means of custom programming features and associated compiler and runtime support. Concretely, we support priority-based QoS and guaranteed services by means of end-to-end channels during parallel computing. To achieve the maximum efficiency, all software components have been tightly-coupled with the hardware platform, enabling fine-grain runtime reconfiguration of QoS services (i.e., priority-based or guaranteed services). Parallel applications can in fact trigger these services in just few clock cycles, with a minimal involvement from the software developer.

Our experimental results demonstrate that a proper integration of the software components with the underlying hardware platform enables efficient QoS for each task and thread, helping to boost performance, balance the workload and meet application requirements under different application mappings and operation modes. Results for experiments using our QoS extensions at the cluster-level show on average up to 70–80% overall improvement, and a reduction in some cases of the performance degradation caused by unbalanced workload due to NUMA effects. When multiple parallel applications are running and generating interfering traffic flows in the whole multi-cluster system, we show that our QoS-enhanced OpenMP-MPI parallelization can meet QoS requirements for two concurrent high-priority applications (i.e. MJPEG). Indeed, our support allows to reach 83% to 87% of their ideal speedup (i.e., one in absence of interference), and to meet soft deadlines even in presence of scaled core frequency.

Acknowledgments

This work was partly supported by Catalan Government Grant Agency (Ref. 2009BPA00122), ERC – European Research Council (Ref. 2009-adG-246810), projects FP7 VIRTICAL (288574) and JTI SMECY (ARTEMIS-2009-1-100230), funded by the European Community.

References

[1] A. Jerraya, W. Wolf, Multiprocessor Systems-on-Chips, Morgan Kaufman, Elsevier, 2005.
[2] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, J. Zook, TILE64 – Processor: A 64-Core SoC with mesh interconnect, in: IEEE International Solid-State Circuits Conference, ISSCC 2008, Digest of Technical Papers, 2008, pp. 88–598.
[3] 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, S. Borkar, An 80-tile sub-100-W TeraFLOPS processor in 65-nm CMOS, IEEE J. Solid-State Circuits 43 (1) (2008) 29–41.

[4] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, T. Mattson, A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS, in: IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010, pp. 108–109.
[5] Intel Corporation, Many Integrated Core (MIC) Architecture (<http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html>, 2011).
[6] Plurality Ltd., The HyperCore Processor,, <www.plurality.com/hypercore.html>.
[7] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, D. Dutoit, Platform 2012, A many-core computing accelerator for embedded SoCs: performance evaluation of visual analytics applications, in: Proceedings of the 49th Annual Design Automation Conference, DAC '12, 2012, pp. 1137–1142.
[8] NVIDIA, NVIDIA's Next Generation CUDA™ Compute Architecture: Fermi™, <http://www.nvidia.com/object/fermi_architecture.html>, 2009.
[9] Tilera Corporation, <http://www.tilera.com>.
[10] W.J. Dally, B. Towles, Route Packets, Not wires: on-chip interconnection networks, in: Proceedings of the 38th Design Automation Conference, 2001, pp. 684–689.
[11] L. Benini, G. De Micheli, Networks on chips: a new SoC paradigm, IEEE Comput. 35 (1) (2002) 70–78.
[12] L. Benini, G.D. Micheli, Networks on Chips: Technology and Tools, Morgan Kaufmann Publishers, San Francisco, CA, USA, 2006.
[13] E. Bolotin, I. Cidon, R. Ginosar, A. Kolodny, QNoC: QoS architecture and design process for network on chip, J. Syst. Arch. 50 (2004) 105–128.
[14] B. Li, L. Zhao, R. Iyer, L.-S. Peh, M. Leddige, M. Espig, S.E. Lee, D. Newell, CoQoS: coordinating QoS-aware shared resources in NoC-based SoCs, J. Parallel Distrib. Comput. 71 (2011) 700–713.
[15] L. Dagum, R. Menon, OpenMP: an industry standard API for shared-memory programming, IEEE Comput. Sci. Eng. 5 (1) (1998) 46–55.
[16] W. Gropp, E. Lusk, N. Doss, A. Skjellum, A high-performance, portable implementation of the MPI message passing interface standard, Parallel Comput. 22 (6) (1996) 789–828.
[17] H. Wong, A. Rendell, The Design of MPI Based Distributed Shared Memory Systems to Support OpenMP on Clusters, in: IEEE International Conference on Cluster Computing, 2007, 231–240.
[18] R. Rabenseifner, G. Hager, G. Jost, Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes, in: 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, 2009, pp. 427–436.
[19] H. Jin, D. Jespersen, P. Mehrotra, R. Biswas, L. Huang, B. Chapman, High performance computing using MPI and OpenMP on multi-core parallel systems, Parallel Comput. 37 (9) (2011) 562–575.
[20] EEMBC, EEMBC, The Embedded Microprocessor Benchmark Consortium, <http://www.eembc.org>.
[21] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, R.B. Brown, MiBench: a free, commercially representative embedded benchmark suite, in: IEEE International Workshop on Workload Characterization, 2001, pp. 3–14.
[22] The OpenMP API Specification for Parallel Programming, <http://www.openmp.org/>.
[23] The Message Passing Interface (MPI) standard, <http://www.mcs.anl.gov/mpi/>.
[24] T. Marescaux, H. Corporaal, Introducing the SuperGT network-on-chip; SuperGT QoS: more than just GT, in: Proceedings of the 44th ACM/IEEE Design Automation Conference DAC '07, 2007, pp. 116–121.
[25] E. Rijpkema, K.G.W. Goossens, A. Radulescu, J. Dielissen, J. van Meerbergen, P. Wielage, E. Waterlander, Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip, in: Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, 2003, pp. 350–355.
[26] A. Hansson, K. Goossens, M. Bekooij, J. Huisken, CoMPSoC: a template for composable and predictable multi-processor system on chips, ACM Trans. Des. Autom. Electron. Syst. 14 (1) (2009) 1–24.
[27] S. Murali, M. Coenen, A. Radulescu, K. Goossens, G. De Micheli, A methodology for mapping multiple use-cases onto networks on chips, in: Proceedings of the Design, Automation and Test in Europe DATE '06, vol. 1, 2006, pp. 1–6.
[28] A. Hansson, 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.
[29] D. Bertozzi, L. Benini, Xpipes: a network-on-chip architecture for gigascale systems-on-chip, IEEE Circuits Syst. Mag. 4 (2) (2004) 18–31.
[30] S. Stergiou, F. Angiolini, S. Carta, L. Raffo, D. Bertozzi, G. De Micheli, ×pipes Lite: a synthesis oriented design library for networks on chips, in: Proceedings of the Design, Automation and Test in Europe, 2005, pp. 1188–1193.
[31] B. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, 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.
[32] W.-C. Jeun, S. Ha, Effective OpenMP implementation and translation for multiprocessor system-on-chip without using OS, in: Proceedings of the Asia and South Pacific Design Automation Conference ASP-DAC '07, 2007, pp. 44–49.
[33] A. Marongiu, P. Burgio, L. Benini, Supporting OpenMP on a multi-cluster embedded MPSoC, Microprocessors Microsyst. 35 (8) (2011) 668–682.
[34] A. Marongiu, L. Benini, An OpenMP compiler for efficient use of distributed scratchpad memory in MPSoCs, IEEE Trans. Comput. 61 (2) (2012) 222–236.
[35] The Multicore Association – The Multicore Communication API (MCAPI), <http://www.multicore-association.org/home.php>.
[36] D.W. Walker, J.J. Dongarra, MPI: a standard message passing interface, Supercomputer 12 (1996) 56–68.
[37] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, The MIT Press, 1999.
[38] T.P. McMahon, A. Skjellum, eMPI/eMPICH: embedding MPI, in: Proceedings of Second MPI Developer's Conference, 1996, pp. 180–184.
[39] Open MPI: Open Source High Performance Computing, <http://www.open-mpi.org/>.
[40] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, M. Olivieri, MPARM: exploring the multi-processor SoC Design space with systemC, J. VLSI Signal Process. 41 (2) (2005) 169–182.
[41] A. Hansson, K. Goossens, A. Rădulescu, Avoiding message-dependent deadlock in network-based systems on chip, VLSI Design. 2007 (2007) 10pp.
[42] J.-Y. Le Boudec, P. Thiran, Network Calculus: A Theory of Deterministic Queuing Systems for the Internet, Springer-Verlag, Berlin, Heidelberg, 2001.
[43] M. Bakhouya, S. Suboh, J. Gaber, T. El-Ghazawi, Analytical modeling and evaluation of on-chip interconnects using network calculus, in: Third ACM/IEEE International Symposium on Networks-on-Chip, NoCS 2009, 2009, pp. 74–79.
[44] D. Rahmati, S. Murali, L. Benini, F. Angiolini, G. De Micheli, H. Sarbazi-Azad, A method for calculating hard QoS guarantees for networks-on-chip, in: IEEE/ACM International Conference on, Computer-Aided Design (ICCAD'09), 2009, pp. 579–586.
[45] F. Liu, V. Chaudhary, Extending OpenMP for heterogeneous chip multiprocessors, in: Proceedings of International Conference on Parallel Processing, 2003, pp. 161–168.
[46] J. Psota, A. Agarwal, rMPI: message passing on multicore processors with on-chip interconnect, Lecture Notes Comput. Sci. 4917 (2008) 22.
[47] M. Saldaña, P. Chow, TMD-MPI: an MPI implementation for multiple processors across multiple FPGAs, in: International Conference on Field Programmable Logic and Applications, FPL '06, 2006, 1–6.
[48] D. Göhringer, M. Hübner, L. Hugot-Derville, J. Becker, Message passing interface support for the runtime adaptive multi-processor system-on-chip RAMPSoC, in: International Conference on Embedded Computer Systems (SAMOS), 2010, pp. 357–364.
[49] N. Saint-Jean, P. Benoit, G. Sassatelli, L. Torres, M. Robert, MPI-based adaptive task migration support on the HS-scale system, VLSI, IEEE Computer Society Annual Symposium on 0 (2008) 105–110.
[50] J. Joven, F. Angiolini, D. Castells-Rufas, G. De Micheli, J. Carrabina, QoS-ocMPI: QoS-aware on-chip Message Passing Library for NoC-based Many-Core MPSoCs, in: Second Workshop on Programming Models for Emerging Architectures (PMEA'10), 2010.

[51] T. Mattson, R. Van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, S. Dighe, The 48-core SCC processor: the programmer's view, in: International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2010, pp. 1–11.
[52] R.F. van der Wijngaart, T.G. Mattson, W. Haas, Light-weight communications on Intel's single-chip cloud computer processor, SIGOPS Oper. Syst. Rev. 45 (2011) 73–83.
[53] K. O'Brien, K. O'Brien, Z. Sura, T. Chen, T. Zhang, Supporting OpenMP on cell, Int. J. Parallel Program. 36 (3) (2008) 289–311.
[54] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, T. Nakatani, MPI microtask for programming the cell broadband engine processor, IBM Syst. J. 45 (1) (2006) 85–102.
[55] J. Joven, A. Marongiu, F. Angiolini, L. Benini, G. De Micheli, Exploring programming model-driven QoS support for NoC-based platforms, in: Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, 2010, pp. 65–74.
[56] E. Carara, G.M. Almeida, G. Sassatelli, F.G. Moraes, Achieving composability in NoC-based MPSoCs through QoS management at software level, in: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011, pp. 407–412.