Exploring Programming Model-driven QoS Support for NoC-based Platforms

Jaume Joven
LSI-EPFL
Bâtiment INF
1015 Lausanne, Switzerland
jaime.jovenmurillo@epfl.ch

Andrea Marongiu
DEIS-University of Bologna
Viale Risorgimento 2
40136 Bologna, Italy
a.marongiu@unibo.it

Federico Angiolini
iNoCs SaRL
Route de Chavannes 27D
1007 Lausanne, Switzerland
angiolini@inocs.com

Luca Benini
DEIS-University of Bologna
Viale Risorgimento 2
40136 Bologna, Italy
luca.benini@unibo.it

Giovanni De Micheli
LSI-EPFL
Bâtiment INF
1015 Lausanne, Switzerland
giovanni.demicheli@epfl.ch

ABSTRACT

Networks-on-Chip (NoCs) are being increasingly considered as a central enabling technology to communication-centric designs as more and more IP blocks are integrated on the same SoC. Embedded applications, in turn, are becoming extremely sophisticated, and often require guaranteed levels of service and performance. The complex and non-uniform nature of network traffic generated by parallel applications running on a large number of possibly heterogeneous IPs makes a strong case for providing Quality of Service (QoS) support for traffic streams over the NoC infrastructure.

In this paper we consider an integrated hardware/software approach for delivering QoS at the application level. We designed NoC hardware support, low-level middleware and APIs which enable QoS control at the application level. Furthermore, we identify a set of programming abstractions useful to associate the notion of priority to each running task in the system. An initial implementation of this programming model is also presented, which leverages a set of extensions to a MPSoC-specific OpenMP compiler and runtime environment.

Categories and Subject Descriptors

C.2.1 [**COMPUTER-COMMUNICATION NETWORKS**]: Network Architecture and Design—*Packet-switching networks*; D.3.0 [**PROGRAMMING LANGUAGES**]: General

General Terms

Design, Performance

CODES+ISSS'10, October 24–29, 2010, Scottsdale, Arizona, USA.

Keywords

Quality-of-Service, Networks-on-Chip, Programming Models, Runtime Environments

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 [23]. The capability of integrating a large number of low-power processing elements within a single chip is foreseen to be a dominant trend in the future.

As the number of integrated IP blocks increases, traditional interconnection mediums show their limitations. Shared buses quickly encounter scalability bottlenecks, whereas the complexity of cross-bars becomes too important for a high number of cores.

Networks-on-Chip (NoCs) [5, 6, 14] have been proposed to overcome these limitations and interconnect the many on-chip IP blocks present in a modern MPSoC design in a scalable manner. MPSoCs can be either homogeneous or heterogeneous, featuring a mix of general purpose processors, memories, DSPs, multimedia accelerators, etc. Especially in the latter case, fitting power budgets requires careful optimization. The NoC should therefore provide efficient transport services while not resorting to resource overprovisioning, as is common in large networks.

Hardware-only approaches to the problem are likely to require significant power and area cost. Moreover, increased hardware and software design complexity in turn increases the complexity of on-chip traffic patterns, possibly making their static characterization impractical or overly conservative.

In this paper we aim at investigating the effectiveness of an integrated HW-SW approach to the problem. The NoC supports QoS provisioning by supporting best-effort, high-throughput or low latency traffic classes, which can be controlled by the software stack. Specifically, several levels of Quality-of-Service (QoS) are exposed at the application level, thus allowing for different operating regimes as the use cases alternate at runtime.

Programming models for application development could

be enriched with high-level constructs for QoS management. Simple annotations in the program may convey information about the traffic pattern at a given task to an underlying runtime environment, which should exploit programmers feedback to efficiently take advantage of the available NoC services. This approach would allow application designers to carefully allocate communication resources in a prioritized manner, while requiring minimal area and power cost for the implementation of QoS-related features in the NoC.

As a concrete embodiment of the described approach, we present a NoC architecture providing QoS facilities and a software stack capable of leveraging such hardware provisions. To maximize the applicability of this approach, we integrate our work within the OpenMP programming model.

OpenMP is a widespread interface to exploit shared memory programming models. Its API consists of a collection of compiler directives that allow to easily specify parallel execution within a sequential program. The standard currently supports the C, C++ and Fortran languages. Mainly due to its appealing ease of use, OpenMP boasts a number of MPSoC-specific implementations [11, 25, 28, 33].

Our work is based on one such implementation [28], on top of which we integrated our extensions to support QoS-related features. We choose to adopt a layered approach that incrementally abstracts away hardware-specific details of QoS support at the NoC level, and vertically exposes high-level constructs to the OpenMP runtime environment.

We provide custom directives to associate the notion of priority to OpenMP constructs. This allows to create a privileged environment for the execution of annotated tasks, which are mapped to high-priority threads. Such thread teams 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.

Experimental results on a set of representative OpenMP kernels show that under different traffic patterns and thread allocation schemes, the use of prioritized transactions boosts the overall execution time by up to 66%.

This paper is organized as follows. Section 2 presents related work on QoS support management in NoC-based systems. Section 3 describes the implementation details of QoS support at the NoC level. Section 4 describes the exposure of hardware QoS support to the parallel programming model (i.e. the OpenMP runtime environment). Section 5 shows an overview of the developed NoC-based platform. Section 6 describes our experimental setup, selected benchmarks and the results obtained. Section 7 concludes the paper and provides future work directions.

2. RELATED WORK

In traditional networks [12, 17] 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 run-time; a proper mix of hardware facilities and software-controlled management policies is vital to achieving efficient results. Previous work to provide different levels of QoS at the hardware level is presented in [9] where a complete NoC framework named QNoC is designed to split the NoC traffic in different QoS classes (i.e. signaling for inter-module control signals, real-time for delay-constrained bit streams, RD/WR that model short data access and block-transfers for handling large data bursts), but more work would be needed to expose these facilities across the software stack.

The work presented in [27] tries to combine Best-Effort (BE) and Guaranteed-Throughput (GT) streams by handling them with a time-slot allocation mechanism in the NoC switches. A clockless NoC that mixes BE and guaranteed services (GS) are presented in [8, 22]. In addition, in [1, 37] the authors shows to combine GT and BE in a efficient way, taking into account the required resource reservations for worst-case scenarios. Finally, a more generic approach following the idea presented in [9] is presented in [19, 31] by showing a methodology to map multiple use-cases or traffic patterns onto a NoC architecture, satisfying the constraints of each use-case.

Very recently, the authors in [10] introduce a method to manage QoS at the task level on a NoC-based MPSoC, and in [20, 21] a detailed model on how to enable application-performance guarantees by applying dataflow graphs, as well as aelite, a NoC backbone based with composable and predictable services are presented.

In our work, we face a complete HW-SW co-design complementing previous works [10, 24, 26] designing explicitly the interface between hardware and software components, and extending them, in the sense that we are going further by exposing the QoS support on top of an OpenMP parallel programming model, and the runtime library specially targeted to NoC-based MPSoC platforms.

3. QOS SUPPORT AT NOC LEVEL

The NoC research community is focusing on a variety of design issues concerning heterogeneous NoC design, such as those presented in [35] (e.g. channel width, buffer size, application or IP mapping, routing algorithms, etc), as well as the design of Network Interfaces (NIs) for different bus-based protocols (e.g. AMBA/AXI [2, 3], OCP-IP [34]), adding virtual channels [13, 30] to improve the latency, designing efficient application-specific or custom topologies automatically [32], and bringing these designs towards different VLSI technologies, such as 65nm [36]. Not a lot of effort has been spent to study how to support QoS making it available at runtime across the software stack from application to packets. Most NoC designs actually offer BE facilities only, and the communication traffic cannot exploit any guarantees on delay or throughput.

In this work, we are going to offer runtime QoS on top of the fixed best-effort (BE) scheduling based on *Fixed Priorities* or *Round Robin (RR)* arbitration schemes on the channels in each switch as in ×pipes [7, 39].

We support soft QoS using differentiated services based on a priority scheme (to fit multiple use cases) 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 individual flows.

To guarantee services in a "hard" way, we also enable the capability of reserving channels to ensure end-to-end bandwidth or latency for certain traffic between specific IP cores on the NoC-based system. Thus, this work tries to support

NoC services by combining BE and GT, as well as differentiated services, but without impacting a lot on the design of hardware components (in terms of area and consequently in power consumption).

Thus, we build NoC switches and network interfaces to allow the capability of reserving channels or bandwidth for certain traffic or arbitrating packets with different priorities set by the application or the programming model.

To disclose the QoS features at higher levels as NoC services, and to enable runtime reconfiguration of the NoC backbone, we tackle a design at NoC level by extending the basic elements of ×pipes NoC library: *(i)* on the Network Interfaces (NIs) and *(ii)* within the switches.

3.1 Runtime QoS Support at Network Interface (NI) Level

The aim of designing QoS features in the NIs is to expose them towards higher levels of abstraction. 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.

In 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 on each individual packet at runtime.

A 4-bit QoS field embedded in the NoC packet (see encoding in Listing 1) lets us design up to 8-level priority schemes to classify different types of traffic within the system. Thus, the level 0 has lowest priority, and level 7 is the most prioritized in the NoC architecture. To program a priority packet two phases must be performed; first, a NI programs the transaction to the specific configuration register with the priority level required, and afterwards, one or more real transactions or packets with the actual data. The actual priority scheme implementation is done in each switch depending on the channel request and the priority level embedded in the packet.

Listing 1: QoS encoding on header packets

```
// Priority levels
#define ENC_QOS_PACKET_0          4'b0000
#define ENC_QOS_PACKET_1          4'b0001
#define ENC_QOS_PACKET_2          4'b0010
#define ENC_QOS_PACKET_3          4'b0011
#define ENC_QOS_PACKET_4          4'b0100
#define ENC_QOS_PACKET_5          4'b0101
#define ENC_QOS_PACKET_6          4'b0110
#define ENC_QOS_PACKET_7          4'b0111

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

On the other side, to trigger the opening and closing of circuits in order to block/release channels for a certain period of time, we choose to inject "fake" transactions in the beginning and in the end, in order to notify the switches. This mechanism is based on the emulation of a circuit switching on top of a wormhole packet switching scheme.

The normal behaviour to open/close an exclusive transmission/reception guaranteeing QoS is done in three phases:

1. Open the circuit (unidirectional or bidirectional "fake" transaction).
2. Perform a normal stream of transactions or packets over the NoC under GT conditions.
3. Close the circuit (unidirectional or bidirectional "fake" transaction).

The activation overhead to trigger the QoS is null since the QoS field is directly embedded in the packet header as a 4-bit field. The software overhead to program the NI is based on the memory-mapped store transaction, i.e. few clock cycles depending on the NoC configuration.

3.2 Handling QoS Traffic at Switch Level

From QoS point of view, the main important module on the switch is the arbiter/allocator, which can be configured on a BE basis for fixed-priority or round-robin policies. In this work, we extend the BE behaviour of the allocator/arbiter block, enabling up to 8-level of priority, and guaranteeing GT, through the establishment and releasing of exclusive circuits in case it is necessary. This is enough to deal with the different use cases present in MPSoC.

In Figure 1 is shown a block diagram of a a general allocator/arbiter of N ports and M levels of priority. The original ×pipes allocator has been extended with:

- A QoS tag detector logic.
- A flip-flop (i.e. `qos_channel`) to store whether a GT connection is established or not.
- An extended arbitration tree to enable multiple levels of priority.
- A modified grant generator using a priority encoder based on the selected priority level, or the `qos_channel` flip-flop.

In the allocator (see Figure 1), in *QoS detector* is identified which QoS is requested for a specific packet or flow. The QoS field is parsed by selecting the proper range of bits of the first flit of the request header packet. As long as the QoS bits are related to circuit reservation, and depending on its encoding (i.e. `ENC_QOS_OPEN_CHANNEL` or `ENC_QOS_CLOSE_CHANNEL`), a set up or tear down of the circuit is done by updating the value on *QoS channel FF* ('1' if is established and '0' if not). The value of this register is used in the arbitration process to grant the next packet. Once the circuit is established, all the other flows will be blocked until the QoS detector identifies a tear down packet. Once the circuit is released other flows can progress because the allocator will grant them under BE basis.

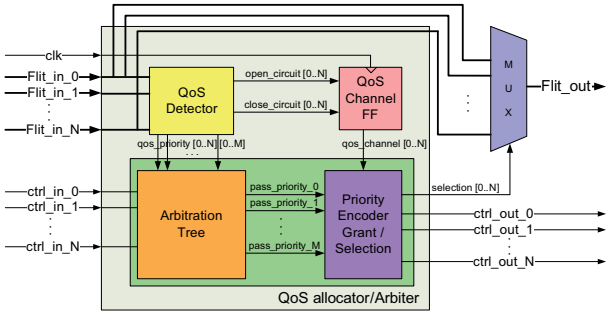

Figure 1: QoS support on allocator/arbiter

3.3 Hardware Synthesis QoS Results

In this section, we synthesis the components on different FPGAs devices[1]. We report and we quantify the overhead in terms of resource usage, and the degradation of the circuit performance in terms of f_{max} according to the implemented QoS features.

At NI level, as shown in Figure 2, the average overhead to include QoS extensions on the AMBA 2.0 AHB NI is about 10-15% depending on the FPGA device. The main overhead is due to the necessity to encode the different QoS levels, and to include the configuration registers, as well as several changes to the request FSM in order to generate the "fake" transactions to open/close channels or circuits.

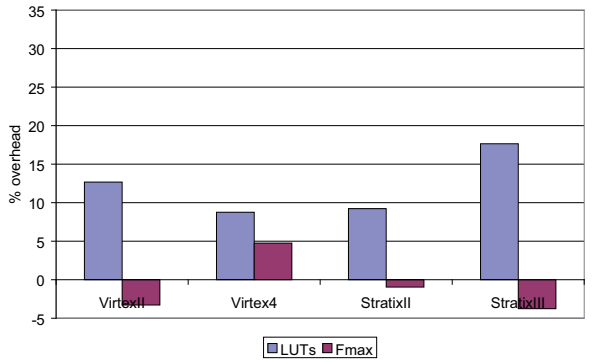

Figure 2: QoS impact on AMBA NIs (LUTs, f_{max})

In terms of f_{max} of NIs, the inclusion of QoS can be considered negligible. Surprisingly due to synthesis heuristics in Virtex4 FPGAs, the synthesis gives better f_{max} for the extended NI with QoS than without, even if a priori, the QoS NI is a little bit more complex. However, on average among the tested FPGA devices the circuit performance is basically unaffected.

In Figure 3 we explore the switch scalability varying the number of priority levels on each of the different configurations used to build our NoC-based platform presented in Figure 5. As expected, when 8 priority levels are used, the area cost increases considerably, roughly doubling the area of the switch without QoS. Despite this fact, the area resources are moderately impacted when 2 or 4 priority levels are used with an overhead ranging from ≈23-40% in Virtex FPGAs. In Stratix chips the trend is similar, but now the overhead to include 2 or 4 levels is higher going from ≈25-56% for the worst case.

In terms of f_{max} at switch level, as shown in Figure 3, as expected the circuit frequency drops when more levels of priority are supported. In fact, for most scenarios of traffic classes and use cases, in the embedded domain, the requirements can be fitted using up to 8 levels of priority. In addition, even if the overhead to include 2 and 4 priority levels is not negligible, it can be assumed taking into account the potential benefits to have runtime QoS on the final system.

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

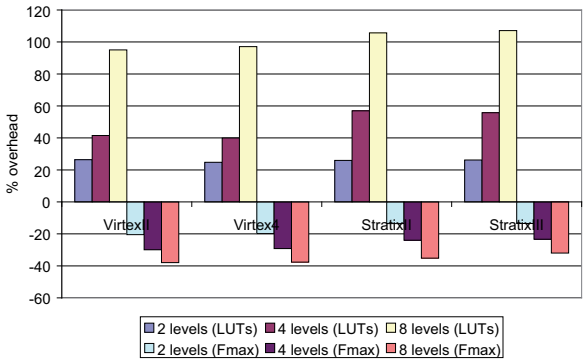

Figure 3: QoS impact at switch level (LUTs, f_{max})

4. EXPOSING NOC-LEVEL RUNTIME QOS FEATURES TO THE SOFTWARE LAYER

Usually HW-SW network systems are organized in layers to create different levels of abstraction that hide the complexity of the whole system, and expose the interactions between components.

Figure 4 shows the layered organization of our NoC-based MPSoC and the integrated components based on the micro network stack proposed in literature [15, 18, 29, 38].

- *Application layer:* At the topmost level of the software stack there is the parallel application, i.e. the OpenMP program. Parallel execution is supported by the underlying runtime environment (RTE). QoS features are integrated within the RTE, and are implemented as a wrapper around the middleware routines of the low-level API.

- *Transport layer:* It is in charge of injecting/receiving packets using NIs initiator and target over the on-chip network, respectively.

- *Network layer:* It is responsible for the transmission/reception of packets using BE or the QoS features.

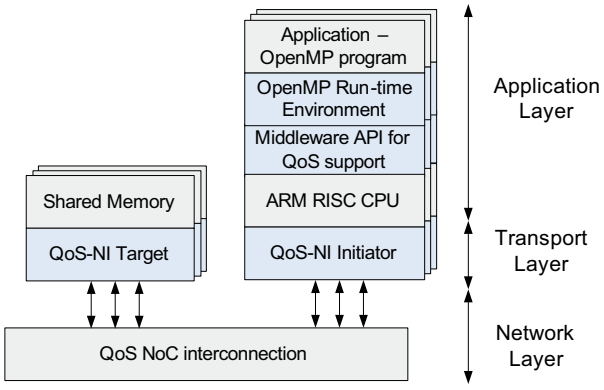

Figure 4: HW-SW layered view of our NoC-based MPSoC architecture

4.1 Low-level and Middleware QoS Support APIs

To efficiently exploit QoS features we implemented a set of low-level APIs, which directly interact with the NIs. The functions enclosed in the API are extremely lightweight (only few assembly instructions, executing in few processor cycles). The four functions of which the low-level QoS API is made up of are described below.

Listing 2: Low-level API QoS support

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

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

// Write a priority packet to a specific address
int ni_send_priority_packet(uint32_t address,
                            int data, int level);

// Receive a priority packet from a specific address
int ni_recv_priority_packet(uint32_t address,
                            int level);
```

In order to expose to the programming model layer functions that closely resemble the OpenMP semantics, we provide three additional functions to set/release priority between two end-points, and two more to enable GT data streams. These functions are described below.

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
void 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
void resetPriority(int PROC_ID, int MEM_ID);

// Reset all priorities W/R packet on system
void 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);
```

4.2 Exposing QoS-related Features to the Programming Model

OpenMP is a widely adopted shared memory programming API. 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 [11, 25, 28] have been proposed.

There are a number of ways in which priorities can be exploited in parallel regions and sections within the OpenMP programming framework.

In this work, we choose to extend the OpenMP API with custom directives to trigger prioritized execution for a particular thread (or group of threads). This option is useful to give the knowledgeable programmer the possibility of specifying appropriate prioritization patterns at different program points. This can be done by providing a custom `prioritized` clause, to be coupled with OpenMP constructs to outline parallelism.

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

All threads belonging to an annotated parallel team will be treated by the RTE as high-priority entities. The compiler inserts calls to the runtime library within the outlined parallel code, which is executed by every prioritized thread.

```
// Code executed by prioritized threads
{
  // Call RTE to determine priority settings
  int PROC_ID = get_proc_num ();
  int MEM_ID = // Determine target memory ID
  int level = get_pri_level (PROC_ID, MEM_ID);

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

  // PARALLEL REGION

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

Since the RTE dispatches threads to available cores, and it can be aware of the architectural topology[2], it will automatically set prioritized communication at the NoC level to deliver QoS. There are currently two main means to determine the ID of the memory being targeted by transactions issued within prioritized threads.

Method 1: the programmer to specify a target bank, annotating it in the `prioritized` clause. This task is not as onerous as it may appear, since data allocation can be controlled with the `distributed` directive.

```
// Specify that array A must be allocated
// on memory with ID 2
int A[100];
#pragma omp distributed (A, 2)
```

The programmer then knows exactly on which memory a given data item resides, and can annotate the information on the `prioritized` clause to set privileged transactions towards that memory. However, this solution requires deeper programmer involvement, and may compromise the ease of programming to some degree.

Method 2: we adopt an alternative approach of determining which target memory is referenced within prioritized threads. Application profiling allows to inspect each thread's data access pattern, at any point in time, and guarantees that highest possible priority is precisely established for each parallel region.

Once a processor/memory pair responsible for transaction action has been identified the RTE is invoked to determine a suitable priority level for corresponding transactions.

[2] i.e. the physical position of the core in the NoC, and how it is interconnected with memory devices.

Within the `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. Maximum priority level is assigned to each channel in case no conflicting transactions take place (i.e. each thread accesses a different memory). In case the target memory is shared between two or more threads different priority levels are assigned depending on the physical path that each thread has to traverse, and taking into account the effect of the routing algorithm on the network.

OpenMP also allows to model task parallelism with the `sections` directive, which we also enhance with prioritized execution-related features. In the example below 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 below.

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

  #pragma omp section prioritized
  task_B();

  #pragma omp section channel(MEM3)
  task_C();
}
```

If a task has hard constraints and requires an exclusive communication channel towards a given memory or I/O device, we allow the programmer to set such a communication link through the use of the custom `channel(mem_id)` clause as shown above for `task_C`. The `channel(MEM_ID)` clause establishes an exclusive communication channel between the processor hosting the annotated task and the memory device `MEM_ID`, indicated by its ID in the system. As explained previously for the `prioritized` clause, in general, a programmer willing to use the `channel` clause should know the ID of the target memory. However, if no such information is available at the application level, we can use a mix of compiler analysis and profiling information to determine which memory is accessed within the annotated task. Thus, the programmer does not need to know the physical address and the memory map of the system.

5. OVERVIEW OF NOC-BASED MPSOC ARCHITECTURE

Our architecture, shown in Figure 5, is a 4x4 2D Quasi-Mesh NoC-based MPSoC platform. The NoC is based on the ×pipes library [7, 39].

The system is representative of a real embedded handset mobile template. It includes 8 ARM processors which are in charge of executing audio and video encoding/decoding, transmitting and receiving data from the base band, etc. OpenMP program execution also takes place therein, as well as dynamic or static set-up or tear-down of the NoC QoS features. One processor executes the role of a master, and the rest of the cores in the cluster is configured as "slaves". The system includes eight banks of shared memory, as well as several typical I/O devices.

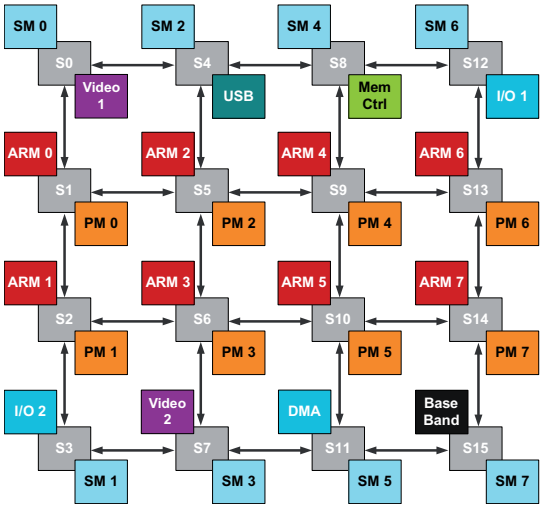

Figure 5: 4x4 2D Mesh NoC-based MPSoC platform

The memory subsystem leverages private and shared segments, and is organized as a two-level hierarchy. Each PE has on-tile L1 memory, which features separate instruction and data caches, and is logically associated to a private L2 memory bank, where by default program code and data private to the core are allocated. Private L2 memory is only cacheable by local L1 cache. Each PE is also logically associated to a shared L2 memory bank. All shared L2 memory modules are mapped in the address space of the processors, globally visible within a single shared memory space.

Processors can directly communicate through the L2 shared memory, which features both cacheable and non-cacheable banks. Data allocated in the cacheable bank can be cached by every processor, therefore multiple copies of the same shared memory location may exist simultaneously in the L1 caches. This requires a cache coherence protocol to be implemented. OpenMP specifies a relaxed consistency memory model, which requires that a coherent view of shared data is enforced only at specific synchronization points. Cache coherence is thus enforced through software flush instructions in our runtime library.

6. EXPERIMENTAL RESULTS

In this section we describe the experimental setup that we considered to implement and evaluate the proposed framework (see Figure 5 and execute real application and benchmarks on it. All HW and SW components have been integrated in a NoC-based platform on MPARM, a full-featured SystemC full system simulator [4].

Furthermore, we integrate the support for QoS (i.e. the middleware API) at the programming model level in a MPSoC-specific implementation of the OpenMP compiler and runtime library [28] based on the GCC 4.3 toolchain.

To test the effectiveness of the approach we employ a set of variants of the *Loop with dependencies* benchmark from the OmpSCR (OpenMP Source Code Repository [16]) suite. In this program a number of parallelization schemes are considered to resolve loop carried dependencies. Due to the alternation of communication- and computation-intensive loops, as well as the possibility of finely tuning the workload through several parameters, this benchmark provides

several interesting case studies, representative of real application patterns. Typically, OpenMP programs achieve balanced parallel execution time between worker threads by allocating similar amounts of work to each of them. The most common example in this sense is loop parallelization with static scheduling, where the iteration space is evenly divided among threads. Communication is also evenly distributed: threads reference distinct equally-sized subsets of a shared data structure (e.g. an array).

From the architecture-agnostic application (or compiler) point of view, this is all that can be done to achieve workload balancing. Unfortunately, when mapping the application onto the hardware resources, many issues arise that can break balancing. High contention for the memory device where shared data is allocated may cause one (or more) thread(s) to be delayed in accessing its dataset. Due to the OpenMP semantics, where a barrier is implied at the end of each parallel region, this delay leads to overall program execution lengthening. Our aim is to explore the effectiveness of prioritize packets in solving this issue.

To model the described problem we allocate four of the eight available ARM processors to the OpenMP program, whereas the remaining four processors host independent threads that generate interfering traffic. We consider three cases:

- All OpenMP threads within a *communication-intensive* loop are delayed in accessing shared data on a unique memory device.
- All OpenMP threads within a *computation-intensive* loop are delayed in accessing shared data on a unique memory device.
- Every OpenMP thread accesses a separate memory device: a single thread is delayed.

The effect of our custom language features for guaranteeing services on critical tasks is also studied. We describe each experiment in detail in the following sections.

6.1 Communication Dominated Loop

One of the techniques considered in our benchmark for loop parallelization in presence of dependencies is that of replicating the target array prior to overwriting its locations with newly-computed data. The copy operation is fully parallel, with threads simply reading and writing to memory. In this experiment OpenMP threads are hosted on processors with even IDs (i.e. ARM_0, ARM_2, ARM_4, ARM_6). We allocate the target program arrays on a single memory device, namely SM 7. The odd processors (i.e. ARM_1, ARM_3, ARM_5, ARM_7) execute application code generating interfering traffic with the OpenMP team.

Figure 6 reports the normalized execution time for this experiment with and without prioritization. When no prioritization is enabled, it is possible to notice that OpenMP threads experience very unbalanced execution, due to the effect of memory device contention. Adding priorities to each thread at runtime allows to achieve perfect workload balancing.

Program execution – whose performance degradation was originally dictated by the huge delay on ARM_0 – is sped up by 40.31%. It is possible to notice a kind of "system-level" effect of priorities. The time saved on even processors is re-distributed across odd processors.

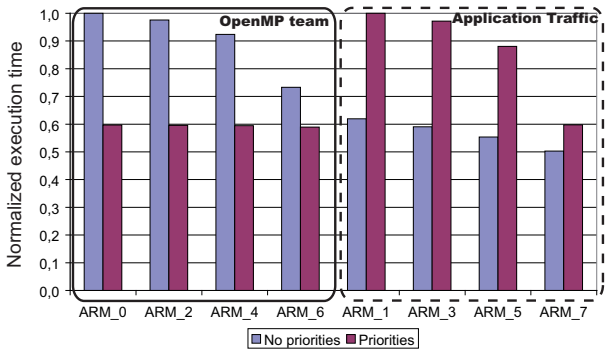

Figure 6: Effect of prioritized transactions on an OpenMP program unbalanced by memory contention (communication dominated loop)

6.2 Computation Dominated Loop

Once a copy of the original array content is stored in a replica, the second loop of the benchmark computes the new array element values by applying a computation-intensive kernel. The results for this experiment are reported in Figure 7. Here the setup is largely similar to that described in the previous section. For this reason the behaviour of this loop closely resembles that of the previous one. Unsurprisingly, due to the reduced computation to communication ratio, here the effect of the high priority transactions is less pronounced. Nonetheless, a speedup of 32.83% is achieved on loop execution.

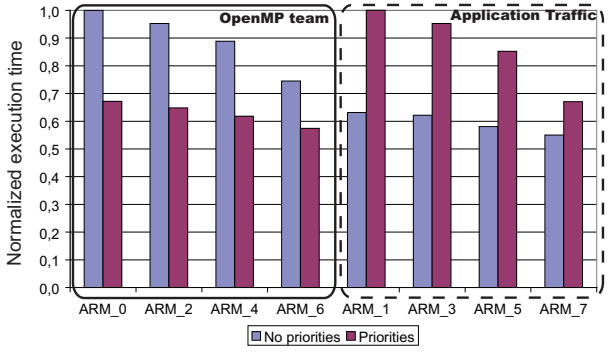

Figure 7: Effect of priority transactions on an OpenMP program unbalanced by memory contention (computation dominated loop)

6.3 Single Thread Interference

As a final experiment, we want to explore the role of priorities in a situation in which every OpenMP thread is accessing a separate shared memory device (thus no contention generates from the program itself). However, a single thread is delayed in communicating with a memory by concurrent operations performed by other devices in the system (e.g. an accelerator, DMA performing transfers). We model this situation in the following way. We map the OpenMP threads onto ARM_0 - ARM_3. Each of them executes the whole loop with dependencies benchmark, but we use a custom feature of our compiler that allows us to partition shared

arrays in as many tiles as cores, and place the individual tiles onto nearby memories [28]. Thus ARM_0 only accesses data on SM 0, ARM_1 one only accesses data on SM 1, etc.

On the other hand, all of the other application traffic threads and other elements on the NoC-based SoC, such as DMAs, are generating transactions to SM 3, namely the memory accessed by OpenMP thread on ARM_3. Figure 8 shows that this placement seriously degrades the performance of the OpenMP program, since thread three on ARM_3 is delayed by the other applications on the platform which are also issuing transactions on SM 3.

Even in this case it is possible to see that priorities completely solve the issue. Figure 8 shows that the prioritized OpenMP program achieves balanced execution, distributing its original delay over non-prioritized threads. In that case, the overall OpenMP program execution time achieves a 66,04% speedup.

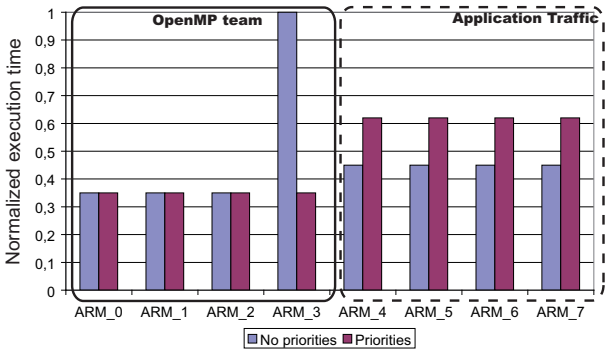

Figure 8: Effect of priorities on an OpenMP program unbalanced by a delayed thread

6.4 Guaranteeing Services to Critical Tasks

In this section, we experiment with a completely different approach of leveraging the custom `channel` clause to allocate/de-allocate channels to tasks with critical requirements. We consider a situation in which each processor in the system is busy executing a different task, one of which has real-time constraints (e.g. video encoding/decoding). To model this system behavior we start from the OpenMP program shown below.

Thus, we define three scenarios according to the proposed code without guarantees:

1. No guarantees for any task.
2. *Task0* is annotated with a `channel(video1)` clause, thus achieving guaranteed services.
3. *Task7* is annotated with a `channel(video1)` clause, thus achieving guaranteed services.

On each scenario Tasks 1-6 execute generic workloads, with the sole purpose of generating interfering traffic with critical transactions issued by tasks 0 and 7.

In Figure 9, it is easy to notice that ARM_0 without BW guarantees is not extremely delayed when accessing video 1 since it is only one-hop distant. However, the service is not guaranteed because of the routing or the fixed priorities of the NoC backbone. Thus, we try to guarantee the service on ARM_0 by opening a channel.

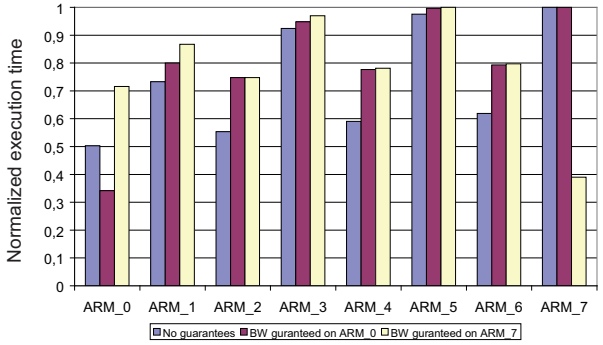

Figure 9: Effect of communication channels on critically-constrained tasks

Thus, as shown in the code below, we guarantee the services in the critical *Task0* (i.e. video encoding) simply by inserting the `channel(video1)` clause in the *Task0* section. The consequence is that the runtime environment automatically will embed calls to open/close GT channels using the QoS middleware functions presented in Section 4 in order to ensure bandwidth and latency bounds.

```
#pragma omp parallel sections
{
  #pragma omp section
    task0; //(video encoding)
  #pragma omp section
    task1;
  #pragma omp section
    task2;
  #pragma omp section
    task3;
  #pragma omp section
    task4;
  #pragma omp section
    task5;
  #pragma omp section
    task6;
  #pragma omp section
    task7; //(video decoding)
}
```

```
#pragma omp parallel sections
{
  #pragma omp section channel(video1)
    task0; //(video encoding)
  #pragma omp section
    task1;
  #pragma omp section
    task2;
  #pragma omp section
    task3;
  #pragma omp section
    task4;
  #pragma omp section
    task5;
  #pragma omp section
    task6;
  #pragma omp section
    task7; //(video decoding)
}
```

As shown in Figure 9, the performance is improved by 16.13% with respect to the previous scenario with no guarantees, where other tasks were generating interfering transactions. This experiment shows that we can ensure bandwidth or latency requirements avoiding any other interference from the rest of the system.

Finally, let us focus on the decoding task assigned to ARM_7. It is clear by looking at the results for the initial scenario without no guarantees that the *Task7* could not meet its deadlines, thus potentially leading to some frame-dropping. Thus, as before, we establish a channel from ARM_7 to video 1 by annotating the corresponding `section` directive with our `channel(video1)` clause. The results are shown in Figure 9 (see BW guaranteed on ARM_7 columns). It is possible to observe that ARM_7 has now latency bounds, and achieves a speedup of 61.98% with respect to the traffic with no gurantees.

After the completion of an annotated critical task on the sections, its established channel is torn down, and the non-prioritized and non-guaranteed communication is restored in the NoC.

7. CONCLUSION AND FUTURE WORK

In this paper we presented an integrated HW-SW approach for QoS support in NoC-based MPSoCs. More specifically, we employ a layered design to expose the hardware QoS features of our NoC architecture to the runtime environment of an enhanced OpenMP programming framework. This allows to build an infrastructure to assign varying levels of priority and guaranteeing services by means of allocation channels issued by OpenMP tasks.

By tightly-coupling HW with a streamlined SW implementation of the QoS-support API we obtained encouraging results on the effectiveness of the approach. Experiments on a set of OpenMP kernels, representative of patterns found in typical embedded applications, show that under different traffic patterns and thread allocation schemes the use of prioritized transactions boosts the overall execution time by up to 66%.

Future work will focus on further extending the OpenMP support to QoS-related features. Dynamic priority adjustments entail negligible overhead even at the single read-/write level. Therefore the design overhead vs. performance gain of very fine grain QoS tuning is open for future study.

Acknowledgments

This work was partially supported through AGAUR (Ref. 2009BE200202), ITEA2 ParMA project, as well as, JTI SMECY and FP7 SHARE and PRO3D projects funded by the European Community.

8. REFERENCES

[1] 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.

[2] AMBA 3 AXI overview, 2005. `http://www.arm.com/products/system-ip/interconnect/axi/index.php`.

[3] ARM AMBA 2.0 AHB-APB Overview, 2005. `http://www.arm.com/products/system-ip/interconnect/amba-design-kit.php`.

[4] 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*, 41(2):169–182, September 2005.

[5] L. Benini and G. De Micheli. Networks on Chips: A new SoC Paradigm. *IEEE Computer*, 35(1):70 – 78, January 2002.

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

[7] D. Bertozzi and L. Benini. Xpipes: A Network-on-Chip Architecture for Gigascale Systems-on-Chip. 4(2):18–31, 2004.

[8] T. Bjerregaard and J. Sparso. A Router Architecture for Connection-Oriented Service Guarantees in the. MANGO Clockless Network-on-Chip. In *Proc. Design, Automation and Test in Europe*, pages 1226–1231, 2005.

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

[10] 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.

[11] 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*, pages 1–8, 2009.

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

[13] W. J. Dally. Virtual-Channel Flow Control. *IEEE Trans. Parallel Distrib. Syst.*, 3(2):194–205, Mar. 1992.

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

[15] 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*, volume 00, page 3pp., Mar. 2–3, 2006.

[16] A. Dorta, C. Rodriguez, and F. de Sande. The OpenMP Source Code Repository. In *Parallel, Distributed and Network-Based Processing, 2005. PDP 2005. 13th Euromicro Conference on*, pages 244–250, Feb. 2005.

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

[18] 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*, pages 423–425, Mar. 4–8, 2002.

[19] 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, pages 233–242, 2007.

[20] A. Hansson, M. Subburaman, and K. Goossens. Aelite: A Flit-Synchronous Network on Chip with Composable and Predictable Services. In *Proc. DATE '09. Design, Automation. Test in Europe Conference. Exhibition*, pages 250–255, Apr. 20–24, 2009.

[21] A. Hansson, M. Wiggers, A. Moonen, K. Goossens, and M. Bekooij. Enabling Application-Level Performance Guarantees in Network-Based Systems on Chip by Applying Dataflow Analysis. *IET Computers Digital Techniques*, 3(5):398–412, Sept. 2009.

[22] I. Miro Panades, A. Greiner., A.Sheibanyrad. A Low Cost Network-on-Chip with Guaranteed Service Well Suited to the GALS Approach. In *Nano-Net 2006*, 2006.

[23] A. Jerraya and W. Wolf. *Multiprocessor Systems-on-Chips*. Morgan Kaufmann, Elsevier, 2005.

[24] A. A. Jerraya, A. Bouchhima, and F. Pétrot. Programming Models and HW-SW Interfaces Abstraction for Multi-Processor SoC. In *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, pages 280–285, New York, NY, USA, 2006. ACM.

[25] 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*, pages 44–49, Jan. 23–26, 2007.

[26] L. Kriaa, A. Bouchhima, M. Gligor, A. Fouillart, F. Pétrot, and A. Jerraya. Parallel Programming of Multi-processor SoC: A HW-SW Interface Perspective. *International Journal of Parallel Programming*, 36(1):68–92, 2008.

[27] 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*, pages 116–121, June 4–8, 2007.

[28] 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*, pages 809–814, Apr. 20–24, 2009.

[29] 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*, pages 693–696, 2004.

[30] R. Mullins, A. West, and S. Moore. Low-Latency Virtual-Channel Routers for On-Chip Networks. In *Proc. 31st Annual International Symposium on Computer Architecture*, pages 188–197, June 19–23, 2004.

[31] S. Murali, M. Coenen, A. Radulescu, K. Goossens, and G. De Micheli. A Methodology for Mapping Multiple Use-Cases onto Networks on Chips. In *Proc. Design, Automation and Test in Europe DATE '06*, volume 1, pages 1–6, Mar. 6–10, 2006.

[32] 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*, pages 355–362, 2006.

[33] K. OBrien, K. OBrien, Z. Sura, T. Chen, and T. Zhang. Supporting OpenMP on Cell. *International Journal of Parallel Programming*, 36(3):289–311, 2008.

[34] OCP International Partnership (OCP-IP). Open Core Protocol Standard, 2003. `http://www.ocpip.org/home`.

[35] U. Y. Ogras, J. Hu, and R. Marculescu. Key research Problems in NoC Design: A Holistic Perspective. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 69–74, 2005.

[36] A. Pullini, F. Angiolini, S. Murali, D. Atienza, G. De Micheli, and L. Benini. Bringing NoCs to 65 nm. *IEEE Micro*, 27(5):75–85, Sept. 2007.

[37] 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*, pages 350–355, 2003.

[38] 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*, pages 667–672, 2001.

[39] 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*, pages 1188–1193, 2005.