On-Chip Network and OmpSs Co-design: Exploiting Task Criticality

Iván Pérez*, Emilio Castillo†, Ramón Beivide*, Miquel Moretó†, Enrique Vallejo*, Marc Casas†, Fernando Vallejo*

*University of Cantabria †Barcelona Supercomputing Center

Introduction

Tasks of data-flow programming models like OmpSs may have different levels of criticality within a program. For some applications, reducing duration of critical tasks could reduce significantly their execution times.

Our goal is to improve performance of OmpSs applications reducing the miss penalty of critical tasks trough QoS in the network on-chip (NoC) of multiprocessors.

Networks on-Chip (NoC)

NoCs are becoming more important with the increasing number of cores in modern SoCs. Most of them have memory hierarchies with a shared level of cache, banked along the chip to increase its bandwidth as shown in Figure 1. Memory data travelling across the memory hierarchy move through the NoC. The most usual NoC topologies of commercial multiprocessors are rings and meshes.

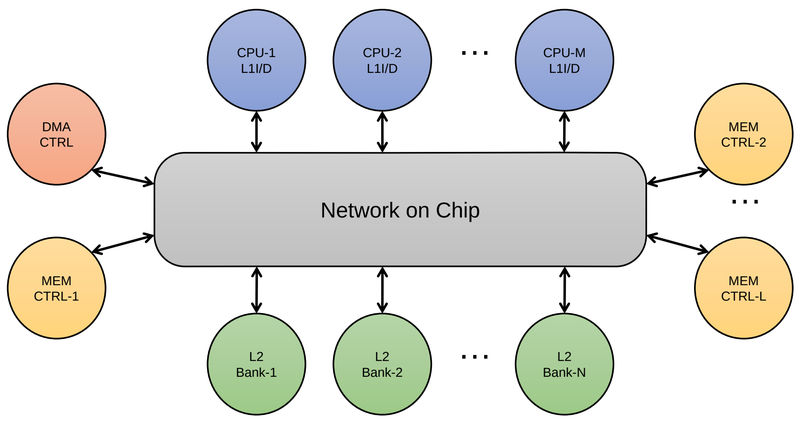

Figure 1: Typical memory hierarchy diagram of multicore processors with: *M* CPUs with private L1I/D caches, *N* banks of shared L2 cache, *L* memory controllers and a DMA controller

OmpSs

OmpSs follows a dataflow paradigm defined by data dependencies. A task is a sequential unit of work that can be executed by a thread (or core) when its data dependencies are satisfied.

Defining tasks dependencies are suitable for managing fine grain synchronization which is hard in complex parallel algorithms using traditional thread programming models.

Figure 2 shows the task-dependency graph of a Cholesky's factorization. Each circle represents a task, and edges are data dependencies between tasks. In the ideal case in which all tasks have the same length, tasks in red are critical, as they are chained and must be executed sequentially.

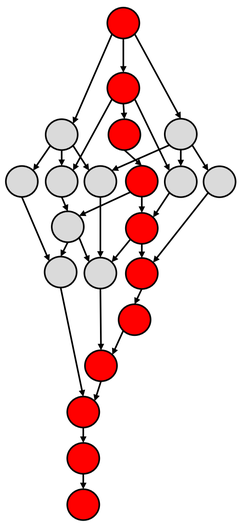

Figure 2: Cholesky's task-dependency graph

Exploiting Criticality of Tasks on NoC

We define criticality as the relevance that a task has for an efficient progress of the application. Our idea to exploit criticality from the NoC is to prioritize traffic from critical tasks over non-critical using QoS. This way we can increase the NoC bandwidth for critical tasks.

We assign a priority value to each memory instruction according to the criticality of the task that each core is executing. In a first approach, we assign the priorities for each task inside the source code, within the OmpSs' pragmas for task definition. These priorities are already used by the OmpSs scheduler to serve first critical ones. Then, priorities propagate from the scheduler to the cores running tasks using special instructions as described in [1].

NoC packets, which are the result of cache misses, inherit these priorities that are used to assign resources and segregate traffic.

The QoS configuration we evaluated works as follows:

- We segregate traffic in two classes of packets: high and low priority if their priority overpass a threshold or not. High priority packets correspond with critical tasks, and critical states of the OmpSs' runtime like task creation. Then, we divide VC resources between both classes as shown in Figure 3.
- In addition, priority values are used during arbitration (Virtual Channel Allocation: VA, and Switch Allocation: SA) to favour packets with the highest priority. Figure 4 shows the router architecture used and remarks the arbitration stages where packet priorities are used.

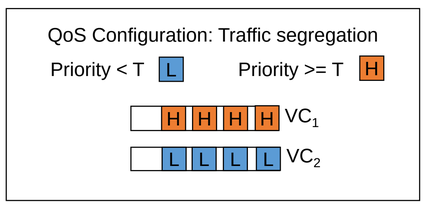

Figure 3: VC configuration for traffic segregation

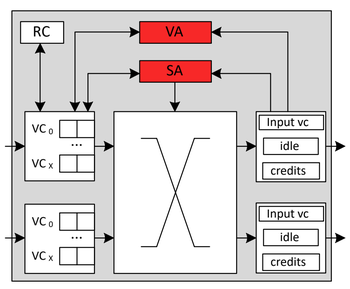

Figure 4: Router architecture. Arbitration stages are represented in red.

Tools

Our main tool is gem5 [2], a full-system multiprocessor simulator. We employ the MESI Two Levels coherence protocol from the Ruby memory model to evaluate the NoC. With this configuration, gem5 allows for co-designing OmpSs and NoC.

We use BookSim [3], a versatile and accurate NoC model, to evaluate the QoS configuration with synthetic traffic. And to evaluate real applications, we integrated BookSim, within gem5, replacing the standard models which have several limitations, like a fixed router architecture, flow control and routing algorithm.

Preliminary Results

First we evaluated the QoS configuration under random uniform synthetic traffic in a 4x4 mesh using BookSim:

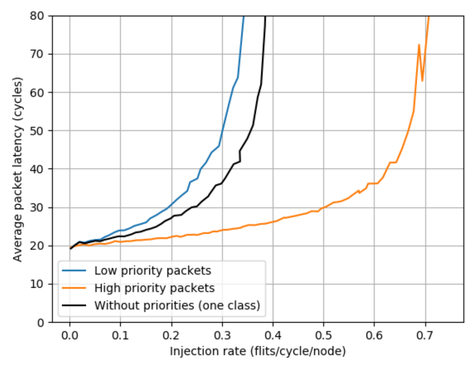

Figure 5: Average packet latency with and without priorities. Half of the load is represented without priorities for a fair comparison

Figure 5 shows a throughput increment of around a 185% for high priority packets, while low priority packets reduce it in a 12% when compared to the case without priorities.

Preliminary full-system results in this poster were obtained from a synthetic application which has a high amount of memory instructions and a high miss rate. It has two types of tasks executing the same code but one of them is defined as critical, while the other not. Thus, we are able to isolate the impact of the QoS configuration in the miss penalty easily.

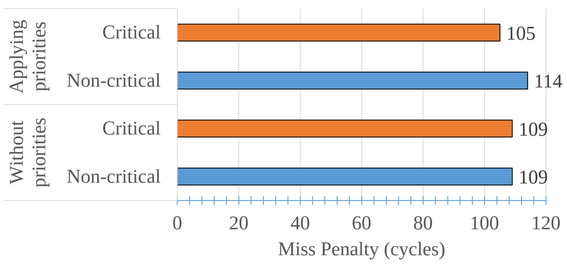

Figure 6.a: Average miss penalty of tasks

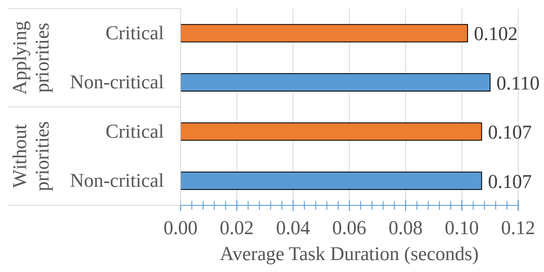

Figure 6.b: Average task duration

As shown in Figure 6.a, miss penalty of critical tasks is reduced in a 4%, but it is increased for non-critical tasks in a 6%. In terms of task duration the results are similar, a reduction of a 5% and an increment of a 3% for critical and non-critical tasks respectively as shown in Figure 6.b. These low differences are caused by a low level of NoC contention which is a result of low injection rates under the full-system simulation (lower than 0.1 flits/cycle/node). The high throughput increment showed with synthetic traffic (Figure 5) is almost irrelevant for real applications as the NoC operates at very low loads, where packet latencies are very similar for both classes of traffic. Another important fact is that, as we improve performance of critical tasks, we degrade performance of non-critical ones.

Therefore we have two important challenges to face in the future:

1. Find a configuration that stresses more the NoC to increase the margin of improvement for critical tasks.
2. Look for applications with non-critical tasks that can tolerate a huge miss penalty degradation without becoming the new critical path.

References

[1] Castillo, E. et al. CATA: Criticality Aware Task Acceleration for Multicore Processors. In *Parallel and Distributed Processing Symposium, 2016 IEEE International* (pp. 413-422). IEEE.

[2] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., ... & Sen, R. (2011). The gem5 simulator. *ACM SIGARCH Computer Architecture News*, *39*(2), 1-7.

[3] Jiang, N., Balfour, J., Becker, D. U., Towles, B., Dally, W. J., Michelogiannakis, G., & Kim, J. (2013, April). A detailed and flexible cycle-accurate network-on-chip simulator. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on* (pp. 86-96). IEEE.

Acknowledgements

Apart from the collaboration grant received from the Red Consolider SyeC (TIN2014-52608-REDC), this work has been supported by the Spanish Ministry of Economy, Industry and Competitiveness under contract TIN2016-76635-C2-2-R (AEI/FEDER, UE), the European Union FP7 under Agreement ERC-321253, the European HiPEAC Network of Excellence, The Mont-Blanc project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 671697.

ivan.perezgallardo@unican.es