Deploying a Telecommunication Application on Multiprocessor Systems-on-Chip

Daniela Genius, Etienne Faure, Nicolas Pouillon

Laboratoire LIP6/Département SOC
Université Pierre et Marie Curie
E-mail: {daniela.genius,etienne.faure,nicolas.pouillon}@lip6.fr

Abstract

The particular form of the task graph of many telecommunication applications permits a high level of coarse grained parallelism. We consider a classification application on a telecommunication oriented multi-processor system-on-chip (MP-SoC) platform. The hardware architecture on which this type of application is executed can contain a variable number of programmable processors and of dedicated hardware co-processors, sharing the same address space. Inter-task communications are implemented via Multi Writer Multi Reader FIFOs placed in shared memory. This paper analyzes additions and modifications required to enable the Design Space Explorer DSX to meet the requirements for the deployment of such an application.

1 Introduction

Telecommunication applications can be considered a special case of streaming applications; They are usually processing packet streams, where the same processing is performed on each packet, but the actual computing depends on the packet contents. Throughput requirements are variable: backbone equipments, such as routers, require high throughput and low processing per packet, while traffic analysis requires less throughput but intensive computation per packet. For [17], this variable processing time, depending on the packet type, is the main characteristic of network applications.

We focus on telecommunication applications written in the form of a set of coarse grain parallel threads communicating with each other. Inter-task communications can be done through message passing like in STepNP [12], modeled in the form of data flow graphs like StreamIt [6] and Ptolemy [3] or can use the shared memory capabilities of the multi-processor hardware architecture.

Kahn Process Networks (KPN [9]) propose a semantics of inter-task communication through infinite FIFO channels with nonblocking writes and blocking reads. Such infinite channels are impossible to implement, thus KPN formalism has been adapted for example by YAPI [5]. To deal with the select problem YAPI introduces the select function, which makes the model nondeterministic. Implementations of YAPI are COSY [2] and SPADE [18]. Disydent (Digital System Design Environment [1]) is also based upon KPN and uses point to point FIFOs. While the KPN formalism is well suited for video and multimedia applications, that can be modeled by a task graph where each communication channel has only one producer and one consumer, it is not convenient for telecommunications applications where several tasks access the same communication buffer, in order to consume (or produce) packet descriptors.

MWMR (Multi-Writer / Multi-Reader) channels [8] are software FIFOs that can be accessed by several reader and writer tasks. The communication protocol is described in more detail in [7]. The generic MWMR communication channel supports both hardware or software producer or consumer, making possible to decide quite late whether a task should be implemented in software or hardware.

DSX (Design Space Explorer, [10]) is based upon the MWMR concept. It extends Disydent by a comfortable user API for design space exploration. The aim of DSX is to have *one* application description, *one* hardware description, and mapping rules which can be easily modified, all written down in *one* common language. It offers extensive debug capabilities: the arduous task of deployment comes before the fine-tuning of parameters.

2 Application Specification

In order to extract the coarse grain parallelism from a sequential application, two basic approaches exist. The first one relies on the coarse-grained segmentation of the sequential application. The algorithm is split into functional tasks that execute sequentially. This is called *pipeline parallelism*. The other consists in duplicating the whole sequential application into many clones. All the tasks are doing the same job, but each one on different data. This is known as *task farm parallelism*. This task farm model is convenient for telecommunication applications processing successive and independent packets in a Gigabit Ethernet stream.

Task graph Task farm and pipeline parallelism can be combined to yield any hybrid of graph between these two forms such as figure 1. All communication between tasks

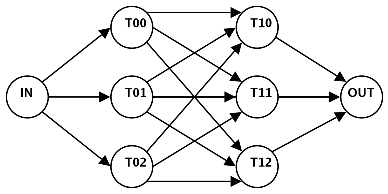

Figure 1. Example of hybrid parallelism

use point to point channels, that can be implemented as software FIFOs, in order to handle the asynchronous behavior of the tasks. Figure 1 represents communication channels by arrows between tasks. The FIFOs implementing the communication channels are implicit.

MWMR channels In many cases, the data produced by a task is not destined to one particular task, but rather to a class of tasks. Assume that tasks T00, T01 and T02 in figure 1 are three instances of the same computation, and that T10, T11 and T12 are three instances of another computation. In this case, the first three tasks can send their output to any of the three others. It is evident that we should try to replace the nine separate communication channels by one single, multi-access communication channel. Figure 2 shows one single FIFO, shared by three producers and three consumers.

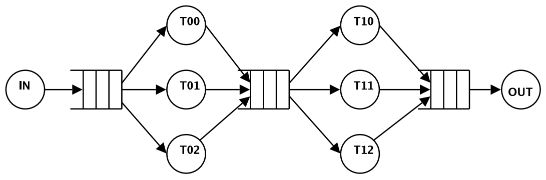

Figure 2. explicit MWMR channels

This new task and communication graph (TCG) is now a bipartite graph that describes the intrinsic coarse grain parallelism of the application, without precising the type of implementation: As both programmable processors and hardware coprocessors can read from or write to a given software MWMR, each task can be implemented either as a software task (running on a a programmable processor), or as a dedicated hardware coprocessor. KPN channels can be implemented as a special case of the proposed MWMR communication formalism: In order to implement the KPN semantic, the task graph must have only one producer and one consumer per channel, and all the accesses to the FIFOs must be enclosed into a loop [11].

3 The Target Hardware Architecture

The target hardware architecture is a multi-processor system on chip based on SoCLib [16]) components and running the MUTEK [14] micro kernel. It contains a variable number of 32 bits processors (currently MIPS R3000), a variable number of embedded RAM banks and other components such as lock engine, interrupt controller, and several I/O coprocessors. All these components are communicating through a VCI/OCB compliant micro-network [19]. There are two types of components: initiators and targets. Initiators send request packets, routed to the appropriate target by the interconnect, targets send response packets.

All initiators and targets share the same address space. In such hardware platform, using a large amount of processors, coprocessors and ram banks, the central interconnect has to provide a large throughput between initiators and targets. A conventional bus cannot offer such a throughput as it can only serve one communication at time. Thus we replace it with a NoC, which provides the required throughput. The NoC prevents us from using a snoop mechanism to ensure data coherency.

The communication protocol is based upon a shared memory multi-processor architecture. All MWMR channels are mapped in shared memory and access is protected by a single lock (one lock per channel). Each channel may have several readers and writers, but ignores their number. As illustrated by the following write request, the MWMR protocol requires five steps:

1. get the lock protecting the MWMR (READ access).
2. test the status of the MWMR (READ access).
3. transfer a burst of data between a local buffer and the MWMR (READ/WRITE access).
4. update the status of the MWMR (WRITE access).
5. release the lock (WRITE access).

For performance reasons, all transfers to or from a MWMR channel must be a multiple of 32 bit words. Figure 3 shows the hardware architecture with two processors and one memory bank. On the TTY the progress of the application can be observed, the Locks Engine manages the locks when more than one initiator is present (here four initiators: two processors and two coprocessors). These two components are VCI targets. The MWMR channel is located in on-chip RAM; it implements a communication channel between a software task running on CPU0 and a hardware task executed by coprocessor 1.

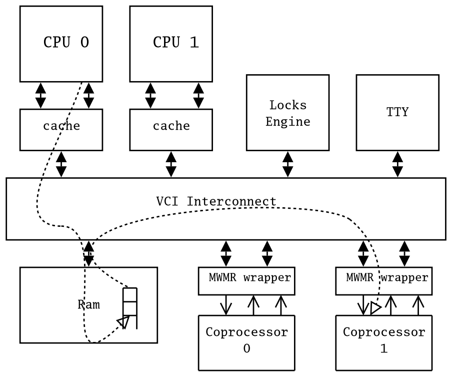

Figure 3. Use of a MWMR channel located in RAM, with one producer running on CPU0 and one consumer implemented in coprocessor 1

For performance reasons, MWMR channels are located in cacheable memory. Their coherency is guaranteed by software mechanism. Each cache line containing MWMR channel data is invalidated twice: before the five step access to ensure that data is read from memory and after the access to ensure that memory is updated. The latter is only necessary in the case of a write back cache mechanism.

4 The Telecommunication Platform

Until now, we have presented the *generic* hardware architecture. The telecommunication platform can be obtained by replacing the two coprocessors in Figure 3 by two application specific coprocessors called *InputEngine* and *OutputEngine*.

As usual in the domain of network processors, packets are cut into chunks of equal size which can be handled more efficiently and economize on-chip memory [4]. The basic idea of our coprocessors is that apart from this economy of space, time can be economized, too. A packet is represented by an eight byte *descriptor*, containing only a pointer to the beginning of the packet and the mandatory information to retrieve it. Necessary data are the address of the next slot, the total size of the packet, and an offset for eventual additional headers like for Multi Protocol Label Switching [15]. This leaves 120 bytes for the payload and an eventual offset. The use of descriptors allows us to avoid the copying of packets in memory most of the time. Consequently, the MWMR channels contain only descriptors.

Thus, I/O coprocessors must have a MWMR interface to send and retrieve descriptors as well as a VCI interface, the latter directly connected to the on-chip interconnect. SoCLib components are required to have a VCI interface, while coprocessors use FIFO interfaces. Thus, to implement MWMR channels a hardware wrapper with two VCI interfaces is required: a target interface for configuration and an initiator interface to fetch descriptors from the software channels located in on-chip RAM by issuing VCI requests. This wrapper is completely transparent for the user of DSX. Figure 4 shows on the lower right an *InputEngine* with one outgoing VCI and three FIFO channels. *InputEngine* and *OutputEngine* are symmetric concerning the interfaces.

A fundamental assumption of our architecture is that for a large majority of networking applications it is sufficient to consider the beginning of a packet. We privilege this first slot in so far as we store it in on-chip RAM, taking into account that this penalizes applications such as encryption/decryption or content control. Nevertheless the policy can be modified easily as only one bit has to be manipulated in order to determine whether a slot is to be stored on-chip or not.

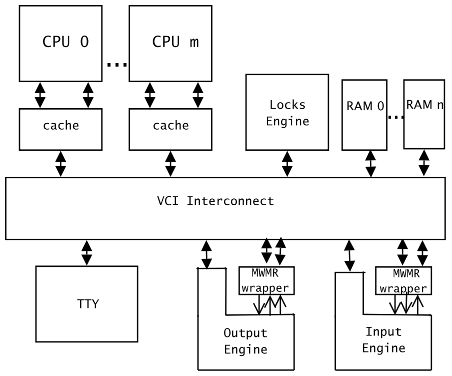

Figure 4. Telecommunication platform

The work described in [7] showed that a *clustered* architecture using a two level interconnect is better adapted than many initiators and targets grouped around one single interconnect. In such a NUMA (Non Uniform Memory Access) architecture, memory access times differ depending on whether a processor accesses a memory bank local to the cluster, or on another cluster. The architecture proposed in [7] has nine clusters, two of which contain only an I/O coprocessor each, on-chip RAM and an external RAM controller for packet storage (Figure 6) and no general purpose processor. Six clusters (Figure 5) contain four MIPS R3000 processors and four memory banks each. The last cluster contains only targets: one memory bank where the code of the application is located, and one lock engine that provides locks for the embedded OS.

5 The Classification Application

Classification is an important and resource-consuming part of many telecommunication applications [4] which takes place just after a packet arrives at the *framer chip* (input coprocessor in networking terminology). Packet headers are analyzed and sometimes more in-depth analysis, useful for traffic management, is performed on the packet content. The packet is then sent on to one of several priority queues. We restrict to header analysis; more precisely, the destination address determines to which of twelve priority queues the packet will go.

Besides input and output tasks, there are two levels of tasks, which makes the parallelism expressed in our task graph a hybrid of pipelined and task farm parallelism (see section 2). There is also a bootstrap task that organizes startup. Figure 7 shows the task graph of the classification application. In the following we will describe the five different types of tasks in some more detail.

Input Task The input task reads a packet from a file, determines its size, performs some basic checks, then computes the number of slots required taking into account the offset, and finally copies the slots to internal and external memory, respectively. In a last step, the slot is generated, and only its eight byte descriptor is sent to the outgoing MWMR channel, thus limiting the size of the memory allocated to the FIFOs. This write is non-blocking: if the channel is full, the packet is dropped and its address recycled. Addresses for the slots are obtained from either of three sources. Bootstrap: in the beginning, the input task needs to have a pool of internal and external addresses to allocate the slots to. Output task: when a packet leaves the system without having been dropped or discarded, the addresses of all its slots can be liberated. Classification task: errors are detected and packets can be discarded.

Classifi cation Task The classification task reads one or more descriptors and suspends itself if this is not possible. When it succeeds, it reads the first slot from on-chip memory. The validity of the checksum bits contained in the header is verified. The packet has to be dropped if one of

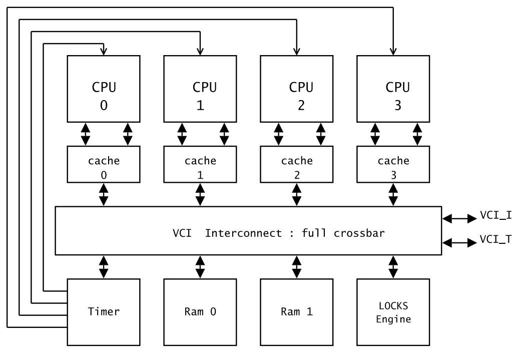

Figure 5. Architecture of a processing cluster

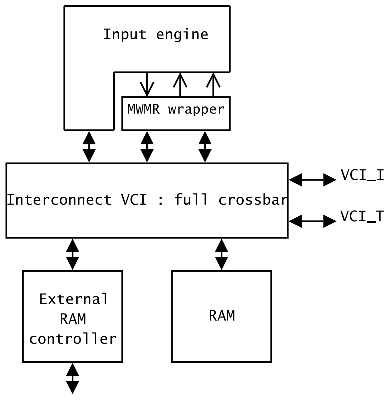

Figure 6. Architecture of the Input Cluster

those checks fails. Each slot begins with a descriptor containing the address of the next slot and one INTERNAL bit. Deallocation proceeds along these addresses from one slot to the next until the last slot is reached (next slot address is NULL). Next, depending on its destination address, the outgoing IP route is determined by consulting a routing table. This table is located in a shared memory segment so that all classification tasks can have a local copy in the cache of the processor on which they run. Update is assured by having the classification task sending a signal to all other classification tasks to invalidate their data cache. Our routing tables are relatively small: for lookup a simple iteration loop suffices. For fast lookup of larger routing tables, specific hardware is required. Finally the classification task writes the descriptor to one of the twelve priority queues[1].

Scheduling Task The scheduling task uses algorithm which ponders by the priority of the current queue and the number of packets that have already left for the output task in order to reach the percentage assigned (example: the highest priority queue is read 40% of the time in the long run, the lowest 5% etc.). The FIFOs are tested for eligibility in a round robin manner, necessitating a non blocking read primitive in order not to suspend execution on an empty queue. The descriptor of the eligible packet is then written to the unique output queue.

Output Task The output task is a coprocessor and as such a VCI initiator. It constantly reads the output queue of descriptors and blocks if this queue is empty. The address contained in the first part of the descriptor tells the memory location of the first slot. The address of the descriptor contained in the first eight byte of this slot tells the address of the second slot, and so forth. A buffer contains the packet during reconstitution. Finally, the packet is written to an

[1]For reasons of readability, Figure 7 shows the writing of liberated addresses to their respective FIFOs only for the last classification task.

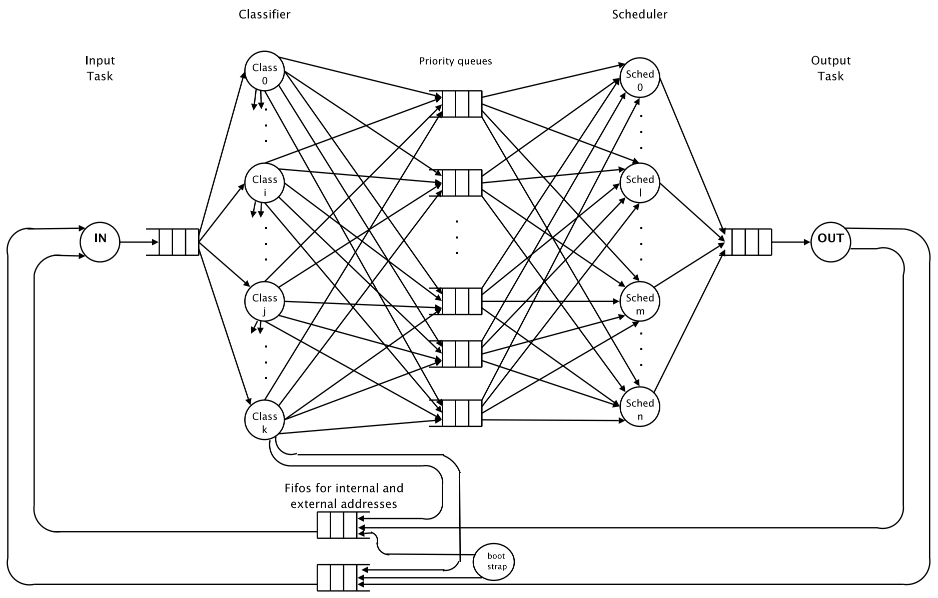

Figure 7. Task graph of the classification application

output file, with optional information on latency and other data useful for performance evaluation. Each time a slot is read and sent to the buffer its released address is sent to either of the two internal or external address FIFOs.

Bootstrap Task This task is responsible for everything having to do with the startup of the application, originally located in the application `main()`. The input task will need addresses to which it can allocate the first and following slots, otherwise the packet treatment cannot start. The bootstrap task is responsible for this; Figure 7 shows that it writes into two FIFOs: addresses of internal slots into the FIFO `slint` and addresses of external slots into the FIFO `slext`. The generated addresses have 128 bytes distance between them, to accommodate one slot.

6 DSX Design Space Explorer

DSX describes the task graph in a completely static manner. The number of tasks is fixed, also the association of channels to ports and their sizes. The management of software tasks which are implemented as POSIX threads created in the `main()` function of embedded code is automatic, the configuration of the MWMR wrappers is also done automatically. Debug messages are filtered through different levels of verbosity. Furthermore, the instantiation of the SoCLib components and the rather error-prone task of connecting all signals to their respective ports in the SystemC netlist is automatically generated and has completely disappeared from the designer's immediate view. The designer provides a descriptions of the hardware, the task graph and the mapping, in the Python language [13].

The following few lines describe most of the mono processor architecture. It includes the description of cached and uncached memory segments. The last five lines describe the interconnection of interfaces between processor and cache and around the VCI interconnect where the cache is initiator and all others are targets.

```
def architecture(self):
  vgmn = Vgmn("vgmn",
    latency = self.getParam('min_latency'))
  cache = Xcache('cache1',
    dcache_lines=64, dcache_words=8,
    icache_lines=1024, icache_words=16)
  mips0 = Mips('mips0')
  cram0 = Segment('cram0',Cached)
  uram0 = Segment('uram0',Uncached)
  reset = Segment('reset',Cached,addr=0xbfc00000)
  excep = Segment('excep',Cached,addr=0x80000080)
  ram0 = MultiRam('ram0',cram0,uram0,reset,excep)
  locks = Locks('locks')
  tty = MultiTty('tty', 'tty0')
  mips0.cache // cache.cache
  cache.vci // vgmn.getTarget()
  ram0.vci // vgmn.getInit()
  locks.vci // vgmn.getInit()
  tty.vci // vgmn.getInit()
```

Each task requires a short description of its *model*: name, ports, and some precisions on implementation. or instance, the *InputEngine* is defined by:

```
input_task = TaskModel('input_engine',
    infifos = ['slint','slext'],
    outfifos = ['desc'],
    impl = [ SwTask('inputengine',
        stack_size=0',
        sources = ['src/input_engine.c'],
        defines = ['FILE'] ),
        HwTask(InputEngine)] )
```

The Input Task reads from the FIFOs connected to the ports `slint` and `slext` and writes to the FIFO connected to the port `desc`. It exists in a software (keyword `SwTask`) and a hardware (keyword `HwTask`) implementation[2].The code of the software tasks has to be supplied by the programmer and indicated where it can be found (here, the `src` subdirectory). The `define` contains file names and constants. MWMR channels are connected to ports, forming the TCG.

```
tasks = Task('ie0', models.input_engine,
  portmap = {'slin':cin,'slext':cext,
             'desc':cdesc},
  defines = {'FILE':'"packets"'}),
```

All *Makefiles* are generated automatically. DSX dimensions the memory space to what really is required by making a second compilation pass after mapping. Finally, if not demanded otherwise, it generates an executable for both a version consisting entirely of POSIX threads and a version with SoCLib models of hardware coprocessors.

DSX provides both blocking and non blocking primitives to access MWMR channels. Non-blocking primitives are required when the scheduling task reads from a priority file if there is a descriptor available. The blocking read and write primitives return when the requested transaction is complete, i e. the requested number of words was successfully read or written, whereas the corresponding non blocking functions will always return an integer that indicates the number of words that have been accessed, even if the request is not satisfied. If the returned number is less than required, it is up to the software task to decide.

In order to distinguish between internal and external address spaces, DSX allows to define *memspaces*. Memspaces are located on one memory segment, either cacheable or uncacheable. *Barriers* ensure task synchronization. Like channels, memspaces and barriers are connected to ports, again their declarations are completely static. Strobe signals for starting up the hardware *InputEngine* and *OutputEngine* are adequately placed in the bootstrap task. These signals go to the MWMR wrapper, their names are deduced from coprocessor names. The bootstrap task runs only once, then suspends, whereas DSX tasks run continuously.

On the one hand the MIPS R3000 does not support multiple contexts, context switching is thus expensive. On the other hand the MIPS R3000 architecture is relatively simple, we can thus afford to add a large number of processors and allocate only one task per processor. For any task, its software objects are mapped to memory banks. For example, all classification tasks can be mapped to their respective processors and memory banks in a loop.

[2]DSX considers hardware coprocessor and software implementation as two implementations of the same *task* with the same functionality and identical interfaces.

```
for i in range(nclassif):
  mapper.map("classif%d" %i,
  run =  hardware.mips[i],
  stack = hardware.cram[i],
  desc = hardware.cram[i],
  status = hardware.uram[i],
  code = hardware.uram[i])
```

Lock, status and description of a MWMR channel or a barrier are equally mapped to memory banks. Note that the memory banks of the lock and the channel appear explicitly in the mapping, which is a significant improvement over SPADE where only tasks can be mapped. Memspaces are mapped to one memory bank each.

7 Validation and Performance Results

All hardware components are described by simulation models from the SoCLib library [16]. A direct mapped write through cache policy is used for both the 16 Kbytes instruction cache and 512 bytes data cache. This article does not aim at design space exploration; the choices were guided by the results obtained in [7]. Performance results (Figure 9) were obtained for a mono processor platform with three memory banks (one for the application code, one for internal slots, one for external slots) running one task of each type, and two multiprocessor platforms running 20 classification, 3 scheduling and the bootstrap task, one task per processor. The first multiprocessor platform has only one level of interconnection network, around which the processors, eight memory banks, the TTY, lock engine and coprocessors are grouped. For the clustered platform with its six processing clusters, the optimal mapping places the three scheduling tasks on the last cluster, together with all priority queues and the output channel (Figure 8). The input file contains 8.000 packets of 54 bytes, which is the minimal size for Ethernet encapsulated IPv4 packets: 40 bytes plus 14 bytes header. This represents the worst case for our type of platform: the number of headers to verify and of packets to classify is the highest possible for the volume of transmitted data.

The graphs and histograms in Figure 9 show the evolution of latencies of packet traversal. For better readability, the left hand diagrams show the evolution of mean latency taken over 50.000 simulation cycles. Packets are timestamped before leaving the *InputEngine* and these timestamps compared to the time they leave for the output file. Measurement begins once the bootstrap task has completed its work; to ensure this, we use barrier synchronization. Performances on the mono processor platform are weak as expected (Figure 9.a). The first packet arrives after 5 million cycles, then latencies are increasing without ever reaching a steady state. The channel between the input task and the classification task quickly overflows, leading to excessive

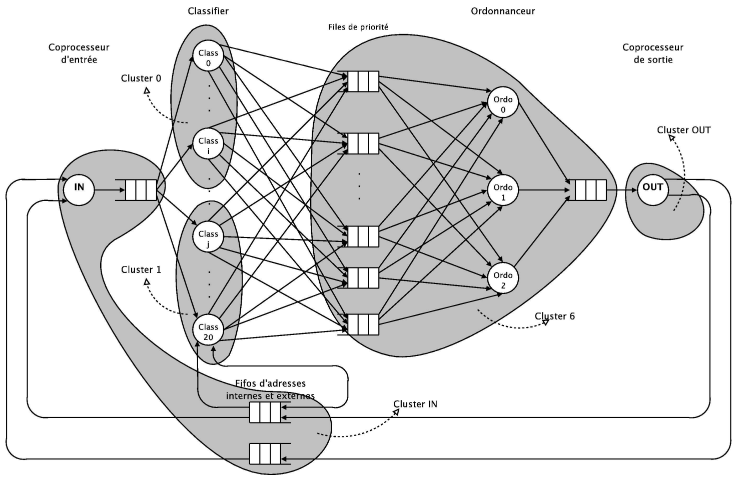

Figure 8. Mapping of channels and tasks on the clustered platform

packet loss (of 8.000, only 1552 arrive). When scaling to 24 processors grouped around a one level interconnect, performances improve as expected, but are still not satisfactory. After reaching a steady state rapidly, they vary considerably around 450.000 cycles (Figure 9.b). Results for the clustered platform (Figure 9.c) are weaker than those obtained in [7]. A possible explanation is that the application code is around 15% larger (statically allocated space for the memory banks not counted) accounting for more misses in the instruction cache. A smaller peak around 230.000 cycles indicates packets for which the effect of NUMA was not compensated, neither do we observe a decrease of latencies after an initial phase.

8 Conclusion and Perspectives

With the help of DSX we achieved within less than a month the deployment of a multi threaded telecommunication application on several variants of our MPSoC platform. Our goal was to prove that DSX can handle task graphs that do not use the KPN semantic, as this kind of graph is often used in telecommunication applications.

Two telecommunication specific hardware coprocessors have been integrated into DSX in a first place. Some specific needs of our application were already met by DSX.

Reading from and writing to a such a large number of tasks efficiently is guaranteed by using MWMRs. Barrier primitives for synchronization and memspaces for placement of slots are part of the original version of DSX. Our application also requires mechanisms which enable tasks to run during a limited time only, whereas DSX tasks run continuously. The integration of a generic bootstrap task into DSX itself, which manages the entire initialization process (starting up coprocessors, initializing variables and filling up channels if required), is currently discussed. Primitives to ensure cache coherency by selective invalidation have been added. The impact of the larger DSX executable on the instruction cache will have to be investigated in more detail. For performance reasons, the code of the tasks should be replicated on each cluster, which is not yet possible.

With these adaptations, DSX will significantly ease the addition of application specific hardware coprocessors such as table lookup and cryptography.

References

[1] I. Augé, F. Pétrot, F. Donnet, and P. Gomez. Platform-based design from parallel c specifications. *CAD of Integrated Circuits and Systems*, 24(12):1811–1826, Dec. 2005.

[2] J.-Y. Brunel, W. M. Kruijtzer, K. Kenter, F. Pétrot, L. Pasquier, K. Kock, and W. J. M. Smits. COSY communication IP´s. In *Proceedings, 37th Conference on Design Automation*, pages 406–409, NY, June 5–9 2000. ACM/IEEE.

[3] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: a framework for simulating and prototyping heterogeneous systems. *Readings in hardware/software co-design*, pages 527–543, 2002.

[4] D. Comer. *Network System Design using Network Processors*. Prentice Hall, 2003.

[5] E. A. de Kock, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, K. A. Vissers, and G. Essink. Yapi: application modeling for signal processing systems. In *DAC '00: 37th conference on Design automation*, pages 402–405, New York, 2000. ACM Press.

[6] M. Drake, H. Hoffman, R. Rabbah, and S. Amarasinghe. MPEG-2 Decoding in a Stream Programming Language. In *International Parallel and Distributed Processing Symposium*, Rhodes Island, Greece, Apr. 2006.

[7] E. Faure. *Communications matérielles-logicielles dans les systèmes sur puce orientés télécommunications (HW/SW communications in telecommunication oriented MPSoC)*. PhD thesis, UPMC, 2007.

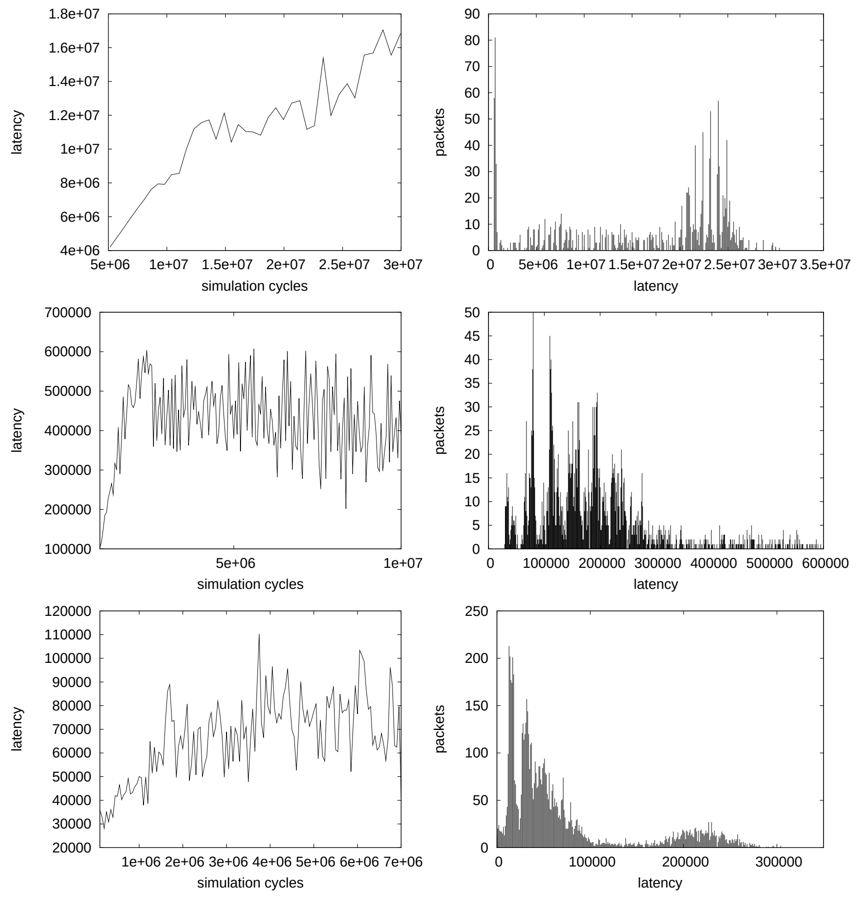

Figure 9. Evolution (left) and histograms (right) of packet latencies: (a) mono processor (b) 24 processor non clustered (c) 6*4 processor clustered platform

[8] E. Faure, A. Greiner, and D. Genius. A generic hardware/-software communication mechanism for Multi-Processor System on Chip, targeting telecommunication applications. In *ReCoSoC'06 : Proceedings of the 2006 conference on Reconfi gurable Communication-centric SoCs*, 2006.

[9] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74*, pages 471–475. North-Holland, NY, 1974.

[10] N. Pouillon and A. Greiner. DSX. URL=https://www-asim.lip6.fr/trac/dsx/wiki/MjpegCourse, 2006-2007.

[11] T. M. Parks. *Bounded scheduling of process networks*. PhD thesis, University of California at Berkeley, CA, USA, 1995.

[12] P. Paulin, C. Pilkington, and E. Bensoudane. StepNP: A system-level exploration platform for network processors. *IEEE Des. Test*, 19(6):17–26, 2002.

[13] Python Software Foundation. Python programming language - official website. URL=http://www.python.org, 1990-2007.

[14] F. Pétrot, P. Gomez, and D. Hommais. Lightweight implementation of the POSIX threads API for an on-chip mips multiprocessor with VCI interconnect. In A. A. Jerraya, S. Yoo, D. Verkest, and N. Wehn, editors, *Embedded Software for SoC*, part 1, chapter 3, pages 25–38. Kluwer Academic Publisher, Nov. 2003.

[15] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol label switching architecture. Technical report, RFC, US, 2001.

[16] SOCLIB Consortium. Projet SOCLIB: Plate-forme de modélisation et de simulation de systèmes integrés sur puce (SOCLIB project: An integrated system-on-chip modelling and simulation platform). Technical report, CNRS, 2003. http://www.soclib.fr.

[17] L. Thiele, S. Chakraborty, M. Gries, and S. Kuenzli. A framework for evaluating design tradeoffs in packet processing architectures. In *Proceedings of the 39th conf. on Design automation*, pages 880–885, NY, USA, 2002. ACM Press.

[18] P. van der Wolf, P. Lieverse, M. Goel, D. L. Hei, and K. Vissers. A mpeg-2 decoder case study as a driver for a system level design methodology. In *CODES '99: Proceedings of the 7th international workshop on Hardware/software codesign*, pages 33–37, New York, 1999. ACM Press.

[19] VSI Alliance. Virtual Component Interface Standard (OCB 2 2.0). Technical report, VSI Alliance, Aug. 2000.