An adaptive MP-SoC Architecture for embedded Systems

Gilles Sassatelli, Nicolas Saint-Jean, Pascal Benoit, Lionel Torres, Michel Robert
LIRMM, UMR 5506, University of Montpellier 2-CNRS, 161 rue Ada, 34392 Montpellier Cedex 5, France
Tel.: (33)(0)4-67-41-85-69-69
Email: {first_name.last_name@lirmm.fr}

Abstract. *Scalability of architecture, programming model and task control management will be a major challenge for MP-SOC designs in the coming years. The contribution presented in this paper is HS-Scale, a hardware/software framework to study, define and experiment scalable solutions for next generation MP-SOC. Our architecture, H-Scale, is a homogeneous MP-SOC based on RISC processors, distributed memories and an asynchronous network on chip. S-Scale is a multi-threaded sequential programming model with dedicated communication primitives handled at run-time by a simple Operating System we developed. The hardware validations and experiments on applications such as MJPEG and FIR filters demonstrate the scalability of our approach and draws interesting perspectives for distributed strategies of task control management.*

1. Introduction

"MP-SOC is not just coming: it has arrived"[1]. The OMAP platform [2] with the recent 3430[2] system is one of many existing designs to date. If MP-SOC [3] is a reality, the increasing number of general purpose or dedicated processors on a single chip brings several issues to address: architecture scalability, programming models, task control management and debug are those referenced in [1].

IP core reuse has driven industrial system designers for obvious productivity and performance reasons. One major drawback is that these solutions are poorly scalable in terms of software and hardware. Although being aware of these economic constraints, we strongly believe that an alternative is possible from a basis of a scalable hardware and software framework.

The work presented in this paper aims at exploring and defining principles which grant both hardware and software scalability. The H-Scale architecture is based on a NPU (Network Processing Unit), which is essentially a programmable RISC processor, a small memory and a routing unit. The communication infrastructure is based on an asynchronous packet switching network-on-chip which allows connecting the NPUs in a mesh-based fashion. The S-Scale is a multi-threaded procedural programming model with communication primitives. Implementations carried out show that the HS-Scale framework guarantees any application to be executed regardless the target platform features (number of NPUs) and the chosen mapping. Moreover, several experiments conducted on thread duplications suggest some strategies to automate the task control management and distribute it over the system.

In the following, Section 2 tackles the issues at stakes regarding MP-SOC design and programming. Section 3 presents and details the intrinsic hardware principles. Section 4 is devoted to the programming model and more generally the software part of the framework which includes the online management mechanisms. Section 5 presents the hardware realizations, particularly the FPGA prototype and its debugging interface and provides some performance figures of this realization. Section 6 gives application results on a FIR filter and a MJPEG decoder.

2. Related works

During the last decades, improvements in microprocessor design and compilers were mainly aimed at improving Instruction Level Parallelism. It has nevertheless been stated that trying to further increase ILP is not the best choice, D. Patterson refers it to as the "ILP wall" [6]. Thread (or Task[3]) Level Parallelism (TLP) enables significant speedups and proves more flexible than ILP. TLP is now supported by a growing spectrum of programming environments through programming models, libraries, etc.

From an architecture point of view, a MP-SOC may be either homogeneous, *i.e.* all the processing elements are the same (*e.g.* for server applications), or heterogeneous (CMP, or Chip Multi Processing, *e.g.* for embedded applications). One typical example of a heterogeneous MP-SOC system is a cell phone (for instance, those based on an OMAP platform). One of the toughest aspects in heterogeneous MP-SOC is that software modules have to interrelate with hardware modules. For instance, the authors of [4] advocate the use of high level programming for the abstraction of HW-SW interfaces. Their programming model is made of a set of functions (implicit and/or explicit primitives) that can be used by the SW to interact with HW. In the reconfigurable computing domain, alternative approaches have also been investigated, as the original one in [5] where a scalable programming model (SCORE) is associated to a homogeneous scalable reconfigurable architecture. The model allows indifferently computing a set of tasks in time or in space, following the resources available: the advantage is that software is reusable for any generation of component based on that model.

There are two major programming models deriving from the memory architecture: SMP (Symmetric Multi Processing) where all the processors have a global vision of the memory (shared memory) and AMP (Asymmetric Multi Processing) where the processors are loosely coupled and have generally dedicated local memory resources. Procedural sequential programming (*e.g.* C) is generally the basis of MP-SOC systems as it stands on compilers that are widely available and because "Everybody knows C…". As MP-SOC provides resources to compute several tasks concurrently, multi-threaded programming models have to be examined. With multi-core processor architectures, libraries such as open MP (SMP model) and MPI (Message Passing Interface) (AMP model) provide an interface to the

[1] Grant Martin, Design Automation Conference, 2006 [1]
[2] Superscalar ARM CortexA8 core, IVA2+ accelerator for H264 video, Image Signal Processor, 2D/3D Graphics accelerator
[3] *Thread* and *task* are interchangeable within the scope of the presented work

programmer for execution directives inside the source code. Using this kind of library is currently not realistic for MP-SOC designs for the overhead they imply.

Task control management is also an issue to consider. Threads can be handled at execution time (dynamically on a single processor) by an operating system or at design time (statically) by complex scheduling techniques. A part of the MP-SOC community focuses on static task placement and scheduling in MP-SOC. Indeed, having a complex operating system in memory taking care of run-time mapping is often not feasible for a SOC, because of the restricted memory resources and associated performance overhead. Moreover, these systems are often heterogeneous and dedicated to a few tasks, and a single but efficient scheduling of tasks may be more adapted. For instance in [7], the authors summarize the existing techniques (ILP based or heuristics) and have developed a new framework based on ILP solvers and constraint programming to solve at design time the task allocation/scheduling problem. There are also contributions for the improvement of local performances in [8] and for energy savings in [9].

3. H-Scale architecture model

3.1. System overview

One of the leading principles of our approach relies in the exploration of massive parallelism for embedded MP-SOC; we target scalability of both hardware and software and expect performance to emerge from multitude and not from the intrinsic performance of the processing tile. For that reason, the main concerns regarding the processing element architecture are flexibility and compactness.

Figure 1 shows a system-level overview of the H-Scale architecture and the surrounding components it is supposed to be connected to. As illustrated in this figure, our contribution is a homogeneous MP-SOC as a component of a heterogeneous system. It is based on a scalable architecture with distributed memory (AMP model). It is made of a regular arrangement of processing elements (PE) interconnected by a packet-switching communication network. As we assume that the HS-Scale architecture is a component of a realistic system, some of the PE of this architecture is responsible of establishing the communications with the rest of the system (interface PEs).

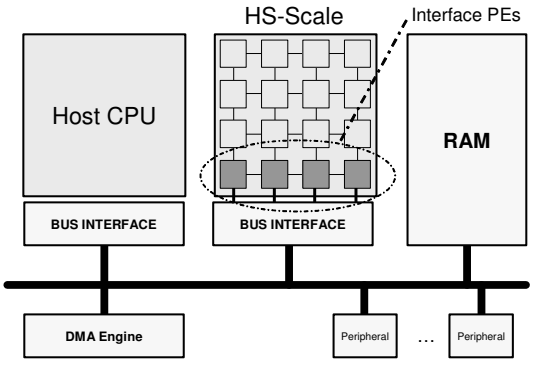

Figure 1: system-level overview

3.2. Network Processing Unit

The architecture we present is made of a homogeneous array of PE communicating through a packet-switching network. For this reason, the PE is called NPU, for Network Processing Unit. Each PE, as detailed later, has multitasking capabilities which enable time-sliced execution of multiple tasks. This is implemented thanks to a tiny preemptive multitasking Operating System[4] which runs on each NPU. The structure of the NPU is depicted in figure 2. It is built around two main layers, the network layer and the processing layer.

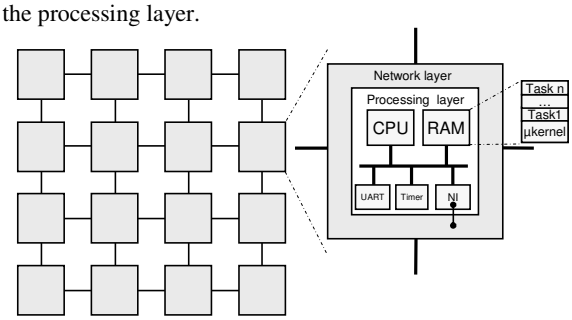

Figure 2. Network Processing Unit

The Network layer is essentially a small routing engine (XY routing). Packets are taken from incoming ports, then either forwarded to outgoing ports or passed to the processing layer. It is compliant with the communication infrastructure presented below. When a packet header (the first flit) specifies the current NPU address, the packet is forwarded to the network interface (NI in figure 2). The network interface buffers incoming data in a small hardware FIFO and simultaneously triggers an interrupt to the processing layer.

The processing layer is based on a simple and compact RISC microprocessor, its static memory (no cache) and a few peripherals (timers, one interrupt controller, UART) as shown in figure 2. A multitasking OS implements the support for time-multiplexed execution of multiple tasks. The microprocessor we use has a compact instruction set comparable to a MIPS-1 [10]. It has 3 pipelines stages, no cache, no Memory Management Unit (MMU) no memory protection support in order to keep it as small as possible.

3. 3. Communication infrastructure

For technology-related concerns, a regular arrangement of processing elements (PEs) with only neighboring connections is favored. This helps in a) preventing using any long lines and their associated undesirable cross-talk effects in deep sub-micron CMOS technologies b) synthesizing the clock distribution network since an asynchronous communication protocol between the PEs might be used. Also, from a communication point of view, the total aggregated bandwidth of the architecture should increase proportionally with the numbers of PEs it possesses, which is granted by the principle of abstracting the communications through routing data in space. The Network-on-Chip paradigm (NoC) enables that easily thanks to packet switching and adaptive routing.

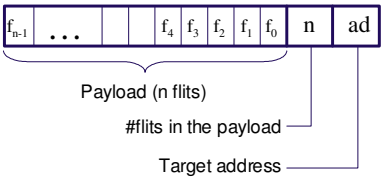

Figure 3: Packet format

The communication framework of H-scale is derived from the Hermes Network-on-chip, refer to [11] for more details.

[4] Operating System are often refered to as microkernels in the area of SOC, mainly because of their very limited memory footprint

The routing is of wormhole type, which means that a packet is made of an arbitrary number of flits which all follow the route taken by the first one which specifies the destination address. Figure 3 depicts the simple packet format used by the network framework constituted by the array of processing elements. Incoming flits are buffered in input buffers (one per port). Arbitration follows a round-robin policy giving alternatively priority to incoming ports. Once access to an output port is granted, the input buffer sends the buffered flits until the entire packet is transmitted.

Inter-NPU communications are fully asynchronous, and are based on the toggle-protocol. As depicted in figure 4, this protocol uses two toggle signals for the synchronization, a given data being considered as valid when a toggle is detected. When the data is latched, another toggle is sent back to the sender to notify the acceptance. This solution allows using completely unrelated clocks on each PE in the architecture.

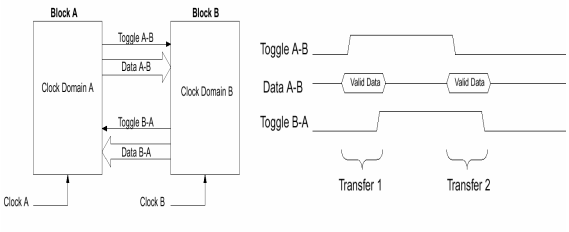

Figure 4: The asynchronous toggle protocol

4. S-Scale programming model

Our goal is to provide a complete scalable solution which assumes first that the architectural model is scalable, but also the programming model. Our model is based on distributed memories (AMP) and allows computing multiple tasks in time (single processor), and in space (multi-processor). For a given application any possible mapping scenario between computing in time and computing in space is supported.

4.1 Multi-Threaded Procedural model, with Communication Primitives

S-scale is a mixed model composed of a Sequential Procedural Programming basis, a Multi-threaded support and Communication Primitives for inter thread communications.

A process is an instance of a program in memory. It consists generally of several functions. When these functions may be scheduled separately, they are called threads. On multiprocessor machines, it is more natural to program applications with multiple threads since they have the possibility to be executed on several processors. The threads in our model are described in C language, the most famous and widely used sequential and procedural language for programming embedded systems.

Since threads may be time-sliced, which means they can run in arbitrary bursts as directed by the operating system, the property of confluence (same result yielded regardless thread execution order) must be guaranteed. The underlying programming style for ensuring the synchronization of the computation in our approach is Kahn Process Networks (KPN) [12]. KPN is a distributed model of computation where processes are connected to each other by unbounded FIFO channels to form a network of processes. KPN can be represented functionally by a Petri net as depicted figure 5.

Reading from a channel is blocking: the single token in the place *P* forbids that the process is executed before the place FIFO IN is filled with data. Writing is non-blocking: when the data has been written to the FIFO OUT, place *P* is filled with its initial marking again allowing new data to be read. A set of communication primitives has been derived from this formalism for ensuring confluence of application execution regardless thread execution order.

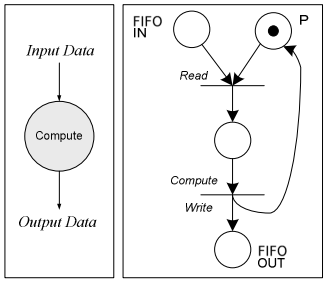

Figure 5. KPN Model of a single task computation

4.2 Communication primitives

They essentially abstract communications so that tasks can communicate with each other without knowing their position on the system (either on the same NPU or a different one). The communication primitives were derived from 5 of the 7 layers of the OSI model as shown on figure 6.

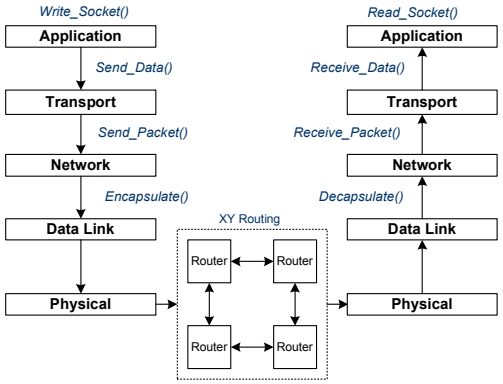

Figure 6. Communication Protocol and Communication Functions on 5 OSI layers

Firstly, communication management between tasks is insured by two dedicated functions. In order to route the packets, these functions use a dynamically updated routing table. *Read_Socket()* and *Write_Socket()* read and write to software FIFO supervised by the operating system. These functions allow transparent data communications between tasks either locally or remotely: the routing is done following this dynamic routing table. When the task is local, the writing of data is done on a local software FIFO. When the task is remote, the operating system must insure that there is enough space for the remote software FIFO to avoid deadlocks on the network. This is done thanks to dedicated functions. As soon as the OS gets a positive answer, he can start encapsulating and sending the data packets to the remote task (*Encapsulate()*, *Send_Data()*) while the remote task can deencapsulate and receive the data packets and write them to its local software FIFO (*Decapsulate()*, *Receive_Data()*).

4.3 Operating System

In order to schedule tasks on a single processor, to handle communications between local and remote tasks with the communication primitives described above, it is necessary to use an Operating System offering these functionalities. After checking the literature and existing embedded OS (uClinux, eCos, etc.), it appeared that our memory restrictions (less than 100kB for data and program on one NPU) were too strong to use these costly solutions.

Therefore, we have developed a lightweight operating system which was designed for our specific needs. Despite being small, this OS does preemptive switching between tasks and also provides them with the communication support for tasks interactions (communication primitives). Figure 7 gives an overview of the operating system infrastructure and the services it provides.

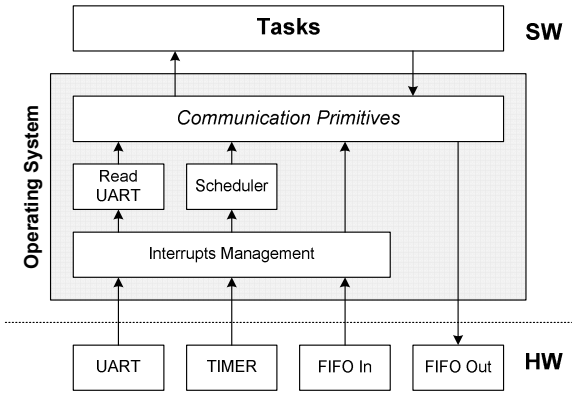

Figure 7: Operating System overview

The interrupts manager may receive interrupts from hardware: UART, Timer and FIFO In. When this happens, it disables the interrupts and save the processor context. Following the type of interruption, it reads from UART, schedules the tasks (timer) or use a communication primitive (interrupt from the FIFO). Afterwards, it restores the processor context and enables again the interrupts. The scheduler is the core of the OS but is quite simple. Each time a timer interrupt occurs, it checks if there is a new task to run. In the positive case, it executes this new task. Else, it has two possibilities: either there is no task to schedule then just runs an *idle* task, or there is at least one task to schedule. This way, each task is scheduled periodically in a round robin fashion (there is no priority management between tasks).

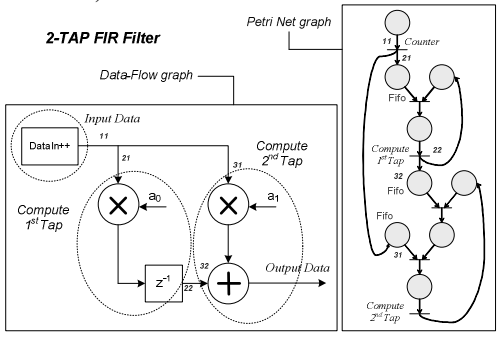

Figure 8: FIR example

4.4 Programming Example

In the following, we give a very simple example on how to program our architecture with the proposed programming model. It shows how from a classical C code of a 2-TAP FIR filter we introduce thread directives based on our communication primitives. This is just an illustration of what a programmer could do with this model and would not make sense "in the real world" to improve the performances of such an algorithm. However, it demonstrates the scalability of the programming model.

The figure 8 illustrates the filter with two representations: the first one is a simple data-flow graph and the other one is a KPN process networks with Petri net.

On a classical processor, the C-code could look like that:

```
int main()
{
  int data_in,r1,r2,r1p=0,a0=1,a1=2,data_out;
  data_in=0;
  while(1)
    {
        r1=data_in*a0;  // Compute 1st tap
        r2=data_in*a1;  // Compute 2nd tap
        data_out = r2 + r1p;
        r1p = r1; //Delay
        data_in++; // data increment
    }
  return 0;
}
```

With our programming model, it is possible to fork the process in 3 different threads as shown below:

```
Type_task thread1(void)
{
  int data_in=0;
  while(1)
    {
        write_socket(21, &data_in, 1, 1);
        write_socket(31, &data_in, 1, 1);

        data_in++; // data increment
  }
  return 0;
}

Type_task thread2(void)
{
  int data_in, r1 ,a0 = 1, zero=0;
  /*Data synchronization*/
  write_socket(31, &zero, 1, 1);
  while(1)
    {
        read_socket(21,&data_in,1,1);
        r1 = data_in * a0; // Compute 1st tap
         write_socket(32, &r1, 1, 1);
    }
  return 0;
}

Type_task thread3(void)
{
  int data_in, r1 ,r2 ,a1 = 2, data_out;
  while(1)
    {
        read_socket(31, &data_in, 1, 1);
        read_socket(32, &r1, 1, 1);
        r2 = data_in * a1; // Compute 2nd tap
        data_out = r2 + r1;
    }
  return 0;
}
```

One can notice that the functions *Write_socket()* and *Read_socket()* are used to establish communication channels between the three tasks. The parameters represent the socket identifiers (21, 31, …) which are used in the routing tables of the NPU, the address of the data block and the number of data in the block.

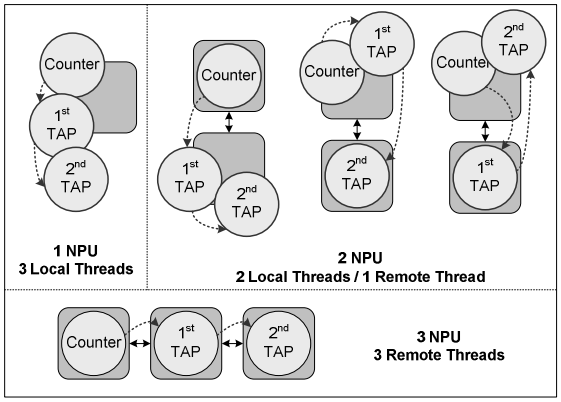

Figure 9: 3 functionally equivalent mappings on 1, 2 or 3 NPUs, with the same C program

By the way, this program can be mapped (statically) to the architecture on a single or multiple NPU indifferently as depicted figure 9, *i.e.* with the same functionality since the communication primitives of the OS ensure task interactions.

5. Validations

5.1. Hardware Prototype

A complete synthesizable RTL level description of the NPU has been developed. It has allowed us to validate the hardware prototype, estimate areas and power consumptions (post place and route, with AMS 0.35µ design kit), and improve the design. Any instance of the H-Scale MP-SOC system may be easily generated with our generic parameters, and then evaluated with CAD Tools (Encounter Cadence flow).

# NPU	*1*	*2*	*4 (2*2)*	*9 (3*3)*
Area (mm²)	18.22	36.63	73.61	165.30
Power Cons. (mW/MHz)	2.56	5.14	10.34	23.26

Table 1: Area and Power consumption[5] scalability

Table 1 summarizes these evaluations. The hardware prototype has been placed and routed with a 64KB local memory, which actually represents in a single NPU 87% of the total area. In the 13% remaining, the Processor represents 54% (1.2 mm²), the router 38% (0.85 mm²) and the rest (UART, interrupt controller, Network Interface, etc.) about 7%. The power consumption has been evaluated thanks to simulation database dump (vcd files) and Cadence tools. It has been then optimised with Gated Clock insertion. The power consumption repartition figure is slightly the same compared to the area. The table above clearly shows the scalability of area and power consumption of our H-Scale System (the very low overhead is due to the wires needed to interconnect the NPU).

5.2. FPGA Prototype

RTL Simulations are too slow for significant applications such as MJPEG performed on streams of data. We decided then to use a Xilinx Development Kit to synthesize and validate our design. 6 NPUs could be fitted on a XC2VP30 FPGA from Xilinx. A NPU occupied 2151 slices on the FPGA which is rather small.

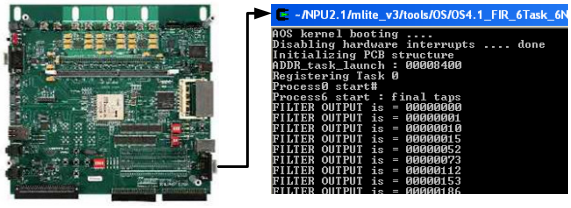

Figure 10: FPGA prototyping board and its debugging interface

Debugging on the prototype takes place thanks to a UART interface between one interfacing NPU and the workstation; some additional services were added to the NPU kernel for feeding back debugging information directly to the PC (figure 10).

5.3. Software Tool Chain

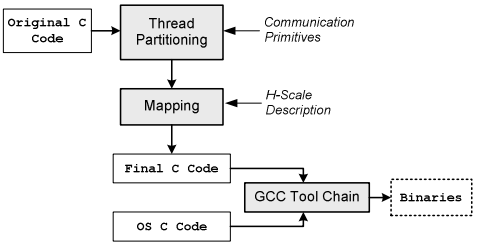

Figure 11: Software flow

Figure 11 depicts the software flow used in our framework to port an application from its original C-code to the HS-Scale MP-SOC. A thread partitioning is first done by the programmer. This can be helped by the original procedure partitions (function calls) and profiling tools. The communication primitives are then used to elaborate the communications between the threads. Then, a hand-made mapping of each thread is performed on the H-Scale instance and a routing table is derived. The final C-code is composed of each thread C-code, and then is compiled with the OS C file, allowing thus generating the binaries to load into the memories of the NPU.

OS Min. Time (cycles)	*OS Max. Time (cycles)*	*Communication Primitives (KB)*	*Total OS Size (KB)*
325	373	2.73	5.75

Table 2: Operating System Time and Memory costs

Table 2 is provided to give an overview of the overhead issued by our Operating System. In terms of time penalty, each time the OS is invoked (each time an interrupt happens), it requires between 325 and 373 cycles to perform its job. The effective time penalty regarding applications performances will be analysed in the next section. In terms of memory overhead, it requires 5.75 KB, which represents less than 10% of the 64KB memories we used in our experiments. The communication primitives represent almost half of the total memory required by our OS.

6. Application Results

6.1. FIR

In order to evaluate the overhead introduced by the OS we carried out some experiments for the FIR application. This application has a very high communication over

[5] Average power consumption performed on NPUs running the OS and several tasks

computation ratio which reveals a much too fine task granularity. Hence, both the kernel scheduler and its communication primitives are highly solicited and therefore tend to slowdown the computation. Table 3 shows the FIR performance results for different task mappings, with and without operating system. The results are given for the processing of 10.000 input samples.

	w/o OS	*With OS*		
# NPU	1	1	1	3
# Threads	1	1 Local	3 Local	3 Remote
Cycles	550634	553992	5982187	2036976
Tp (MB/s)	**7.09**	**7.05**	**0.669**	**1.92**

Table 3: Throughput (Tp) Performance of a 2-TAP FIR Filter

Comparing the results of the two first columns of Table 3 shows that both in terms of processing time and Throughput (Tp) the overhead remain below 1%. As expected, when the FIR algorithm is split into several tasks running sequentially on the same NPU, the communication overhead highly degrades the performance (column 3). Distributing the processing among several NPUs (column 4) shows the benefit of using task-level parallelism; without however matching the performance of the single task implementation.

6.2. MJPEG

In order to evaluate the performance of HS-Scale for realistic applications, we have implemented a MJPEG decoder. We naturally chose to use a traditional task partitioning as depicted in figure 12 with both a task-level dataflow description and a functional Petri net equivalent. The first step of the processing is the inverse variable length coding (IVLC) which relies on a Huffman decoder. This processing time for that task is data dependent. The two last tasks of the processing pipelines are respectively the inverse quantization (IQ) and the inverse discrete cosine transforms (IDCT). The atomic data transmitted from task to task is a 8x8 pixel block which has a size of 256 bit.

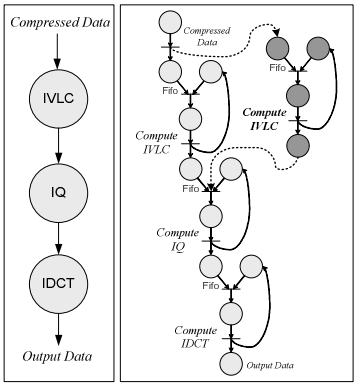

Figure 12: MJPEG Data-Flow and Petri Net Representation

a. From Simple pipeline implementation to multi-threads

Table 4 summarizes the performance figures obtained for several implementations of the MJPEG decoder. Similarly to the FIR implementation, the operating system communication primitives induce a performance overhead when the decoder is splitted into 3 tasks (Table 4, column 2). Distributing the processing on 2 NPUs (Table 4, column 4) immediately pays nevertheless with a significant increase in the throughput. The fully distributed implementation exhibits no performance improvement, which is due to the fact that the critical task in the processing pipeline already fully employs the processing resources of a given NPU.

	w/o OS	*With OS*			
# NPU	1	1	1	2	3
# Threads	1	1 Local	3 Locals	2 Locals, 1 Remote	3 Remotes
Tp (KB/s)	**229**	**228**	**161**	**244**	**246**

Table 4: MJPEG Throughputs (Tp) comparisons

b. From multi-threads, to thread Replication

Many applications such as dataflow applications present tasks that exhibit different and potentially time-changing computational loads over time. Data compression algorithms for instance always feature a variable-length coding task that can be very demanding in performance depending on processed data. In such scenarios, allocating hardware resources at run-time may help better meeting performance requirements without the traditional over-dimensioning problem of static allocation. The principle developed in this section relies in a multi-graph description of the same application; the processors are then responsible to switch from one graph to another depending on run-time requirements.

Figure 13.a depicts a synthetic task graph. A profiling may show that task2 is (i) the most demanding and (ii) exhibit data-dependent computational load. Replicating it helps in increasing the performance which would lead to the scenarios depicted on Figure 13.b and Figure 13.c. In such cases, of course all three instances of task 2 would be hosted on a dedicated processor. The experiments conducted implement the automated replication strategy based on a multi-graph description of the application. Strategies enabling run-time replication may either be simple (fork() and join() in this case) or more difficult, therefore requiring programmer attention.

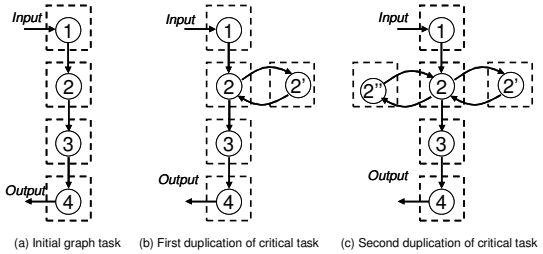

Figure 13: Initial application task graph(a) and replication of task 2 (b and c).

In the case of the MJPEG application the replication is permitted by the absence of inter-block dependencies in the MJPEG application Table 5 shows the performances achieved for replications of some tasks. As clearly suggested by the throughput values, the IVLC task is the critical one in our case since duplicating and triplicating it (columns 5 and 6) significantly increase the performance. Allocating a fourth NPU to this task does not further improve the performance meaning that another task then became the critical step in the processing pipeline.

	w/o Duplic.	*IQ Duplic.*	*IDCT Duplic.*	*IVLC Duplic.*	*IVLC Triplic.*
# NPU	3	4	4	4	5
# Threads	3 Remote	4 Remote	4 Remote	4 Remote	5 Remote
Tp (KB/s)	**246**	**220**	**241**	**332**	**432**

Table 5: MJPEG comparisons with or without thread duplications

c. Load balancing

As mentioned previously, many applications feature highly asymmetric computational load for their constituting tasks. Similarly to the process explained above, where demanding tasks are replicated, we have statically observed the potential benefits of merging several sub-critical tasks onto the same processor. This results in time-sliced execution of those tasks. Figure 14 schematically explains that principle, where task2 fully exploits the processing resource of a given processor (since it has been identified as critical) and task3 and task4 are executed onto a single processor.

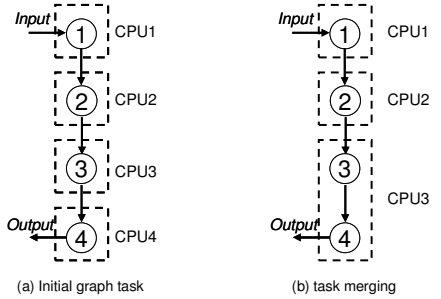

Figure 14 – Principle of load balancing.

Implementing such a mechanism at run-time implies to migrate tasks; which translates in a performance over-head directly proportional to the task code and state to migrate.

The testbench used in this case is similar in every respect to the previous one but a single additional FIR task was moved from NPU to NPU. Table 6 shows the corresponding results, where one can see how much the performance of each application is affected depending on the task mapping. Assigning the FIR filter to the NPU hosting the critical IVLC task (column 3) yields to a significant slowdown of both applications (40% for the FIR, 46% for the MJPEG). Contrarily, assigning the FIR task to one of the other NPUs hosting the sub-critical tasks marginally affects the performance of both applications (columns 3 and 4). The FIR was also assigned for IVLC-duplicated versions of the MJPEG decoder; likewise a similar slowdown is observed only when the FIR is assigned to one of the NPU hosting a critical task.

# NPU	*1*	*3*	*3*	*3*
# Threads FIR	1 Local	1 with IVLC	1 with Iquant	1 with IDCT
Tp (MB/s)	**3.8**	**4.0**	**5.7**	**6.0**
# Threads MJPEG	1 Local	3 Remote	3 Remote	3 Remote
Tp (KB/s)	**122**	**134**	**245**	**246**

Table 6: MJPEG and FIR simultaneous execution performance results

7. Conclusion and Perspectives

A scalable hardware and software framework has been proposed and detailed throughout this paper. Based on a regular arrangement of homogeneous processing units endowed with multitasking capabilities our architecture is capable to support an almost unlimited number of task mapping combinations. The tiny and efficient OS combined to the packet-switching communication architecture we use gives the programmers a huge flexibility at reasonable cost. We have highlighted through some examples that in the case of multiple applications, some mappings may allow to map additional applications at almost no cost.

Regarding the performances obtained in the results section, our current work aims now at extending the OS functionalities to automate the task mapping, migration and duplication (dynamic and continuous task mapping) for achieving run-time adaptability.

8. References

[1] Grant Martin "Overview of the MPSoC Design Challenge**",** Proceedings of the 43rd annual conference on Design automation, San Francisco, USA, 2006

[2] OMAP, Texas Instrument Technology, http://www.omap.com

[3] Ahmed A. Jerraya and Wayne Wolf (editors), M*ultiprocessor Systems-on-Chip*, Elsevier Morgan Kaufmann, San Francisco, California, 2005

[4] Ahmed A. Jerraya, Aimen Bouchhima, Frédéric Pétrot, "Programming Models and HW-SW Interfaces Abstraction for Multi-Processor SoC", Proceedings of the 43rd annual conference on Design automation, San Francisco, USA, 2006

[5] Caspi E., Chu M., Huang R., Weaver N., Yeh J., Wawrzynek J., and A. DeHon, "Stream Computations Organized for Reconfigurable Execution (SCORE)", *FPL'2000*, LNCS 1896, pp. 605-614, 2000

[6] David A. Patterson, "Future of Computer Architecture", Berkeley EECS Annual Research Symposium (BEARS), College of Engineering, UC Berkeley, US, February 23, 2006

[7] M. Ruggiero, A. Guerri, D. Bertozzi, F. Poletti and M.Milano, "Communication-Aware Allocation and Scheduling Framework for Stream-Oriented Multi-Processor Systems-on-Chip", IEEE Design Automation and Test inEurope, Munich, Germany, March 2006

[8] M.T. Kandemir, G. Chen, "'Locality-Aware Process Scheduling for Embedded MPSoCs,"' Proceedings of DATE, pp. 870–875, 2005

[9] Hu J., Marculescu R., "Energy-Aware Communication and Task Scheduling for Network-on-Chip Architectures under Real-Time Constraints." DATE 04, 234-239

[10] MIPS corp., http://www.mips.com

[11] Moraes, F. G.; Mello, A. V. de; Möller, L. H.; Ost, L.; Calazans, N. L. V.. "HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip.", Elsevier Integration, The VLSI journal / Special issue: Networks on chip and reconfigurable fabrics, Vol. 38, Number 1, pp 69-93; Oct. 2004.

[12] G. Kahn, "The semantics of a simple language for parallel programming", *Information Processing*, pages 471-475, 1974