# Adaptation Strategies in Multiprocessors System on Chip

Remi Busseuil, Gabriel Marchesan Almeida, Luciano Ost, Sameer Varyani, Gilles Sassatelli, and Michel Robert

Laboratoire d'Informatique, de Robotique et de Microelectronique de Montpellier (LIRMM), Universite Montpellier 2, CNRS, France
{Remi.Busseuil,Gabriel.Marchesan,ost, Sameer.Varyani,Gilles.Sassatelli}@lirmm.fr
http://www.lirmm.fr

**Abstract.** Multi-processor System-on-Chips (MPSoCs) have become increasingly popular over the past decade. They permit balancing performance and flexibility, the latter being a key feature that makes possible reusing the same silicon across several product lines or even generations.

This popularity makes highlight on new challenges to deal with the increasing complexity of such systems. Programmability issues, for example, are considered with a lot of attention, as those architectures allow new dimensions in design exploration. In this context, the development of adaptable mechanisms permits the optimization of the system behavior. This chapter explors three different adaptable mechanisms, and shows their benefits : frequency scaling, task migration techniques and memory organization. The modification of the frequency of each processor of a multi-core system allows fine tuning of power consumption under a varying process workload. Task migration permits balancing load among the several processors the system is made of. Our long-term vision of future embedded devices lies in adaptive systems made of thousands of processors in which tasks frequently migrate in response to the system state that is continuously monitored. Memory organization is a crucial criterion in MPSoC performance optimization, as memory access latency of remote data increases exponentially with respect to the number of cores. The computation-based programming model commonly used in single-core or few-cores based systems are no more suitable, and a transaction-based model are necessary to reach performances needs of new multimedia application.

**Keywords:** MPSoC, Adaptation, Memory Architecture, Network-on-Chip.

## 1 Introduction

Multiprocessor Systems-on-Chips (MPSoCs) are commonly adopted in electronic industry for their power efficiency and performance capabilities [1]. MPSoCs are multidisciplinary systems that combine multiple homogeneous/heterogeneous

J.L. Ayala, D. Atienza, and R. Reis (Eds.): VLSI-SoC 2010, IFIP AICT 373, pp. 233–257, 2012.

processing elements (PEs), dedicated hardware (e.g. digital signal processing, DSP) and software (e.g. operating systems) in order to cope with different applications requirements (e.g. real time deadlines, throughput). Due to the intensive and parallel communications inherent to embedded applications, networks-on-chip (NoCs) are employed since their are more scalable, flexible and power efficient when compared to traditional on-chip infrastructures (e.g. shared busses) [2] [3].

Given the great dynamism imposed by user-requests and internal system mechanisms (e.g. temperature control), embedded applications can present unpredictable behavior [4]. Such dynamism leads in workload and communication patterns variation, which demands run-time techniques that can provide the necessary adaptability to the system in order to optimize the resource utilization while maintaining high performance [5].

Therefore, the deployment of new techniques to achieve runtime system adaptability is mandatory [6]. Distributed DVFS (dynamic voltage and frequency scaling), power gating, dynamic task mapping, task migration, are examples of runtime techniques that make it possible to optimize various parameters such as application performance and/or power consumption. Dynamic Frequency Scaling (DFS) is a widely used technique aimed at adjusting computational power to application needs. It is often associated to Dynamic Voltage Scaling (DVS) therefore enabling to achieve significant power reductions when computing demand is low; some cited benefits also comprise the reduction of thermal hotspots that participate in the accelerated aging of the circuits due to the thermal stress.

Static and dynamic mapping have been used to define the placement of each task , aiming to reduce the communication latency and energy dissipation inside the NoC. Static mapping defines task placement at design time, while in dynamic mapping tasks are allocated onto PEs at the execution time [7]. However, during the execution the task's performance may degrade due to e.g. higher PE load or NoC congestion. In this case, tasks may be re-mapped to other PEs to meet application requirement. Dynamic re-mapping employs both mapping (static and/or dynamic) and task migration. The task migration process consists in extracting the state of a given task that is already executing on the source PE and transferring it to a destination PE that will execute it. Task migration requires migration points, context saving, context restoring, among other actions not handled by the task mapping mechanisms (e.g. updating the connections of the migrated task with other tasks on the communicating PEs) [8].

Some challenges related to the adoption of task migration into NoC-based MPSoCs still remain. For instance, the task migration performance depends on run-time events that are considered before and during the relocation of given task when it is necessary [9]. Thus, the performance overhead of task migration cannot be very high. The migration execution cost should be low since hard real time deadlines must be met, while maintaining low power dissipation [9] and temperature [10]. Thus, the designer should find a good trade-off between complexity and performance (e.g. implementation and execution), as well as

transparency and reusability, which are directly related to the level that the migration process will be provided (e.g. user, kernel levels) [11]. In this context, an important aspect inherent to the task migration process is the adopted memory organization, which can be classified in shared (Figure 1 - a) and distributed (Figure 1 - b)[8]. In the first case, all PE are entitled to access any location in the shared memory, thus the migrating a task comes down to electing a different processor for execution. In turn, distributed memory MPSoCs, both process code and state have to be physically migrated from a processor local memory to another, while synchronizing exchanged messages [12].

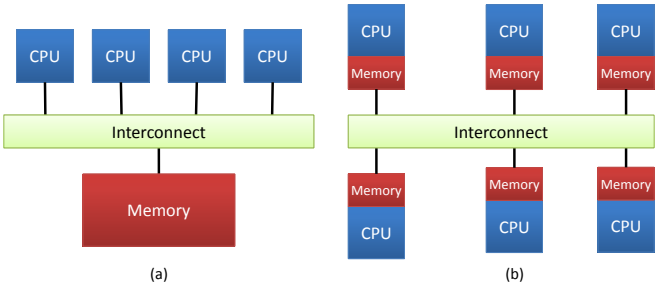


**Fig. 1.** Memory Organization: (a) Shared Memory, (b) Distributed Memory

Both approaches differ in terms of how to access and transfer the task code from one point to another, which has a considerable impact in terms of performance. In this context, this article presents three main contributions: (i) a Dynamic Frequency Scaling strategy using feedback controlers to meet a streaming application throughput requirement,(ii) validation of two task migration mechanisms into an RTL-modeled NoC-based homogenous MPSoC, leading to accurate results, and (iii) a hybrid memory architecture that can be employed to optimize the performance of task migration mechanisms.

This chapter is organized as follow: Section 2 gives an overview of adaptable mechanisms already developed in the literature. Section 3 explains two communication strategies during task migration allowing different kind of performances. Section 5 gives an overview of standard memory models in MPSoC, and then presents a hybrid memory organization allowing more liberty in term of memory placement. Finally, the last section concludes about the different adaptation strategies.

## 2 Adaptation Techniques in NoC-Based MPSoCs

Recently, researchers have put focus on adaptation techniques in order to cope with dynamic and unpredictable behaviors that can appear in nowadays embedded systems. This section presents some work that has been conducted in this direction using techniques such as (i) dynamic voltage and frequency scaling (DVFS), and (ii) task migration mechanisms.

### 2.1 Dynamic Voltage and Frequency Scaling

Numerous dynamic voltage and frequency scaling (DVFS) techniques have been proposed, aiming to allow systems to fine-tune their performance/power consumption trade-off under varying workloads during runtime. Two main approaches to DVFS can be found in the literature. One of them takes a centralized view of power management and considers the global system state, its performance constraints and its current workload to decide voltage and frequency values it will adopt next [13] , while another partitions the system in multiple voltage/frequency islands that can be managed separately [14]. In both centralized and partitioned approaches, the state change (i.e. change of frequency and voltage values) can be triggered by events (e.g. a given threshold on processor utilization was reached [15]) or be driven by a feedback control process that continuously monitors the system workload [16]. The partitioned approach is a natural match to NoC-based multicore chips, but a number of issues arise when different parts of the NoC are on different voltage/frequency islands [17] [18].

For instance, in [19] a DVFS controlling scheme is proposed for NoC-based MpSoCs. In turn, the works [20] and [21] employ communication load as monitoring parameters, which are extracted at run-time in order to tune the DVFS controller. In [22] temperature and task synchronization are used as parameters. In this work, these parameters are employed in a cost function, which is used as input to a game theory model that defines the voltage and the frequency levels. This work was extended in [23], to use communication queues that are placed between each 2 neighbor routers in order to scale the voltage via power supply networks. In [24] the Authors use application profile as the parameter used to control DVFS decisions.

Due to the dynamic variations in the workload of MPSoCs and its impact on energy consumption, PID-based control techniques have been used to dynamically scale the voltage and the frequency of processors [16] [25], and recently, of NoCs [18][26].

A proportional-integral-derivative controller (PID controller) is a generic control-loop feedback mechanism (controller) widely used in industrial control systems. A PID controller calculates an error value as the difference between a measured process variable and a desired setpoint. The controller attempts to minimize the error by adjusting the process control inputs. In the absence of knowledge of the underlying process, a PID controller is a suitable controller [27]. However, for best performance, the PID parameters used must be tuned according to the nature of the process to be regulated.

The proportional, integral, and derivative terms are summed to calculate the output of the PID controller. Defining *u(t)* as the controller output, the final form of the PID algorithm is:

$$u(t) = MV(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{d}{dt} e(t) \tag{1}$$

Proportional gain, $K_p$ : larger values typically mean faster response since the larger the error, the larger the proportional term compensation. 2) Integral

gain, $K_i$: larger values imply steady state errors are eliminated more quickly. 3) Derivative gain, $K_d$: larger values decrease overshoot, but slow down transient response and may lead to instability due to signal noise amplification in the differentiation of the error. Figure 2 summarizes a traditional PID controller.

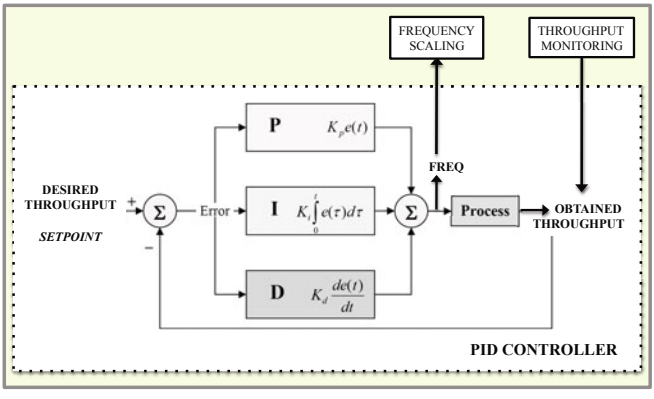


**Fig. 2.** PID Controller

Due to its features, we propose the usage of a PID controller for adjusting the appropriated frequency of the PEs at the same time as deadline miss ratio is reduced. As most applications in embedded systems are based on soft- real time constraints, actual architectures have to be capable of adapting to avoid situations where deadlines are missed. For that reason, the proposed approach considers applications requirements aiming to provide QoS (Quality-of-Service). As entry point (setpoint in Figure 2), the system is fed with the application requirements, e.g. the minimal throughput the application requires to ensure the functionality of the system in a reliable way.

In the proposed approach [28], the system calculates an error value which is obtained by the difference between the desired and obtained throughput. As output of the PID controller a frequency value is indicated. This value is sent to the frequency scaling module which will be responsible for scaling up and down the frequency of the processor to cope with application requirements. The procedure is then repeated and the obtained throughput gradually gets closer to the desired throughput. This is explained by the fact that after each iteration the error value is reduced assuming that the values of P, I and D have been correctly chosen. Figure 3 presents an abstract system model of the proposed strategy for each Network Processing Element (NPU) in the architecture.

Basically each running application is composed of one or multiple tasks. Task are monitored by a throughput monitoring that is responsible for calculating tasks performance in a non- intrusive mode. This information is passed to the PID controller which will be responsible for choosing the appropriated frequency in order to speed up or slow down processing. This strategy can be suitable for power saving in embedded systems.

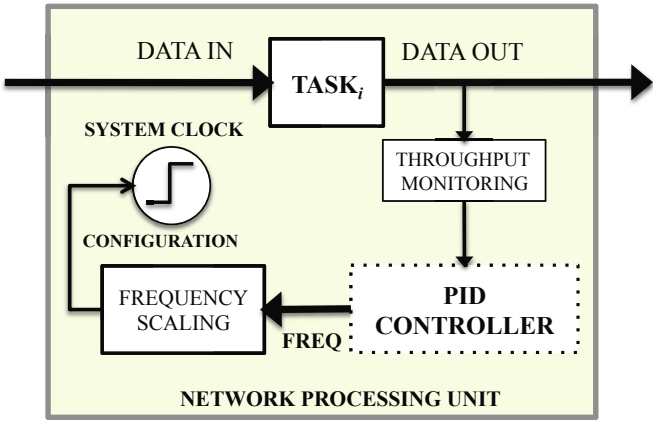


**Fig. 3.** System Model

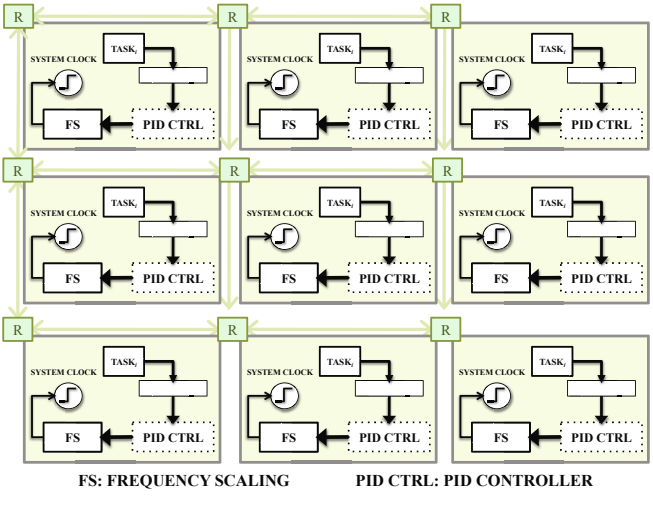


**Fig. 4.** Distributed PID Controllers

Figure 4 illustrates an overview of the proposed approach. As previously mentioned, there is one PID controller devoted to each task in the system that must ensure soft-real time constraints. In this example there is one task per NPU, so one PID controller for each processor is required. In the case that there are multiple tasks in the same NPU, we could build a system with multiple PID controllers in the same NPU, each one being responsible for contributing as a factor that will be added to the final result.

The strategy consists in deciding controller parameters on a task basis. To this purpose, a simulation of the MPSoC system is executed in order to obtain

the step response. Figure 5 shows the cycle-accurate simulation results and the first order extracted model that is used for the process (Figure 2). Based on that high-level model, a number of different configurations of controllers can be explored, each of which exhibits different features such as speed, overshoot, static error.

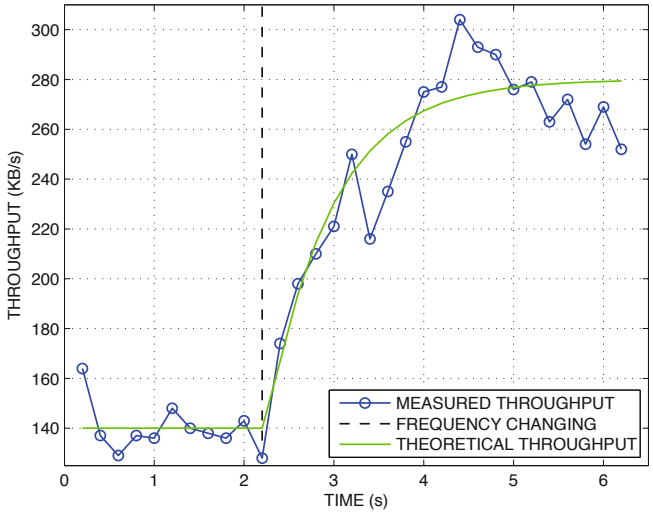


**Fig. 5.** Obtained Throughput *vs* Theoretical Throughput

The values of P, I and D have been chosen to increase the reactivity of the system. Figure 6 presents the PID response according to P, I and D values.

Assuming that the setpoint of the system is around 260KB/s we can observe that in the first case where P = 500, I = 50 and D = 0, the system converges to the setpoint throughput rapidly. As result we observe an overshoot in terms of performance. In the second scenario where P = 800, I = 5 and D = -0.8, due the fact that the value of P is much bigger than I, we can also observe an overshoot in terms of performance.The system throughput presents a high oscillation due to the small value of I. At least, when P = 50, I = 15 and D = -3 we see that system throughput increases slowly. This is explained by the fact that the value of P is very small and then the convergence time is longer. Based on this information we have chosen to use the first controller, because it converges to a stable system relatively fast.

## 2.2 Task Migration Support in MPSoC Architecture

Task migration techniques have been widely explored in the literature, notably in the domains of Graphic Processing Unit (GPU) computing and High Performance Computing (HPC).

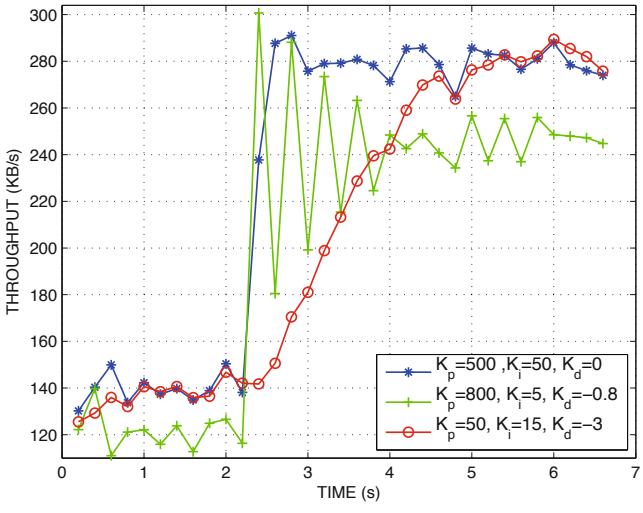


**Fig. 6.** PID Controller Response

Those techniques allows efficient load balancing strategies aiming to satisfy different system requirements. For instance, Streichert et al. [35] employ task migration to keep the correct execution of an application by migrating executing tasks from a PE that presents a fault (stopped working) to another non-fault PE. The architecture details are not presented in this work, although authors claim that this approach can be applied to current FPGA architectures.

As mentioned before, according to the memory organization task migration can be classified in shared memory and distributed memory designs.

**Task Migration in Shared Memory Systems.** Most of today's off-chip multicore systems rely on shared memory architecture. In such cases, the task migration is facilitated by the fact that no data has to be physically moved in the structure (see Figure 1-a): since all cores can access any location in the shared memory, task migration can be done just be electing another CPU to execute the task. Several efficient implementations on general purposes OS, such as Windows or Linux, exist in the literature [36].

Other developments closer to MPSoC have been made, notably based on locality considerations [37] for decreasing communication overhead or power consumption [38]. In [39] authors propose a scalable shared memory MPSoC architecture with global cache coherence. The architecture is built around 4096 cores, which uses a logically shared physically distributed memory with cache coherence enforced by hardware. In [40], authors present a migration case study for MPSoCs that relies on the $\mu$Clinux operating system and a checkpointing mechanism. The system uses the MPARM framework, and although several memories are used, the whole system supports data coherency through a shared memory view of the system.

**Task Migration in Distributed Memory Systems.** The main issue of implementing task migration in shared memory MPSoCs is the scalability of the system. There is a strong tendency for the next generation of homogeneous MPSoC in using systems with distributed memory targeting scalable and massively parallel architectures [41][42].

A number of work in the literature based on distributed memory systems has been implementing task migration in shared memory [8][43] [44]. In [43] each PE runs a single operating system (OS) instance in its logical private memory. PEs execute tasks from their private memory and explicitly communicate with each other by means of shared memory. The target platform uses a shared bus as interconnect.

In the case of distributed memory MPSoCs, both process code and state have to be migrated from a processor private memory to another, and synchronizations must be performed using exchanged messages. While this proves straightforward in typical general-purpose computers thanks to the presence of a memory management unit (MMU), implementing task migration on tiny MMU-less embedded processors is challenging [8].

In [45] a dynamic task allocation strategy is proposed. The work evaluates task allocation strategies based on bin-packing algorithms in the context of MPSoCs. The mechanism adopted is based on a copy model. The whole context (code, data, stack, and contents of internal registers) is migrated and there is no task execution during the transfer. The interprocessor communication is based on send/receive primitives. However, in this work neither explanation about the task migration protocol nor the impact in term of performance of such mechanism is given.

Taking into account the future homogeneous MPSoC systems trend, scalable architectures with purely distributed memory system are required. However, to the best of our knowledge, no purely distributed memory architecture that enables task migration without using shared memory are present in the litterature. Furthermore, very few information are available about migration strategies inside distributed memory architecture, and no migration technique exploration has been done. In the Section 3 of this article, two migration techniques in a complete distributed memory system are presented.

## 3 Task Migration Techniques in Purely Distributed MPSoC

### 3.1 Task Migration Protocols

In [46], we have proposed a new algorithm capable of supporting task migration at run-time. Migration decisions were taken with local information and in a distributed way, each processing unit taking care of its own tasks. This paper puts focus on communication during task migration, stressing the reliability and performance issues raised by such procedure. It presents two communication techniques stressing the benefits and performance penalty of each.

**Hardware and Software Support for Task Migration.** The initial platform used to implement our concepts consists of an homogenous array of processing elements. Those elements are distributed on a 2D-mesh Network on Chip. For this reason, they are called Network Processing Unit (NPU). Each NPU is composed by two layers: a network layer used to route the packets into the network, and a processing layer. Figure 7 shows the architecture of the platform.

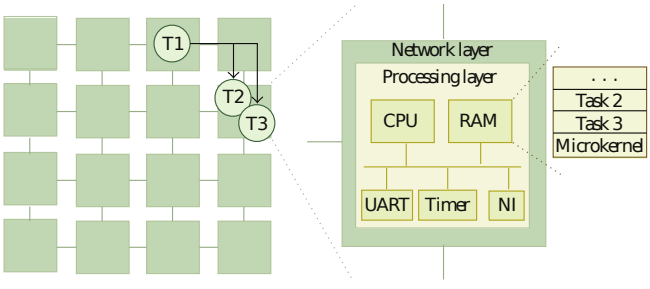


**Fig. 7.** Structural view of the platform

The network layer consists of a light router based on the Hamiltonian Routing Algorithm [47]. The Network on Chip is derived from [48]. It provides packet switching routing with unique predictable route for each packet from the same sender and receiver. Hence, neither reordering nor acknowledgment is necessary.

The processing layer is composed by a compact RISC processor with a MIPS-I instruction set [49]. It also includes some peripherals like RAM, a UART, a timer and an interrupt controller connected through a bus. However, no Memory Management Unit, no cache and no memory protection support are provided to keep the NPU as small as possible. A multitasking real-time preemptive micro kernel runs on each NPU, providing all the asynchronous operating system features required (multi task management, exception handling, communication, etc.). Both the software and the hardware part of this layer comes from [50].

Communications are handled through hardware and software FIFO: incoming data coming from an external NPU are buffered into a small hardware FIFO located in the network layer, triggering simultaneously an interrupt to the processing layer. It activates data demultiplexing from the hardware fifo to the software FIFO linked to the appropriate receiver task. In case of communication between two tasks on the same NPU, data go directly to the software FIFO using an exception process of the operating system.

In a programming point of view, two functions are provided: *MPI_Send()* and *MPI_receive()* derived on the Message Passing Interface model [51]. In compliance with the Khan Process Network (KPN) model of computation, the send function is non blocking while the receive function is blocking. As we have a message passing programming model, we do not need any memory synchronization.

To handle communication in such globally asynchronous system, each NPU possesses a routing table with all the tasks running locally, as well as the position of all predecessors and successors tasks. Thus, the position of every task the operating system needs to communicate with is known. A master NPU (named NPU11) keeps tracks of every tasks, successors and predecessors in the entire chip. If a task requires opening a new communication channel, the Operating System will modify its own routing table and the one of the master NPU. If the position of the task to communicate with is not know, a query is sent to the master NPU.

**Non Continuous Communication.** The first migration protocol is depicted in Figure 8. We consider two tasks hosted on two different NPUs communicating together: task 1 is the sender whereas task 2 is the receiver. Task 2 begins a migration procedure. First, the NPU hosting task 2 will remove its entry in the routing table of its predecessor node, as well as in the master node (step 1). As these nodes do not possess task2 position anymore in their routing table, the communication between task 1 and task 2 will stop. Task 2 can now migrate to its destination NPU (step 2).

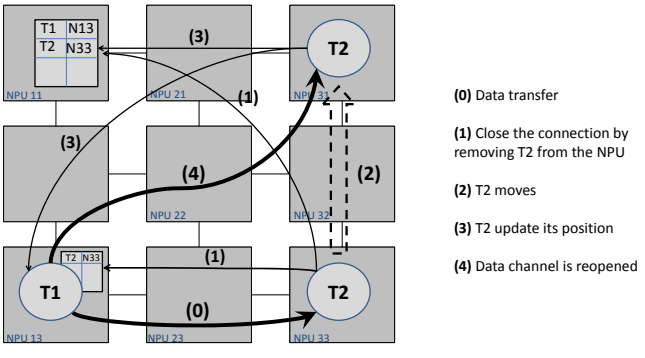


**Fig. 8.** First migration protocol

It is important to notice that during the transfer of the task, the communication channels are closed downstream and upstream, which translates into buffering.

Finally, when the code has arrived in the destination NPU, the task is scheduled and the master routing table as well as the predecessor routing table are updated (step 3). Hence, communication can be resumed (step 4).

Figure 9 shows the datagram of the protocol. No packets are emitted during the migration.

**Continuous Communication with Rerouting.** The second protocol used for task migration was designed with the idea of keeping communication channel open. The purpose is avoiding interrupting data transfer so as to achieve the

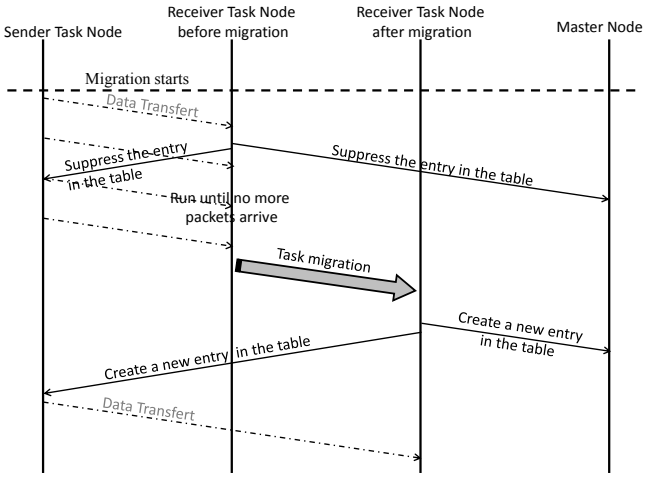


**Fig. 9.** First migration protocol datagram

highest possible level of performance. To fulfill such requirements, two features were developed: storage and forwarding of the incoming packets.

Figure 10 shows the second migration protocol. The initial situation is same as before: 2 tasks are communicating, task 1 sending to task 2 (step 0). When the NPU decides to trigger a migration, it does not have to notify the task predecessors anymore: the operating system will reroute the software fifos linked to this task into another fifo dedicated for migration (step 1). The packets are stored during the transfer of task 2 code to the destination node (step 2). When the task is rescheduled, it informs the node where it came from to initiate the redirection (step 3). At the same time, it updates the routing table of the master node and its predecessors. The Operating system of the new node containing task 2 has to now take care of two FIFOs: one coming from the redirection, which will be serviced first, and the other from the sender with its updated routing table, which has to be serviced after the whole redirected packets (step 4).

To avoid reordering, rerouted packets have to be serviced before those coming from the new communication channel. A *end of transmission* packet is sent in the end of the redirected stream to inform the new receiver NPU that it can process the packets coming directly from the senders.

Figure 11 shows the datagram of the migration procedure with redirection. Some hypothesis can be made comparing the datagram of the two protocols. First, the complexity of the second protocol is higher in term of computation, because of the storage and redirection functions, but also in term of message passing, with many more service messages. Concerning communication load, the fact of keeping the communication channel and sending a burst of redirected packets suggests a heavier bandwidth occupation. However, as closing and re-opening latency are avoided, higher computation rate of the whole structure and better computation performance are expected.

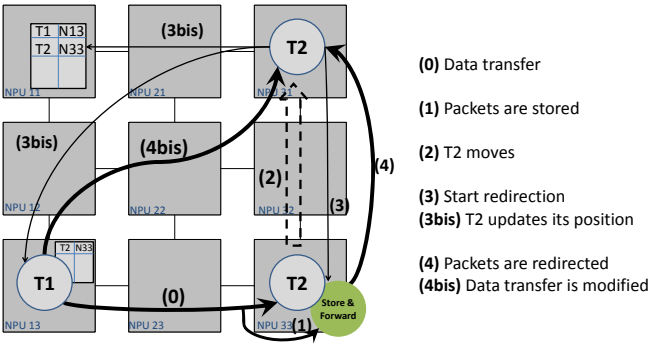


**Fig. 10.** Second migration protocol

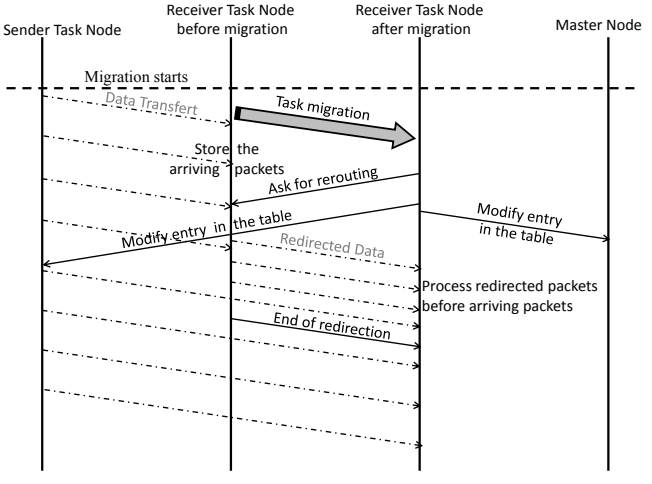


**Fig. 11.** Second migration protocol datagram

### 3.2 Experimental Results

**Benchmark Procedure.** The purpose of this section is to evaluate the overhead of dynamic task migration with the two protocols. In order to fairly assess the performance overhead induced by the proposed task migration strategies, experiments are conducted in best effort mode, therefore all implementations aim at maximizing performance rather than ensuring quality of service. This allows stressing the architecture as much as possible, hence migration cost become prominent rather than compensated by transient increase in cpu usage.

The experiments have been made on a case study yet realistic: the mjpeg decoding application. To simplify the experiment, only the three main tasks of mjpeg decoding have been implemented: IVLC, IDCT and IQUANT. A quick

profiling shows the average percentage of CPU time of each task in a sequential execution: IVLC take around 85% of the CPU time, IDCT 8% and IQUANT 5%. Figure 12 shows the task graph of the MJPEG application with the percentage of CPU time for each.

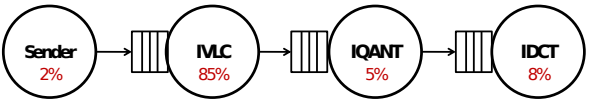


**Fig. 12.** MJPEG Task graph with percentage of computation time

Figure 13 shows the two different mappings that were used for our migration experiment: starting from configuration 1, the task 2 will migrate to NPU 2, which corresponds to the configuration 2. As the purpose here is to validate our algorithm and to evaluate the efficiency of each one, no considerations about mapping efficiency are treated. The only assumption done here is that configuration 2 is better suited for high performance, because the heaviest task (IVLC) is computed in a single NPU. This assumption will be confirmed by the performance benchmarks on static mapping in configuration 1 and 2, in the next paragraphs. More considerations about mapping issues are available in the articles referenced in Section 2.2 of this chapter.

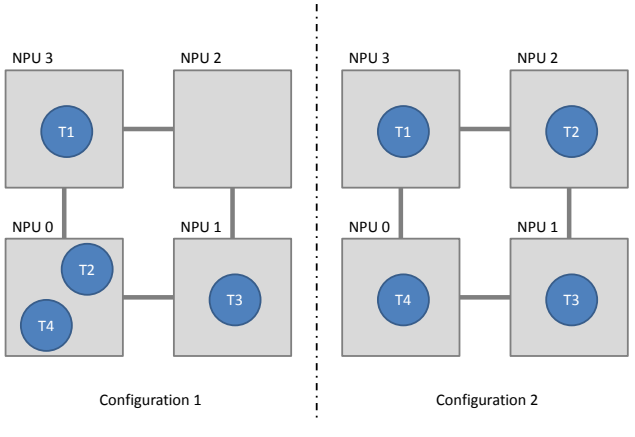


**Fig. 13.** Task configurations used in the benchmarks

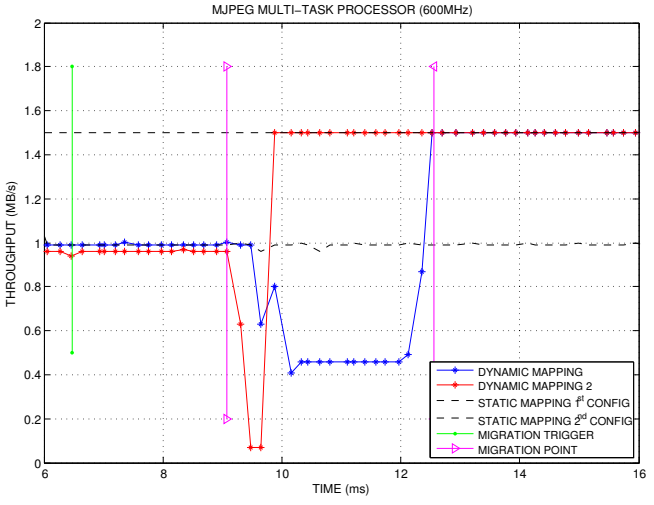


**Fig. 14.** Throughput of MJPEG for each case

**Evaluation of Communication Capabilities.** The evaluations in term of performance were made each time for 4 configurations: considering a static mapping with the configuration 1, considering a static mapping with configuration 2, triggering a migration of task 2 to go from configuration 1 to configuration 2 using the first protocol, and finally the same scenario with the second protocol. The migration order is triggered 6.5 ms after the beginning of the computation, to ensure the system is in a steady state (every task is running and computing packets).

Figure 14 shows the throughputs of the MJPEG for the 4 cases presented. The static mappings have, as expected, almost regular throughputs: the first configuration providing a 1 MB/s throughput while the second providing a 1.5 MB/s throughput. This result confirms the assumption made in the previous section about the higher compute capability of the second mapping.

Few observations can be done comparing the two migration protocols. First, even if the migration order is triggered at the same time, the migration process occurs at a different time. This can be explained by the fact that the two migration protocols use two different procedures before migrating the task: the first one closes the connection before the migration while the second relies on the forwarding techniques explained previously.

The second point stressed by Figure 8 is the migration time. The migration time here refers to the time between two steady states where the throughput is stable. In that case, the second protocol, with reordering, migrates much more rapidly than the first one: only 0.8 ms are necessary to reach the new throughput while 3.45 ms are necessary for the first one. We can see here the benefit of the second proposed technique, compared to the first one: the latency induced by closing and reopening the communication is here almost negated.

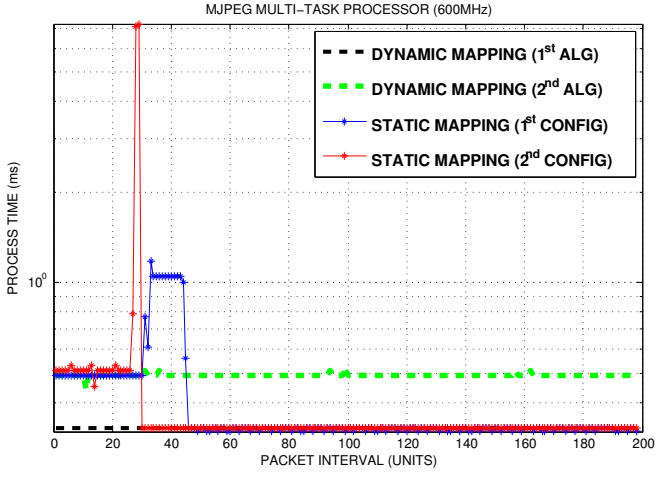


**Fig. 15.** MJPEG timing jitter for each case

However, in term of minimum bandwidth, the first protocol sees its bandwidth only dropping down to 0.45 MB/s whereas the second one goes down to 0.07 MB/s. This can be explained by the higher complexity of the second protocol: storage and forwarding processes prove computationally intensive and therefore reduce application performance during the transient phase.

This last consideration can be expected to be the main parameter when a strategy of migration has to be chosen. In streaming application like MJPEG for example, a minimum throughput is required, which can be prohibitive for the second protocol. However, in application which does not require real-time constraint, like compression or non streaming communication, the efficiency of the second protocol is higher.

Figure 15 shows the timing jitter for the 4 cases. The Y axis has been made logarithmic to clarify the results. We can observe that the timing jitter is, as expected, better in the second mapping than in the first one, and that the dynamic mapping scenarios vary as expected from the jitter value point of view. Moreover, this figure confirms the observations made previously: the timing jitter of the second dynamic mapping scenario shows a high and sharp peak during migration.

However, if we average the jitter during the migration time, the first algorithm gives a process time of 0.83 ms while the second gives 1.01 ms, i.e. less than 22% larger. The same average evaluation of the bandwidth during migration gives a throughput of 0.93 MB/s for the first algorithm and 1.31 MB/s for the second.

### 3.3 Task Migration - Closing Remarks

In this section we have proposed two task migration mechanisms for distributed memory MPSoCs: the first one closing the communications during the process,

the second one using buffering and redirection to avoid closing communications. Performance analysis has been made to compare both algorithms. Results have been validated thanks to a 2*2 homogeneous MPSoC architecture based on a distributed memory structure and a NoC.

Those benchmarks show that the overhead of the migration mechanisms is low and amortized by the performance gain of a better load distribution. The key parameter concerning the choice of the migration algorithm lies in the application specifications: in the case of real-time application, the first algorithm may prove to be more suitable because of the lower observed jitter in packet arrival. For best effort-application, the second algorithm will achieve a better global performance.

## 4 Memory Organization in MPSoC

### 4.1 Two Memory Management Philosophies, Implying Two Computation Models

In the literature, memory management in MPSoCs can be divided into two categories: shared memory models and distributed memory models.

Historically, shared memory models, with a vision of a single unified memory that can be accessed by any component are the most used. This popularity came from the original single core, single memory Von Neumann machine. To cope with multi-cores new architectures, caches strategies, including data coherency protocols have been implemented so that the compatibility with the old architectures still remains. Simultaneous Multi Threading (SMT), derived from this philosophy, is still the more commonly used model of computation in standard home computer CPUs. However, the heavy cost due to memory coherency controllers leads to a reconsideration of the memory model in embedded systems from shared memory to distributed memory.

The distributed memory model considers each core having its own independant memory. Communication and synchronization between the cores are made through messages. This model is derived from HPC, where a number of standards and API have been created, such as the Message Passing Interface (MPI) [51]. This model does not need any hardware memory coherency controllers (as each core is data independent), but leave the synchronizations to the software, making parallel computation harder. Nowadays, designing a complete message passing multi-tasking Operating System still remains a challenge [52].

Although the link between a memory model and a hardware architecture is obvious, it is interesting to note that a memory model does not rely on a particular hardware, and that some architectures possessing different memory modules distributed across the chip finally used a shared memory model. Those architectures are called Distributed Shared Memory models (DSM) as they have a physically distributed memory, but logically shared [53].

Such architectures imply a Non Uniform Memory Access (NUMA). To achieve correct performances, caches are provided that store remote memory data into a local faster memory. To maintain memory consistency inside those platforms, a

cache coherency protocol is provided. Hence, those architectures are also called Cache-Coherent NUMA (cc-NUMA) architectures.

### 4.2 A Hybrid Memory Model

**General Description.** The hybrid memory model described in this section has the aim of taking advantage of the two historical models. It consists in considering a distributed memory model, but with the addition of a Remote Memory Access (RMA). Hence, every core can access every memory inside the chip. However, no hardware cache coherency system has been implemented. The shared memory in this model is considered as critical small portion of memory, used only for general decision. No hardware cache coherency has been implemented to avoid the heavy cost of hardware data synchronization.

To preserve coherency of the shared memory, software cache coherency has to be implemented. The main advantage of software cache coherency is the fact that the programmer has a global view of the system, and so can optimize the protocol of synchronization depending on the nodes possessing the data [54]: knowing the owner of the data and the order of read/write operations on it allows more precise flush/invalidate actions in the caches. However, software cache coherency implies a higher performance penalty for the computation of the synchronization protocols. For that reason, the use of shared memory has to be made with caution.

**Initial Hardware Platform.** The initial platform used to develop this hybrid memory model was the same as described in section 3.1. To increase the compute capability of each NPU, the MIPS-I CPU was replaced by an Harvard based architecture using the Microblaze instruction set: the SecretBlaze [55].

**The Remote Memory Access.** To provide the access of memory located in a distant node, a RMA module has been implemented in the Network layer, which creates a Request/Acknowledge protocol to read data through the NoC. This RMA module is composed by two blocks: a *Send-RMA* module and a *Reply-RMA* module. The connections between these modules and the NoC were made with two more asynchronous fifos, as shown in figure 16.

The *Send-RMA* is responsible of servicing memory requests from the CPU. For a Write request, it first packetizes a flag corresponding to the write request, and then the addresses and data to write. For a Read request, it sends the flag corresponding to the read request, and then only the addresses to read. It then waits for the response and sends the data back to the CPU. The FSM describing the *Send-RMA* behavior is depicted figure 17-a.

The *Reply-RMA* receives the request coming from the other cores, and services them. For a Write request, data are written into the memory, and for a Read request data are read from the memory, then packetized and sent back to the source NPU. The FSM describing the *Reply-RMA* behavior is depicted figure 17-b.

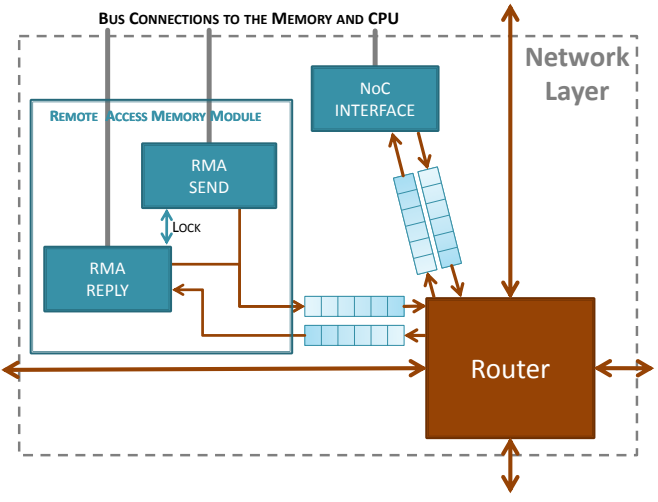


**Fig. 16.** Network on Chip with Remote Memory Access module

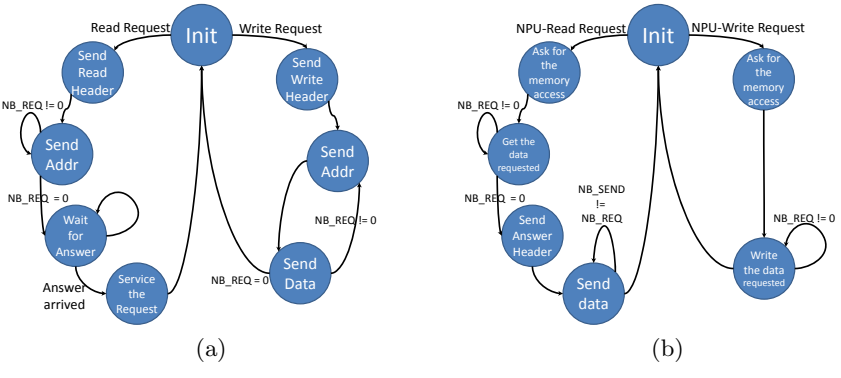


**Fig. 17.** RMA Finite Machine States: (a) *Send-RMA*, (b) *Reply-RMA*

Therefore, a master bus connection is needed for the *Reply-RMA* module to access memory, while a slave bus connection is sufficient for the *Send-RMA* module.

**Memory Mapping.** The distributed memory model which our system is derived has private memory for each node. Therefore, every node may have the same address mapping. However, the addition of the RMA module, allowing access to every memory modules in the architecture, requires an addressing scheme enabling remote NPU memory access. For legacy reason with our previous architecture, the 4 highest bits are kept for the selection of the bus connection. Bits 20 to 27 are used to select the coordinates of the memory node, in case of a remote access. The last 20 bits are used for memory address. This mapping

provides a small static virtualization of the memory - as the local memory is located at the same address space for every node. This allows simpler memory management, as local access can be detected using the same strategy for each node.

Thanks to this memory mapping, we can address up to 256 NPUs, with 1 MB of RAM each.

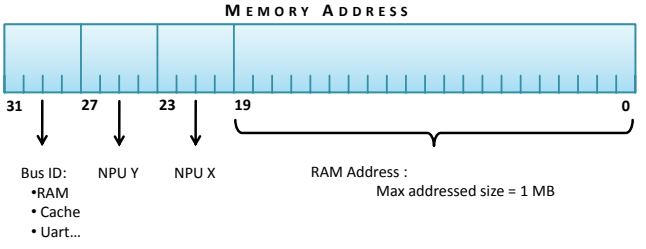


**Fig. 18.** Memory mapping

**Performance.** Since the transmission of the data through the NoC is time consuming, the local bus of each NPU has to be controlled carefully, to avoid unecessary stalling. Hence, the *Reply-RMA* possesses a buffer to store the addresses and data coming from the NoC before servicing them. As the requests are cache requests, the buffer has the size of a cache line. Thanks to that mechanism, the *Reply-RMA* module has an access time to the bus similar to the one of the local cache.

Figure 19 shows the number of clock cycles for each type of memory request. The number of 32-bits words of a cache line has been set to 8. So a memory operation handles 8 words of 32 bits, i.e. 256 bits. Measures have been made for different memory requests: from the local NPU, from a 1-hop distant NPU, and from a 2-hop distant NPU (i.e. the data have to go through 2 routers plus the local one).

An important factor in the memory access performance is the maximum NoC throughput. The Hermes router (introduced in Section 3.1)uses an asynchronous channel to send the data, with a tunable number of bit send for each transaction. As a transaction takes two clock cycles, the maximum throughput is directly linked to the number of bits send per transaction (the channel width). The throughput can be calculated using equation 2.

$$Throughput = \frac{Channel\ width}{2} * Frequency \tag{2}$$

The frequency of our design has been set to 50 MHz. The experiments have been made with a channel width of 4, 8, 16, and 32 bits, which leads to a maximum throughput of 100 MBps, 200 MBps, 400 MBPs, and 800 MBPs respectively. The maximum throughput of the wishbone bus is 1600 MBps.

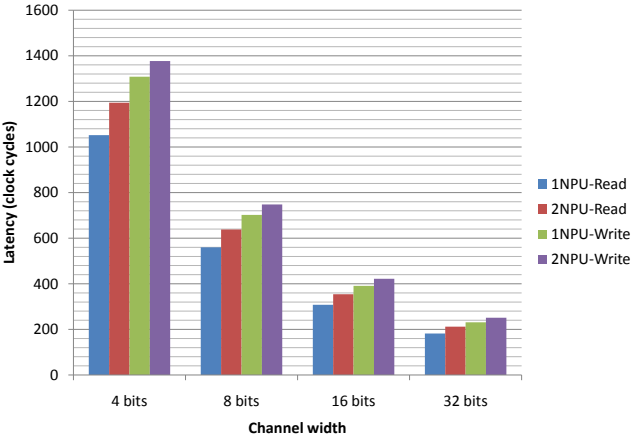


**Fig. 19.** Time to memory access in clock cycles

The figure shows that channel width is an important factor regarding the time to access the memory. Doubling the number of bits in the channel leads to a decreasing time to access of 53%, 55% and 59% respectively. Bus read and write transactions for a cache line takes 10 clock cycles. With a 200 clock cycles access time approxymately, the 32-bit wide channel router gives competitive results for a classical L1 cache remote fetch: if we assume a remote miss rate of 1% of the total miss rate, the performance of the cache will only decrease to 84% compared to local miss only. In comparison, standard DDR memory access latency is around 5000 clock cycles.

Another point stressed by this figure is the importance of the location of the data. Fetching a data from a 2 node distant NPU leads to a inscrease of 13% to 16% in access time compared to fetching from a 1 node distant NPU. We can notice that the wider the channel, the bigger the access time. This is due to the fact that the data are more buffered for small channels, which leads to a bigger latency during the access process. For these channels, this buffering, which is only slightly affected by the number of nodes crossed, takes a bigger part of the access time. Wide channels, however, are less affected by the buffering, and so comparatively more affected by the data location.

### 4.3 Memory Organization - Closing Remarks

In this section we have seen the different constraints and so the possible optimizations given by a hybrid distributed/shared memory structure. This new degree of freedom offers new load balancing strategies, since we can distribute the data independently to the execution.

## 5 Conclusion

This paper has presented an overview of adaptation issues in distributed memory MPSoCs. A new frequency scaling strategy for streaming applications has been developed, using feedback controllers. Next, two task migration protocols have been presented and compared. Finally, a hybrid memory approach has been described, allowing remote memory access inside a distributed archiecture.

The complementarity of those tools allows a global adaptation strategy whith increasing performance thanks to the combination of each. Indeed, frequency scaling can be used to ensure the reach of system requierements, while the two task migration protocols will balance the load through the platform. The throughput variations leaded by the migration processes will be lowered thanks to the frequency scaling.

Concerning the new memory organization, future work will include the development of a load balancing strategy based on the execution of data from another NPU without transfering the code, thanks to the remote memory access. The long term goal will be to conduct experiments with dynamic selection of the best suited load balancing techniques - from remote execution and the two task migration techniques - with respect to either application specifications or local parameters such as FIFO occupation or CPU workload.

## References

1. Wolf, W.: The Future of Multiprocessor Systems-on-Chips. In: Design Automation and Test in Europe (DATE 2004), pp. 681–685 (2004)
2. Marculescu, R., et al.: Outstanding Research Problems in NoC Design: System, Microarchitecture, and Circuit Perspectives. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 28 (2009)
3. Beraha, R., Walter, I., Cidon, I., Kolodny, A.: Leveraging Application-Level Requirements in the Design of a NoC for a 4G SoC - a Case Study. In: Design, Automation and Test in Europe (DATE 2010), pp. 1–6 (2010)
4. Chou, C.-L., Marculescu, R.: Run-Time Task Allocation Considering User Behavior in Embedded Multiprocessor Networks-on-Chip. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 29, 78–91 (2010)
5. Singh, A.K., et al.: Communication-aware heuristics for run-time task mapping on NoC-based MPSoC platforms. Journal of Systems Architecture 56 (2010)
6. Almeida, G.M., Sassatelli, G., Benoit, P., Saint-Jean, N., Varyani, S., Torres, L., Robert, M.: An adaptive message passing mpsoc framework. International Journal of Reconfigurable Computing, 1–20 (2009)
7. Faruque, M.A., et al.: ADAM: Run-time Agent-based Distributed Application Mapping for on-chip Communication. In: DAC 2008 (2008)
8. Bertozzi, S., Acquaviva, A., Bertozzi, D., Poggiali, A.: Supporting task migration in multi-processor systems-on-chip: A feasibility study. In: DATE 2006: Design, Automation and Test in Europe, pp. 1–6 (2006)
9. Barcelos, D., Brião, E.W., Wagner, F.R.: A hybrid memory organization to enhance task migration and dynamic task allocation in NoC-based MPSoCs. In: Proceedings of the 20th Annual Conference on Integrated Circuits and Systems Design, SBCCI 2007, pp. 282–287 (2007)

10. Cuesta, D., Rodrigo, A., Luis, J., Perez, H., Ignacio, J., Atienza Alonso, D., Acquaviva, A., Macii, E.: Adaptive Task Migration Policies for Thermal control in MPSoCs. In: IEEE 2010 Annual Symposium on VLSI, pp. 110–115 (2010)
11. Milojicic, D.S., Douglis, F., Paindaveine, Y., Wheeler, R., Zhou, S.: Process migration. ACM Computing Surveys 32 (2000)
12. Marchesan Almeida, G., Busseuil, R., et al.: Evaluating the impact of task migration in multi-processor systems-on-chip. In: SBCCI 2010: Proceedings of the 23rd Symposium on Integrated Circuits and System Design, pp. 73–78 (2010)
13. Simunic, T., Benini, L., Acquaviva, A., Glynn, P., Micheli, G.D.: Dynamic voltage scaling and power management for portable systems. In: Proceedings of the 38th Annual Design Automation Conference, pp. 524–529. ACM, Las Vegas (2001)
14. Herbert, S., Marculescu, D.: Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In: Proceedings of the 2007 International Symposium on Low Power Electronics and Design, pp. 38–43. ACM, Portland (2007)
15. Talpes, E., Marculescu, D.: Toward a multiple clock/voltage island design style for power-aware processors. IEEE Trans. Very Large Scale Integr. Syst. 13, 591–603 (2005)
16. Wu, Q., Juang, P., Martonosi, M., Clark, D.W.: Formal online methods for voltage/frequency control in multiple clock domain microprocessors. SIGARCH Comput. Archit. News 32, 248–259 (2004)
17. Shang, L., Peh, L., Jha, N.K.: Dynamic Voltage Scaling with Links for Power Optimization of Interconnection Networks. In: Proceedings of the 9th International Symposium on High-Performance Computer Architecture, p. 91. IEEE Computer Society (2003)
18. Ogras, U.Y., Marculescu, R., Marculescu, D.: Variation-adaptive feedback control for networks-on-chip with multiple clock domains. In: Proceedings of the 45th Annual Design Automation Conference, pp. 614–619. ACM, Anaheim (2008)
19. Yin, A.W., Guang, L., Nigussie, E., Liljeberg, P., Isoaho, J., Tenhunen, H.: Architectural Exploration of Per-Core DVFS for Energy-Constrained On-Chip Networks. In: DSD, pp. 141–146 (2009)
20. Alimonda, A., Carta, S., Acquaviva, A., Pisano, A.: Non-Linear Feedback Control for Energy Efficient On-Chip Streaming Computation. In: IES, pp. 1–8 (2006)
21. Alimonda, A., Carta, S., Acquaviva, A., Pisano, A., Benini, L.: A Feedback-Based Approach to DVFS in Data-Flow Applications. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 28, 1691–1704 (2009)
22. Puschini, D., Clermidy, F., Benoit, P., Sassatelli, G., Torres, L.: Temperature-Aware Distributed Run-Time Optimization on MP-SoC Using Game Theory. In: ISVLSI, pp. 375–380 (2008)
23. Puschini, D., Clermidy, F., Benoit, P., Sassatelli, G., Torres, L.: Adaptive energy-aware latency-constrained DVFS policy for MPSoC. In: SOCC, pp. 89–92 (2009)
24. Goossens, K., She, D., Milutinovic, A., Molnos, A.: Composable Dynamic Voltage and Frequency Scaling and Power Management for Dataflow Applications. In: DSD, pp. 107–114 (2010)
25. Zhu, Y., Mueller, F.: Feedback edf scheduling exploiting hardware- assisted asynchronous dynamic voltage scaling. SIGPLAN Not. 40, 203–212 (2005)
26. Sharifi, A., Zhao, H., et al.: Feedback control for providing qos in noc based multicores. In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2010), pp. 1384–1389 (2010)
27. Bennett, S.: A History of Control Engineering, 1800-1930. Institution of Electrical Engineers, Stevenage, UK (1979)

28. Marchesan Almeida, G., Busseuil, R., Alceu Carara, E., Hebert, N., Varyani, S., Sassatelli, G., Benoit, P., Torres, L., Gehm Moraes, F.: Predictive Dynamic Frequency Scaling for Multi-Processor Systems-on-Chip. In: IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1500–1503 (2011)
29. Wildermann, S., et al.: Run time Mapping of Adaptive Applications onto Homogeneous NoC-based Reconfigurable Architectures. In: FPT 2009 (2009)
30. Kangas, T., et al.: UML-based multiprocessor SoC design framework. ACM Transactions on Embedded Computing Systems 5(2) (2006)
31. Hölzenspies, P.K.F., et al.: Run-time Spatial Mapping of Streaming Applications to a Heterogeneous Multi-Processor System-on-Chip (MPSoC). In: DATE 2008 (2008)
32. Chou, C.-L., et al.: User-Aware Dynamic Task Allocation in Networks-on-Chip. In: DATE 2008 (2008)
33. Carvalho, E., et al.: Dynamic Task Mapping for MPSoCs. Design and Test of Computers 27(5) (2010)
34. Smit, L.T., et al.: Run-time mapping of applications to a heterogeneous SoC. In: SoC 2005 (2005)
35. Streichert, T., Strengert, C., Haubelt, C., Teich, J.: Dynamic task binding for hardware/software reconfigurable networks. In: SBCCI 2006: Proceedings of the 19th Annual Symposium on Integrated Circuits and Systems Design, pp. 38–43 (2006)
36. ARM Limited. Mpcore linux 2.6 smp kernel and tools
37. Barak, A., Laadan, O., Shiloh, A.: Scalable cluster computing with mosix for linux. In: Proceedings of Linux Expo 1999, pp. 95–100 (1999)
38. Carta, S., Acquaviva, A., Del Valle, P.G., Atienza, D., De Micheli, G., Rincon, F., Benini, L., Mendias, J.M.: Multi-processor operating system emulation framework with thermal feedback for systems-on-chip. In: Proceedings of the 17th ACM Great Lakes Symposium on VLSI, pp. 311–316 (2007)
39. Greiner, A.: Tsar: a scalable, shared memory, many-cores architecture with global cache coherence. In: 9th International Forum on Embedded MPSoC and Multicore (2009)
40. Bertozzi, D., Benini, L.: Mparm: Exploring the multi-processor soc design space with systemc. The Journal of VLSI Signal Processing 41(2), 169–182 (2005)
41. Held, J.: From a Few Cores to Many. Citeseer, 12. Intel White Paper, `http://download.intel.com/research/platform/terascale`
42. Dumitrascu, F., Bacivarov, I., Pieralisi, L., Bonaciu, M., Jerraya, A.: Flexible MPSoC platform with fast interconnect exploration for optimal system performance for a specific application. In: Proceedings of the Conference on Design, Automation and Test in Europe: Designers' Forum (DATE 2006), European Design and Automation Association, 3001 Leuven, Belgium, Belgium, pp. 166–171 (2006)
43. Acquaviva, A., Alimonda, A., Carta, S., Pittau, M.: Assessing task migration impact on embedded soft real-time streaming multimedia applications. EURASIP J. Embedded Syst., 1–15 (2008)
44. Pittau, M., Alimonda, A., Carta, S., Acquaviva, A.: Impact of task migration on streaming multimedia for embedded multiprocessors: A quantitative evaluation. In: Chakraborty, S., Eles, P. (eds.) ESTImedia, pp. 59–64 (2007)
45. Briao, E.W., Barcelos, D., Wagner, F.R.: Dynamic task allocation strategies in mpsoc for soft real-time applications. In: DATE 2008: Proceedings of the Conference on Design, Automation and Test in Europe, pp. 1386–1389 (2008)

46. Marchesan Almeida, G., Varyani, S., Sassatelli, G., Busseuil, R., Benoit, P., Torres, L., Robert, M.: Self-adaptability in Multi-processor Embedded Systems. In: GDR SOC-SIP (2009)
47. Lin, X., McKinley, P.K., Ni, L.M.: Deadlock-free multicast wormhole routing in 2-d mesh multicomputers. IEEE Trans. Parallel Distrib. Syst. 5(8), 793–804 (1994)
48. Moraes, F., Calazans, N., Mello, A., Moller, L., Ost, L.: Hermes: an infrastructure for low area overhead packet-switching networks on chip. Integration, the VLSI Journal 38(1), 69–93 (2004)
49. MIPS Corp. Mips technologies, `http://www.mips.com`
50. Rhoads, S.: Plasma - most mips i(tm), `http://www.opencores.org/project,plasma`
51. The Message Passing Interface Standard, `http://www.mcs.anl.gov/research/projects/mpi/index.htm`
52. Dasgupta, P., LeBlanc Jr., R.J., Ahamad, M., Ramachandran, U.: The Clouds distributed operating system. IEEE Computer 24, 34–44 (1991)
53. Protic, J., Tomasevic, M., Milutinovic, V.: Distributed shared memory: concepts and systems. IEEE Parallel and Distributed Technology: Systems and Applications 4, 63–71 (1996)
54. Ophelders, F., Corporaal, H., Bekooij, M.: A Tuneable Software Cache Coherence Protocol for Heterogeneous MPSoCs. Master's thesis, Eindhoven University of Technology (2009)
55. Barthe, L., Cargnini, L.V., Benoit, P., Torres, L.: The SecretBlaze: A Configurable and Cost-Effective Open-Source Soft-Core Processor. IEEE IPDPS/RAW (2011)