Run-time remapping algorithm of dataflow actors on NoC-based heterogeneous MPSoCs

Mostafa Rizk, Kevin J. M. Martin, Jean-Philippe Diguet

Run-time remapping algorithm of dataflow actors on NoC-based heterogeneous MPSoCs

Mostafa Rizk, Kevin J. M. Martin, and Jean-Philippe Diguet

***Abstract*—Multiprocessor system-on-chip (MPSoC) platforms have been emerging as the main solution to cope with processor frequency ceiling and power density issues while still improving performances. Then, network-on-chip (NoC) has been adopted to provide the increasing number of processors with the required communication bandwidth as well as with the necessary flexibility. Video processing and streaming applications are adopting dynamic dataflow model of computation as the need for high performance parallel computing is growing. Dataflow applications executed on modern MPSoC-based architectures are becoming increasingly dynamic and more data-dependent. Different tasks execute concurrently with significant modifications in their workloads and resource demanding over time depending on the input data. Hence, adopting any static or offline dynamic scheduling for mapping tasks will not cope with the computation variations. This paper introduces an original run-time mapping algorithm based on the Move Based (MB) method targeting a dedicated heterogeneous NoC-based MPSoC architecture to achieve workload balancing and optimized communication traffic. The performance of the proposed algorithm is verified by conducting cycle-accurate SystemC simulations of the adopted NoC implementing a real MPEG4-SP decoder. The obtained results reveal the effectiveness of our proposed algorithm. For various real-life videos, the proposed algorithm systematically succeeded to enhance significantly the performance.**

***Index Terms*—NoC, Heterogeneous MPSoC, Run-time remapping, Dataflow actor, Move-based algorithm.**

I. Introduction

MULTIPROCESSOR system-on-chip (MPSoC) platforms have been emerging as the main solution to cope with processor frequency ceiling and power density issues while still improving performances. Then, networks-on-chip (NoCs) have been adopted to provide the increasing number of processors with the required communication bandwidth as well as with the necessary flexibility. But legacy code for instance, mainly designed for single or few core architectures, does not scale well with manycore architectures and fails to fully benefit from the available parallelism. However, as discussed decades ago [1], dataflow programming can address the limitations of conventional approaches regarding synchronization and shared memory issues. With the rise of massively parallel architectures, we can reconsider the use of dataflow programming as a solution to efficiently exploit the resources of parallel architectures for computing intensive application domains such as video coding, computer vision, machine learning and physics simulation for instance.

A dataflow application can be specified as a graph where nodes, called actors, process data called token(s). The computational models are based on First-In First-Out (FIFO) buffers and respect their formalized read and write rules. Each FIFO holds a set of tokens. Fig. 1(a) illustrates a network of actors, which exchange tokens through defined FIFO channels [2]. Fig. 1(b) presents an example of a structure of the software FIFO generated with the tool ORCC [3]. A network of actors holds specific features that make it different from a generic task graph. First, an actor is non-preemptive. Once started, an actor ends its execution. Second, the actor can start if and only if there are enough tokens as input, and enough space in the output FIFOs. The FIFOs are considered updated (i.e. tokens consumed and produced) at the end of the execution of the actor, establishing a conservative synchronization scheme, and preventing from any data race.

When the number of actors is larger than the number of processing elements (PEs), then the main design challenge is the mapping of actors on the network of PEs. In the case of static dataflow [4], where the number of tokens produced and consumed by the actors is known, an optimal solution can be computed offline [5]. However, an increasing number of applications cannot be specified with a static graph since the performance improvement of complex applications usually lead to context and data-dependent optimizations. This evolution is, for instance, significant in the domain of video coding. Dynamic models are then used to express data-dependent behavior of some applications [6]. Dynamic dataflow is a useful model of computation (MoC) for handling streaming data and video processing applications.

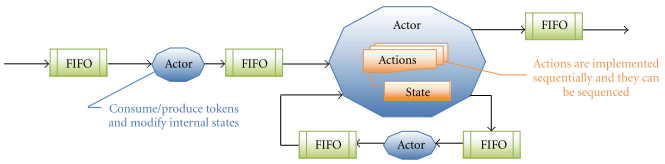
As the workload of an actor may change according to the input data set, adapting the mapping while the application runs is required to optimize the use of the computing and communication resources. The mapping problem is known as NP-complete. Heuristic methods, for a fast response time, are thus required to address manycore architectures. Run-time adaptation relies on system observation, decisions and configurations. Several previous works have addressed the problem of task mapping at run-time. In [7], the authors have proposed a dynamic resource balance algorithm targeting NoC-based Many-core homogenous platforms to enhance the system performance by balancing the utilization of on-chip computing resources and communication resources. In [8], the authors have introduced a hybrid application mapping

M. Rizk is with CNRS and member of Lab-STICC UMR CNRS 6285, Brest, France also with the School of Engineering, International University of Beirut, Lebanon, and with the Physics and Electronics Department, Lebanese University, Lebanon. e-mail: mostafa.rizk@imt-atlantique.fr

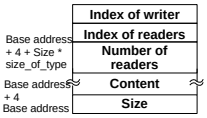
K. J. M. Martin is with Universtité de Bretagne-Sud (UBS) and member of Lab-STICC UMR CNRS 6285, Lorient, France

J-P. Diguet is with CNRS and member of CROSSING, IRL CNRS 2010, Adelaide, Australia

Manuscript received Month DD, YYYY; revised Month DD, YYYY.



(a) Network of Actors



(b) Structure of the software FIFO

Fig. 1. Actors and software FIFO Models [2]

that combines design-time analysis with run-time mapping in the context of dynamic thermal and reliability-aware resource management. Most of the available methods focus on determining the suitable mapping of tasks before starting the execution of the application [9], [10], [11], [12], [13], [14]. The mapping of actors is also an active topic for other target platforms like Coarse-Grained Reconfigurable Arrays (CGRA) [15] or Field Programmable Gate Array (FPGA) [16].

This research work addresses the problem of reconfiguring at run-time and at the application level the mapping of dataflow actors on heterogeneous processors. In this work, heterogeneous means that processors share the same instruction set architecture (ISA) while having different coprocessors and different clock domains. The proposed method, which is called *run-time remapping*, relies on continuous monitoring of exact performance metrics such as the computational time and communication time during real-time execution of the application. Accordingly, a new mapping of the involved actors is determined at run-time targeting the enhancement of the overall performance. This approach is sequentially repeated while the application is running. The application is neither suspended nor modified. The proposed remapping method meets with the dynamic behavior of dataflow applications. Static or offline mapping methods cannot capture the dynamic behavior and thus may not lead to optimal solutions. Also, on-the-fly and hybrid mapping methods suffer from a lack of means to monitor the performance and remap the actors accordingly. In order to apply the proposed method, the architecture of NoCs must be augmented to efficiently provide new services of monitoring performance metrics and remapping the actors, which are not available in conventional networks.

Adopting the devised remapping method and NoC-based architecture leads to balancing the workload. The obtained results show that the adoption of the remapping method reduces the standard deviations of the computational times and communication times of involved processors by 38.58% and 69% respectively. Thus, the variation of the use rate of processors is reduced compared to running the application without remapping. In addition, a reduction of 8.6% in the total execution time has been achieved as well as a reduction of 21% in the number of packets' hops is recorded when comparing to the execution without remapping.

In this paper we introduce three contributions:

- First, we optimize for a NoC-based architecture with heterogeneous processors, a new run-time remapping (RR) algorithm based on the Move Based (MB) method [17], which allows only one actor to move at a time from one processor to another. Our solution is then compared with state of the art methods for dataflow architectures.
- Second, we present new NoC services that allow to implement the observation and adaptation mechanisms.
- Finally, we demonstrate our solution with a full implementation of MPEG4-SP, which is available as a reference of a typical dynamic dataflow application. It is also complex enough to exhibit data-dependent execution and communication times. We consider a SystemC packet cycle-accurate NoC simulator to fully decode reference videos and demonstrate the effectiveness of the adaptation mechanism with a real-life dataflow application.

The rest of the paper is organized as follows. Section II presents the related work. Section III illustrates the adopted architecture model. Section IV describes the processing flow. Section V details the conducted experiments and presents the obtained results. Finally, Section VI concludes the paper.

II. Related Work

The question of mapping parallel applications on multi or many-core architectures is a very wide problem, with a large number of dimensions, including the programming model, the target architecture (homogeneous or heterogeneous, bus-based or NoC-based, etc.), and the optimization goal (throughput, execution time, energy, etc.) [18]. The interested reader can refer to the paper gathering different mapping strategies for NoC-based architectures [19]. Following the taxonomy proposed in [18], the mapping problem can be solved based on two main strategies: design-time, and run-time. When solved at design-time, the mapping is called *static* since it's computed offline and does not change while the application runs. This approach allows for exact methods to find an optimal solution [20] [5] [15], but suffers from a lack of flexibility since it cannot capture the dynamic behavior of some applications. Moreover, even in the case of deterministic execution times of actors in a static context, the paper [21] interestingly shows the difference between the optimal mapping obtained from a well-formalized problem and the real execution trace, due to execution variabilities coming from the hardware.

The dynamic workload should be handled using run-time techniques. The run-time mapping strategies can themselves be divided into two categories: on-the-fly mapping, or hybrid mapping. On-the-fly mapping techniques are application- and platform-agnostic and solve the problem online. Very simple and efficient heuristics should be used to shorten the response time. For NoC-based MPSoCs, various fast heuristics targeting the reduction of communications under constraints have been already proposed [22] [23] [24]. These approaches consider one task per core. Allowing multiple tasks on one core is considered in [10]. Heuristics are fast but can be far from optimal solutions, so hybrid approaches have been introduced. They are based on pre-computed optimal solutions for a set of cases. The job is split into two phases: (1) at design-time, a set of solutions is computed, and (2) one solution

is selected at run-time. A wide variety of approaches can then be cited: based on traces in [25], on priority in [26], on scenario in [27], on previously identified design points in [28], or on WCET and scheduling in [29]. None of these studies demonstrates its efficiency with real video applications running reference sequences. The proposed real-time mapping reconfiguration method in [8] requires to suspend the currently running application and the manager remaps the tasks at run-time according to scenarios previously defined at design-time based on the evaluation of multiple mappings, optimizing for their resource requirements and power consumption. Finally, a last approach can fall into the family of hybrid mappings, which considers to recompute partially the mapping problem at run-time. This is called *run-time remapping*. The work presented in this paper follows such an approach for dataflow applications and leverages design-time analysis profiling results to find at run-time a first mapping. The application is then monitored to update the profiling results and a run-time remapping algorithm runs regularly to check if a new mapping would be better than the current one.

Among the innumerable papers dealing with task mapping, we consider run-time methods for dataflow tasks, and have identified a limited number of solutions. In [30], the mapping is modeled as a graph partitioning problem, and the problem is solved at run-time by METIS tool, based on profiling information obtained by a first run. Though the migration cost of the actors is not taken into account, the results are promising and could be improved if the mapping does not change completely at each iteration. The approach in [17] allows to successively refine the mapping according to the dynamic behavior of the application, by allowing only one actor to move at a time from one processor to the other. This approach assumes dynamic dataflow application and the target architecture is composed of several heterogeneous cores interconnected by a bus or a NoC. The communication cost is computed based on a rough analytical model of the interconnection network, with the loss of accuracy that comes with it, whereas in our work, we consider profiled values gathered automatically by the system, with a finer grain down to the link. In [31], the application is specified with KPN (Kahn Process Network) and the target architecture is a shared-memory based MPSoC, with also a model of the communication channel (bus or NoC). The approach proposes to rely on three main steps: the two usual design-time preparation and run-time mapping steps plus a new customization step. The design time step computes a set of candidates and populates a database. The run-time mapping initialization derives from the candidates a new initial mapping for the given workload. Finally, the run-time customization step incorporates a Scenario-based run-time Task Mapping (STM) algorithm that is applied to find new mapping of tasks when the system detects that an objective is unsatisfied. It first detects the so-called *critical task* and then identifies why it misses its objectives: either poor locality or load imbalance. In case of poor locality, an algorithm that considers the communication between tasks is used to find a new mapping. In case of load imbalance, a load balancing strategy based on computational demands of the tasks is used. This step produces a new mapping that may move several tasks, which leads to a (re-)mapping overhead.

When focusing on the small subset of the existing work around hybrid and run-time (re-)mapping of dataflow applications on NoC-based architectures, we consider the work presented in [31] for comparison.

III. Architecture Model

The target architecture is a heterogeneous Multi-Processor System on Chip (HMPSoC) containing several different PEs and shared memories connected with a Network-on-Chip (NoC). Fig. 2 presents the structure of the adopted NoC-based architecture. Our method is scalable and without loss of generality we consider a specific model of architecture which is required for a data-accurate functional simulation with a packet-level time accuracy. The target architecture is a 4×4 mesh-based NoC with 32-bit links that interconnects 28 intellectual property (IP) cores including 15 memory modules, 12 PEs and a processing element that acts as a manager (MGR). The PEs and memory modules are technologically independent of the structure of the NoC. They communicate through the network using a network interface (NI). We consider a simple NoC model that employs the wormhole packet switching mode, the deterministic XY routing algorithm, and a flow control policy without virtual channels. The implemented routers have one buffer of 3 flits per input port and use distributed arbitration logic (one arbiter per port). The back-end part of the NI is typical and includes a packet maker/un-maker, which are used to assemble and disassemble the packets, and a priority manager to synchronize packet transmission and reception.

In this work, it is assumed that $PE1$ imports the incoming streamed data from an Input buffer and $PE12$ outputs the processed data. Fig. 2 illustrates the buffers in order to communicate with external systems. Each PE has its local memory. It is assumed that there are no restrictions to map any MPEG4-SP application actor to any PE. The used PEs can all work in parallel according to dataflow firing rules. However, some PEs are enhanced by hardware accelerators dedicated to certain functionalities in order to perform them more efficiently. The shared memories are distributed in memory blocks which have a unique NI. From an NoC perspective, the novelty is the introduction of new command packets used as instructions to manage FIFO accesses, broadcast mapping information, collect monitoring data, and the transfer of binary codes. In order to cope with the command packet and associated notification packet concepts, the NIs implement some additional logic modules. The command packets were already proposed in [32] but only produced by the manager and for a specific application.

A. Manager

The manager is a PE dedicated to the following five tasks: (1) map initially the actors on the available PEs, (2) parse the feedback collected data from all modules (memories and PEs), (3) apply the run-time remapping algorithm and selects the actor to be moved (if any), (4) notify the corresponding PEs (*looser*, *gainer*, etc.) about the updated mapping and

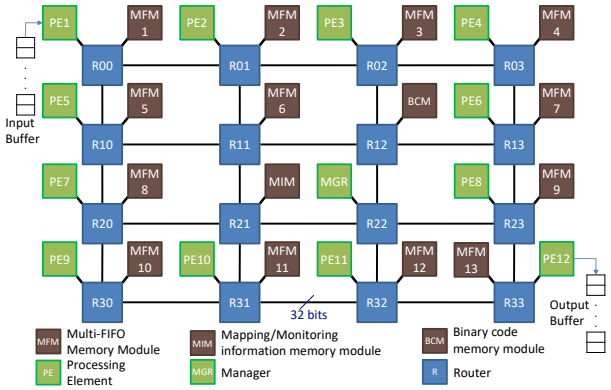


Fig. 2. The structure of the used NoC-based architecture

TABLE I
HARDWARE ACCELERATORS USED IN THE SIMULATION PLATFORM

PE ID	Accelerated Function	Acceleration Ratio
PE3 & PE6	IDCT	1/0.3
PE4	IQ + IAP	1/0.75
PE10	Add	1/0.57
PE11	Interpolation	1/0.4

(5) manage the transferring of the binary code corresponding to the moved actor from the shared memory into the cache of the gainer processor.

B. Processing Elements

The target platform includes twelve PEs. All PEs are supposed to be able to execute any of the forty-one actors involved in the MPEG4-SP application. As the number of PEs is smaller than the number of actors, each PE is considered to run more than one actor. Hence, an actor scheduler is required to manage the order of execution of actors. Mainly, in dataflow applications, all schedulers suffer from inefficient polling which leads to useless memory accesses when a scheduling attempt fails. In this work, the well-known round-robin scheduling technique has been adopted in all PEs. The actors are given the attempt to be executed in a circular order without priority. The PE will execute the allocated actor if there are enough input tokens and enough space in the output FIFOs as specified in dataflow applications.

Furthermore, some PEs are augmented with hardware accelerators in order to perform special functions more efficiently. In this work, we adopt one of the hardware accelerator specification described in previous similar work [17]. Table I shows the list of accelerators adopted in the simulation platform. In addition, the PEs have been specified randomly to operate on different frequencies. Table II shows the randomly chosen operating frequency of all PEs in terms of the NoC operating frequency f.

C. Memory Modules

The tailored platform integrates three types of memory modules. Each module includes a memory block that returns the data allocated at its specified address. Since the PEs and the manager do not recognize the local mapping of stored data

TABLE II
PROCESSING ELEMENT OPERATING FREQUENCY

PE ID	Operating Frequency
PE1, PE12, MGR	f
PE2, PE6, PE10	$2f$
PE3, PE7, PE11	$3f$
PE4, PE8	$4f$
PE5, PE9	$5f$

in each memory module and in order to remain compliant with any available memory, the typical NI is extended to accommodate the services for managing the addressing and arranging the retrieved output bits into flits. These new functionalities are implemented as additional components in the front-end of the NI corresponding to each memory module type in order to be independent of NoC parameters. In the following the functionalities of each memory type is described.

1) Binary code memory module (BCM): It contains the binary codes of all actors. The manager sends a specific packet request to BCM to forward the binary code of the moved actor to a given PE according to the decision taken after executing the RR algorithm. A simple module, so-called memory address mapper (MAM), is integrated into the NI of the BCM in order to find the correct memory address. For a specific actor, MAM determines the starting address of the binary code and its corresponding size based on the actor's ID and by the means of simple look-up-tables that include the starting addresses and the size of the binary codes of all actors. Furthermore, MAM manages the extraction of data from the memory and delivers it to the packet maker unit.

2) Mapping/Monitoring information memory module (MIM): This memory module accommodates twelve memory blocks. Each block is dedicated to a specific PE and is supposed to store two types of data. The first type is the mapping information, which is generated by the manager and indicates which actors are to be executed by each PE in addition to their supplementary information about input and output FIFOs and the reading orders for each input FIFO (III-D1c). The second type is the monitoring information (III-D1e), which is collected by the PEs during processing a specified number of video frames. Storing the monitoring information overwrites the mapping information, which is not needed by the PEs anymore.

When a packet holding either mapping or monitoring information is received, the MIM module first identifies the corresponding PE. Accordingly, it dissembles the packet and stores the data found in the packet payload into the memory block assigned to the identified PE.

Moreover, the MIM informs the manager about the availability of new monitoring information and the corresponding PE about the availability of new mapping information. To do so, the MIM sends notification packets (III-D1a) as per the concept of notifying memory concept demonstrated in [2]. In addition, the MIM responds to reading requests (III-D1b) sent from the manager to acquire the stored monitoring information from the PEs or to get the new mapping information.

3) Multi-FIFO memory module (MFM): This type of memory module is dedicated to store the data which is either

TABLE III
ADDRESSES DETERMINED BY THE MFM-NI CONTROLLER

Packet Type	Starting Address	Offset
Request/Set writing index	$FIFOsize$	0
Request/Set reading index	$FIFOsize + 1$	reading order
Reading Request packet	Reading address	incremented till
Data packet	Writing address	reaching data size

imported to the system or processed by the PEs. Each MFM accommodates a specific number of FIFOs. Only one FIFO is used at once. The inputs of all FIFOs are connected to the module's inputs using demultiplexers whereas the FIFOs' output data ports are multiplexed. This signal is buffered from the value of FIFO address which is specified in the payload of the arriving packets (see Fig. 4). The multiplexer and demultiplexers are added to the adapter in the NI.

Moreover, the MFMs receive the following types of packets: (1) FIFO Index packet (III-D1f) that aims either to retrieve or to set the writing and reading indexes, (2) Data Reading Request packet (III-D1g) that demands to read data from a specified FIFO and (3) Data packet (III-D2a) that is used to write data in a specified FIFO.

A simple circuit is integrated into the adapter of the NI in all MFM modules in order to manage the memory addressing for all listed-above packet types. It is composed of a simple controller and two 4-to-1 multiplexers and an adder in order to generate the appropriate address values to be given to the MFM FIFOs. After disassembling the arriving packet, the packet un-maker delivers the packet type and the data size to the controller. Accordingly, the controller generates the control signals to configure the two multiplexers, which are dedicated to select the values of starting address and the offset as listed in Table III. These two values are then added to compute the memory address. In addition, the controller determines the number and type of the required memory accesses. It incorporates a simple comparator and an address counter which is incremented for each required access.

D. Packets' structure

The developed NoC architecture considers two categories of packets: (1) command packets and (2) data packets.

1) Command packets: Command packets are initiated by the cores and processed by the NIs of destination nodes. Several command packets, described hereafter, have been opted in order to manage FIFO accesses, send mapping information, collect monitoring data, and manage the transferring of binary codes.

a) Notification packets (NP): The NPs aim to inform the PEs that new information is ready to be requested. This technique is inherited from the notifying memories (NM) concept presented in [2]. When receiving a NP, the PE will send a reading request to retrieve the available data at the corresponding notifying memory. In this work, notification packets are used either to inform an ordinary PE that new mapping information is available or to notify the manager that updated monitoring information has been generated and stored. The NP has empty payload and aims to trigger the manager and PEs to request data when it is ready rather than frequent inefficient polling.

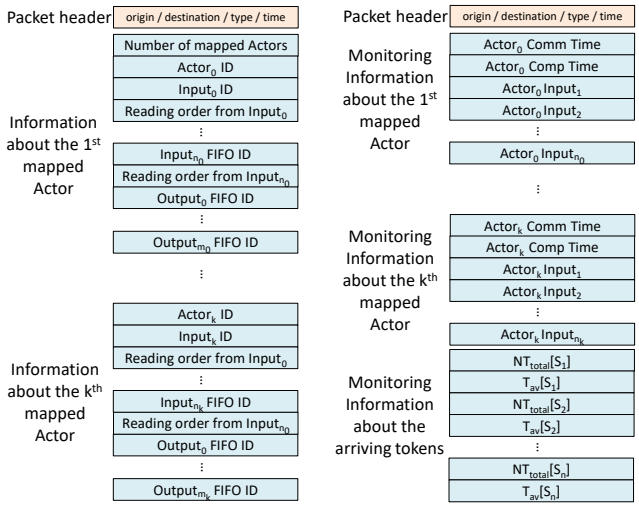


Fig. 3. Packets structure for mapping (left) and monitoring (right) information

b) Monitoring/Mapping information reading request packets (MRP): This type of packet is used to request the information stored in the MIM module as a response to the NP. It is either generated by the manager to acquire the new monitoring information sent from a definite PE or by one of the PEs to get the new mapping information provided by manager. For both information types, monitoring or mapping information, the request packet does not include any payload.

c) Mapping information packets (M_pIP): The manager uses a M_pIP to inform all involved PEs after determining or modifying the actor mapping strategy. Its payload includes the following: (1) the number of actors which are mapped to the PE, (2) the IDs of the mapped actors, (3) the IDs of the input and output FIFOs, and (4) the actor reading order in each input FIFO. Fig. 3 illustrates the structure of the packet holding the mapping information.

d) Mapping Confirmation packets (MCP): A MCP aims to inform the manager that the new mapping information is well received by both the former and the new owner of the actor. The MCP payload is also empty.

e) Monitoring information packets (M_nIP): This type of packet holds the feedback information needed by the manager to perform the RR algorithm. Fig. 3 presents the structure of the monitoring information packet.

f) FIFO index packets (FIP): The FIPs are designed to hold the writing indexes or reading indexes of FIFOs. As mentioned before, DF applications rely on a large number of requests to memories for firing rule checking. So, these indexes are used to determine either the number of available tokens corresponding to each reader actor or the free space in a FIFO. If data is required to be read from input FIFOs, the firing rule is satisfied by checking if the number of available tokens in all input FIFOs is equal or greater than the required number during computation. Whereas, if data has to be written to output FIFOs, the firing rule is satisfied by checking if all output FIFOs have sufficient empty room to accommodate the produced tokens. Hence, before processing an action, a PE

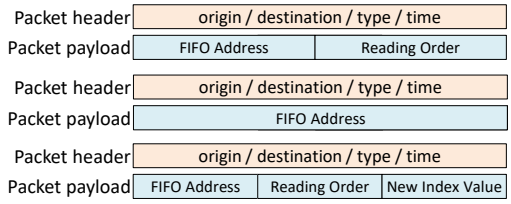


Fig. 4. Packet structures for holding reading (top) / writing (middle) index requests and set index (bottom)

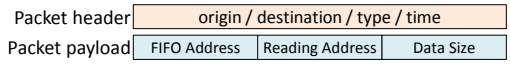


Fig. 5. The structure of the packets holding the reading requests

has to request the reading and/or writing indexes of input and output FIFOs. When the PE receives the value of demanded reading/writing index, it will check the satisfaction of the firing rule. After reading/writing data from/to a FIFO, the reading/writing index has to be incremented by the size of the transferred data. The PE, which consumes/produces data, has to set the new reading/writing index in the targeted FIFO after reading/writing operation is performed. Accordingly, four types of packets are utilized: (1) Request read index, (2) Request write index, (3) Setting read/write index, and (4) Holding read/write index.

As the FIFO may have several reading indexes corresponding to different reader actors, the PE has to determine the reading order of the actor and sends it in the payload of the packet. However, a FIFO has only one writer actor; hence, to attain the value of its writer index the PE has to send the FIFO address in the destination memory module. In both packets, the packet type, given in the packet's header, is used by the NI at the destination memory module to decode the request type. Fig. 4 depicts the structure of the FIP packets holding the requests of a reading index and writing index.

On the other side, whenever a memory module receives a request of reading/writing index it will retrieve its value from the specified FIFO and sends it back to the PE. The NI in the memory module will assemble a 1-flit payload packet as shown in Fig. 4.

In order to set the reading/writing index after finalizing the data transfer operations from/to a FIFO, the PE sends a control packet that notifies the FIFO about its new reading/writing index. It includes one flit that contains the FIFO address in the destination memory module, the reading order of the actor, and the new value of the reading index. Since the FIFOs have only one writer actor, the writing packet payload simply includes the address of the targeted FIFO in the destination memory module and the new value of the writing index.

g) Data reading request packets (DRP): Fig. 5 presents the packet holding the reading request of data from PE to memory module. Its payload consists of one flit that includes the address of the FIFO in the destination memory module, the starting address of reading, and the size of required data.

h) Code transferring packets (CTP): Actor binary codes are stored in a shared memory. When updating the actor mapping, the binary code referring to the moved actor should

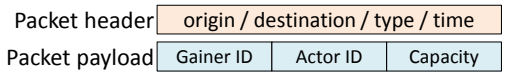


Fig. 6. Manager command requesting the transfer of the moved actor code

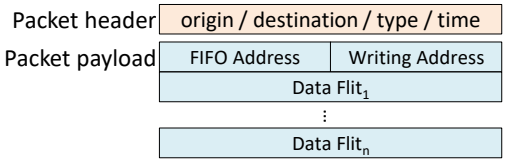


Fig. 7. The structure of packets carrying processed data

be transferred from the shared memory to the cache of the new PE. The manager sends a command of transferring the binary code in the form of a reading request packet. The sent request includes the actor ID, the address of the new PE, and the size of transferred data per packet (see Fig. 6).

2) Data packets: The second category of packets refers to the ordinary flow of data between PEs and memory modules. These packets, described hereafter, carry data that is either processed in a PE and written in a memory module or sent from a memory module as a response to a PE reading request.

a) Dataflow packets (DFP): The DFPs encompass all packets transferred between PEs and the FIFOs distributed in the memory modules. They carry data that is either processed in a PE and will be stored in a FIFO or sent from a FIFO as a response to a PE reading request. Fig. 7 presents the structure of packets carrying processed data in their payloads.

b) Binary code packets (BCP): The BCPs aim to transfer the binary code from the shared memory to the cache memory of the new PE. Note that the binary code is divided into sections of reasonable sizes which are transferred consequently. The size of the transferred data (payload capacity) is specified by the manager according to the monitored traffic in the network and based on the required cache lines to be filled before launching the actor on the new PE. For example, the packet including in its payload 64 flits of 32-bitwidth transfers 256 bytes which form 4 lines of L1 cache.

IV. Processing Flow

A. Initial mapping

Initially, the actors are mapped randomly to the PEs, or can be mapped using the exact method presented in [20]. FIFOs are mapped randomly and are approximately equally distributed on all memory blocks. The manager informs by means of packets all involved PEs. For each PE in charge of executing actors, the manager generates and sends its corresponding mapping information in a separate packet (M_pIP). Packets holding the mapping information are stored in a predefined location in MIM. Then, the involved PEs are notified to retrieve their mapping information from the shared memory using notifying packets. At this stage, the manager waits the PEs, which are incorporated in processing a specific number of video frames N_F to send their monitoring information. Note that N_F is set originally to a default value and may be changed dynamically by the manager.

TABLE IV
PARAMETERS AND VARIABLES USED FOR THE MAPPING ALGORITHM

Parameter	Definition
DPN application graph (DPNapp)	
$\|\mathbb{A}\|$	Number (Nb) of actors
$\|\mathbb{F}\|$	Nb of FIFO channels
$\|\mathbb{K}\|$	Nb of data packets
$\|\mathbb{I}_c\|$	Nb of input ports of actor A_c
Architecture graph (arch)	
$\|\mathbb{P}\|$	Nb of processing elements
$\|\mathbb{M}\|$	Nb of memory modules
Profiling data (profile)	
R_i	Mean number of firings of actor i
W^i	Total computation cost of actor i
Cs^i	Instruction code size of actor i

Before receiving the notification packet about initial mapping of actors, all PEs are in idle state. Once it receives the notification packet, the PE sends a request to retrieve the mapping information which includes IDs of actors to be executed, IDs of input and output FIFOs for each actor, and the reading order of each input FIFO. The mapped actors are scheduled according to the order sent from the manager and the PE begins to execute them in round-robin manner.

B. Monitoring actor execution

The execution of actors continues until receiving a new notification packet about changing the mapping information. All involved PEs monitor their running actors during the processing of N_F video frames, which determine the observation window. Precisely, each PE node accumulates for every mapped actor A_c its communication time $T_{cm}[A_c]$, computation time $T_{cp}[A_c]$, and total number of tokens received to each input port $NT_{total}[A_{c_{[I_j]}}]$ where $c \in \{1, ..., |\mathbb{A}|\}$ and $j \in \{1, ..., |\mathbb{I}_c|\}$. In addition, the adapter, which is embedded in the NI of each node n (processor or memory), extracts from each received packet carrying processed data, the following information for each source S_i: (1) the total number of transferred tokens from S_i to n: $NT^n_{total}[S_i, n]$ and (2) the average time delay consumed per token to reach the node n from source S_i: $T_{av}[S_i, n]$.Table IV gathers the variables and parameters used to formalize our mapping approach.

The total number of transferred tokens is simply determined. First, input packets are classified according to their sources S_i. Then, their corresponding sizes $size_{P_k}[S_i]$, which reflect the number of data-flits, are accumulated.

$$NT^n_{total}[S_i, n] = \sum_{k=1}^{\mathbb{K}} size_{P_k}[S_i, n] \tag{1}$$

where $i \in \{1, ..., |\mathbb{P}| + |\mathbb{M}|\}$.

The average time delay per token per each source $T_{av}[S_i, n]$ is calculated by dividing the time delay of each token transferred from S_i by $NT_{total}[S_i, n]$.

$$T_{av}[S_i, n] = \frac{\sum_{k=1}^{\mathbb{K}} size_{P_k}[S_i, n] \times D_{P_k}[S_i, n]}{NT^n_{total}[S_i, n]} \tag{2}$$

where $k \in \{1, ..., |\mathbb{K}|\}$.

$D_{P_k}[S_i]$ is determined by embedding, at the source node, for each packet P_k its sending time-stamp $T_s[P_k]$ in its header then subtracting it from the reception time $T_r[P_k]$ at the destination node. All tokens in a packet are considered to have the same delay.

$$D[P_k] = T_r[P_k] - T_s[P_k] \tag{3}$$

C. Collecting monitoring information

When the number of the processed frames meets the observed window, each PE node generates its own monitoring information packet. The packet is then sent to the MIM module (presented in III-C). Directly, the accumulated values are reset with the beginning of the new observation window. Then, the PE continues executing the previously mapped actors according to the adopted circular order. This guarantees that the remapping does not impose any additional overhead in terms of latency. The MIM module notifies in its turn the manager when new monitoring data is available corresponding to a specific processor throughout a notification packet (III-D1a). Whenever a new notification packet is received by the manager, the latter directly requests to retrieve the new available monitoring data. Also, the manager requests using command packets from all memory modules to send their monitoring information. Note that memory modules respond to the manager and send the requested data directly without any notification process since the adopted MoC allows the direct communication between a memory and a processor. All received monitoring packets are disassembled and their contents are parsed and saved in the manager local registers.

When the feedback data is collected from all modules incorporated in processing the video frames, the manager applies the run-time remapping algorithm. At this stage, the manager owns locally the following data: (1) the communication time of each actor: $T_{cm}[A_c]$, (2) the computation time of each actor: $T_{cp}[A_c]$, (3) the number of input tokens corresponding to every input port of all actors: $NT^a_{total}[A_{c_{[I_j]}}]$, (4) the number of incoming tokens to each processor module from each memory module m: $NT^n_{total}[S_m, p]$, (5) the average communication delay of received tokens to each processor p from each memory module m: $T_{av}[S_m, p]$, (6) the number of incoming tokens to each memory module m from each processor module p: $NT^n_{total}[S_p, m]$, and (7) the average communication delay of received tokens to each memory module m from each processor module p: $T_{av}[S_p, m]$ where $c \in \{1, ..., |\mathbb{A}|\}$, $j \in \{1, ..., |\mathbb{I}_c|\}$, $m \in \{1, ..., |\mathbb{M}|\}$ and $p \in \{1, ..., |\mathbb{P}|\}$.

D. Estimating NoC communication time delay

Communication time delay is a critical factor in HMPSoC platforms using NoCs. The communication time of the moved actor is affected by the location of the new hosting PE in the network. NoC time-delay estimation impacts directly the prediction process of the communication time of the moved actor. Hence, the accuracy level in estimating the delay latency changes the decision on the actor move in the RR algorithm. In this work, two novel methods have been proposed to estimate the communication time delay for transferring one token in the

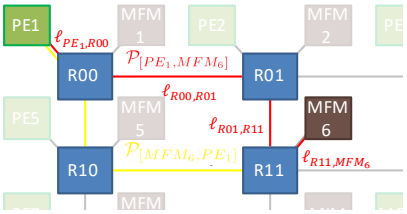


Fig. 8. Example on path declaration in the NoC

NoC. The first method is called the average-path token delay and it is based on finding the average delay for transferring one token depending on the path delays between all nodes of the NoC. The second is called the average-link token delay and considers the time-delay of the token according to the used physical links connecting the NoC components while transferring the token. Both proposed methods make use of the monitoring data, which is collected while processing N_F video frames in the previous observation window. The techniques used in estimating the NoC communication time-delay are described in the following subsections.

1) Average-path token delay (APTD): In this approach, a path is considered to be formed from the set of the interconnections between two specific nodes. As an example, Fig. 8 illustrates in red the path $\mathcal{P}_{[PE_1,MFM_6]}$ between processing element PE_1 to memory module MFM_6. As a deterministic routing is applied in this work, the packets always use the same path between the source node and the destination node. Since the adopted MoC forbids the transfer of packets in between memory modules and in between PEs, the active paths are those connecting either memory modules to PEs or PEs to memory modules. Note that the packets transferred from a processing element p to a memory module m do not follow the same path used in transferring packets from the memory module m to the processing element p. Fig. 8 illustrates in red the followed path to transfer packets from PE_1 to MFM_6 and in yellow the followed path to transfer packets from MFM_6 to PE_1. In APTD, the manager calculates the average path delay per token T_{av} in several steps as shown in Algo. 1. T_{av} refers to the average time delay required to transfer one token from the source node to the destination node, regardless of the path between the source and destination nodes. As an example, the average time delay of all tokens transferred through either the path $\mathcal{P}_{[PE_1,MFM_6]}$ or the path $\mathcal{P}_{[MFM_6,PE_1]}$ (Fig. 8) is considered equal regardless of the number of links constituting each path and the corresponding traffic in each link and the switch conflicts in the connecting routers. T_{av} is computed by dividing the sum of the communication-time delays D_{total} by the total number of transferred tokens in the network NT_{total}:

$$T_{av} = \frac{D_{total}}{NT_{total}} \tag{4}$$

The manager benefits from the collected monitoring data. It makes use of the number of input tokens $NT^n_{total}[S_i, n]$ transferred to each destination node n from each source node S_i to determine the total number of all transferred tokens in the network (NT_{total}) as presented in (5):

$$NT_{total} = \sum_{n=1}^{|\mathbb{P}|+|\mathbb{M}|} \sum_{i=1}^{|\mathbb{P}|+|\mathbb{M}|} NT^n_{total}[S_i, n] \tag{5}$$

Also, the communication-time delays for all tokens transferred in the network are accumulated. The sum of the communication delays D_{total} is determined according to (6):

$$D_{total} = \sum_{n=1}^{|\mathbb{P}|+|\mathbb{M}|} \sum_{i=1}^{|\mathbb{P}|+|\mathbb{M}|} T_{av}[S_i, n] \times NT^n_{total}[S_i, n] \tag{6}$$

where $T_{av}[S_i, n]$ is the collected average time delay required to transfer one token from the source node S_i to the destination node n.

2) Average-link token delay (ALTD): A link is defined as the interconnection between two consecutive components of the NoC: Router, Memory and PE. As an example, Fig. 8 shows the links constituting the path $\mathcal{P}_{[PE_1,MFM_6]}$. In this approach, the average communication time delay per token is determined for each link as shown in Algo. 2. The total communication-time delay in a path $\mathcal{P}_{[S_i,n]}$ connecting the source node S_i and the destination node n is determined from the monitored data as shown in (7):

$$D^{\mathcal{P}}_{total}[S_i, n] = T_{av}[S_i, n] \times NT^n_{total}[S_i, n] \tag{7}$$

Each path is segmented into a set of links $\mathbb{L}_{\mathcal{P}_{[S_i,n]}}$. The average communication time delay per link $D^{\mathcal{L}}_{av}[S_i, n]$ in the path $\mathcal{P}_{[S_i,n]}$ is determined as follows:

$$D^{\mathcal{L}}_{av}[S_i, n] = \frac{D^{\mathcal{P}}_{total}[S_i, n]}{NL[S_i, n]} \tag{8}$$

where $NL[S_i, n]$ is the number of links constructing the path $\mathcal{P}_{[S_i,n]}$. Here, the links constructing a path are assumed to have similar contribution in the total communication time delay monitored in the path. As a link l is shared among different paths, the total link communication-time delay $D_{total}[l]$ is the sum of all average communication-time delay per link computed in all paths in which link l constitutes one of their interconnections:

$$D_{total}[l] = \sum_{n=1}^{|\mathbb{P}|+|\mathbb{M}|} \sum_{i=1}^{|\mathbb{P}|+|\mathbb{M}|} D^{\mathcal{L}}_{av}[S_i, n] \ni l \in \mathbb{L}_{\mathcal{P}_{[S_i,n]}} \tag{9}$$

On the other hand, the tokens passing through a path are definitely passing through all links constructing the path. Hence, the total number of tokens $NT_{total}[l]$ passing through a link l is the sum of all tokens passing through all paths, which link l constitutes one of their interconnections:

$$NT_{total}[l] = \sum_{n=1}^{|\mathbb{P}|+|\mathbb{M}|} \sum_{i=1}^{|\mathbb{P}|+|\mathbb{M}|} NT_{total}[S_i, n] \ni l \in \mathbb{L}_{\mathcal{P}_{[S_i,n]}} \tag{10}$$

The average communication-time delay per token $T_{av}[l]$ for each link l is determined by dividing the accumulated communication-time delay $D_{total}[l]$ by the number of tokens

Algorithm 1 Average-path token delay (APTD)

Step 1: *Find the sum of the communication delays* D_{total}
Step 2: *Find the total number of all tokens* NT_{total}
Step 3: *Calculate the average time delay per token* T_{av}

Algorithm 2 Average-link token delay (ALTD)

Step 1:
for each path $\mathcal{P}_{[S_i,n]}$ **do**
a- *Find the total communication-time delay* $D^{\mathcal{P}}_{total}[S_i, n]$
b- *Calculate average communication time delay per link* $D^{\mathcal{L}}_{av}[S_i, n]$
end for
Step 2:
for each link l **do**
a- *Find the total link communication-time delay* $D_{total}[l]$
b- *Find the total number of tokens* $NT_{total}[l]$
b- *Calculate the average communication time delay per token* $T_{av}[l]$
end for

$NT_{total}[l]$ passing through this link.

$$T_{av}[l] = \frac{D_{total}[l]}{NT_{total}[l]} \tag{11}$$

E. Applying RR algorithm

For each observation window (N_F frames), the manager executes at run-time the RR algorithm, which is divided into two main steps. The first step is dedicated to find all possible candidate actors which their moves would enhance the overall throughput. The second step sets a tradeoff between the cost of migration and the predicted improvement of the performance.

1) Specify the possible candidate actors: In this work, the definitions of the terms period of each processor p ($Period_p$), maximum period ($Period_{max}$) and throughput (Th) have been adopted as introduced in [17]. $Period_p$ is the sum of total computation time $compT_p$ and total communication time $commT_p$ recorded during N_F video frames:

$$Period_p = compT_p + commT_p \qquad \forall p \in \mathbb{P} \tag{12}$$

where $compT_p$ and $commT_p$ of processor p are the sums of the computation times and of the communication times respectively of all actors which are mapped on this processor:

$$compT_p = \sum_{k:\mathbb{P}[k]=p} T_{cp}[A_k] \qquad \forall p \in \mathbb{P} \tag{13}$$

$$commT_p = \sum_{k:\mathbb{P}[k]=p} T_{cm}[A_k] \qquad \forall p \in \mathbb{P} \tag{14}$$

The throughput is defined as the inverse of the maximum period over all processors.

Hence, the first task is to find the PE with the maximum period. The manager computes the periods of all PEs during the current observation window of N_F video frames. Later, a simple comparison between all obtained period values is performed in order to specify the processor with the maximum period. The processor with the maximum period ($Period_{max}$) is nominated as *looser* processor. The algorithm used to determine the *looser* processor is outlined in Algo. 3. The set of candidate actors to be moved $\mathbb{C}$ includes the actors that have been previously executed by the *looser* processor. Fig. 9 demonstrates an example of $Period_p$ and $Period_{max}$. The figure shows three PEs (PE_1, PE_2 and PE_3) that run six actors (A_1, A_2, A_3, A_4, A_5 and A_6). In this example, PE_1 has the largest period, thus it is selected as the *looser* processor.

Algorithm 3 Finding processor with maximum period

$Period_{max} \leftarrow 0$
$looser \leftarrow \phi$
for $p \in \mathbb{P}$ **do**
if $Period_{max} < Period_p$ **then**
$Period_{max} \leftarrow Period_p$
$looser \leftarrow p$
end if
end for

2) Decision of the actor move: The actor selected to be moved should have a maximum total gain. According to the collected monitoring values, the manager estimates the total gain achieved for all combinations of mapping the actors which belongs to the candidate list $\mathbb{C}$ onto all available PEs. The estimated total gain $Gain^e_{total}[C_{A_c,p}]$ of a mapping combination $C_{A_c,p}$, which corresponds to moving A_c to p, is computed by finding the difference between the estimated performance gain $Gain^e_{per}[C_{A_c,p}]$ and the estimated migration cost of the actor $Cost^e_{mig}[C_{A_c,p}]$. The mapping combination that leads to the maximum estimated total gain is then selected. The engaged processor and actor are specified and so-called the *gainer* processor and *moved-actor* respectively.

a) Estimated performance gain: For each actor A_c in the candidate list $\mathbb{C}$, the manager considers it is moved virtually to all PEs except the *looser* processor. For each virtual-move combination, the manager estimates the achieved period of each processing element $Period^e_p[C_{A_c,p}]$. The new period of processor p is estimated by adding to the processor period $Period_p$ the estimated communication time $T^e_{cm}[C_{A_c,p}]$ and the estimated computation time $T^e_{cp}[C_{A_c,p}]$ of the moved actor A_c as shown in the following expression:

$$Period^e_p[C_{A_c,p}] = Period_p + T^e_{cp}[C_{A_c,p}] + T^e_{cm}[C_{A_c,p}] \tag{15}$$

Note that the tokens, which are consumed by a certain reader actor running on a processing element PE_R, are imported from a FIFO f. These tokens are previously generated by another actor running on another processing element PE_W. The generated tokens are first stored in a FIFO f and then transferred once requested to the processing element PE_R where the reader actor is executed. Hence, the tokens pass

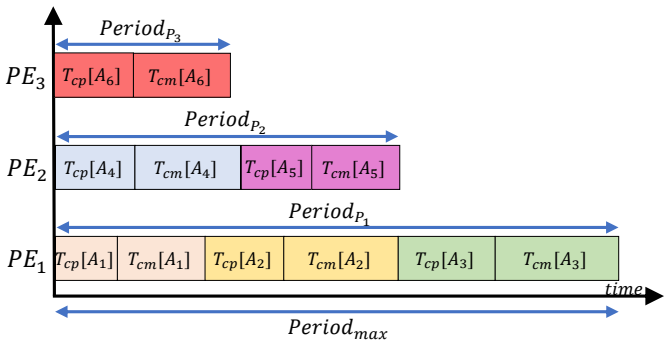


Fig. 9. An example of $Period_p$ and $Period_{max}$

through two paths. The first path $\mathcal{P}_{[PE_W,MFM_f]}$ connects the processing element PE_W, which executes the writer actor, and the memory module that accommodates the FIFO f. On the other hand, the second path $\mathcal{P}_{[MFM_f,PE_R]}$ connects the memory module that accommodates the FIFO f and the processing element PE_R which executes the reader actor. The communication-time delays in both paths are considered when estimating the communication time of the moved actor.

When adopting APTD method for determining the communication delay in the NoC, the estimated communication-time delay per input j for each actor A_c is equal to the total number of input tokens $NT_{total}S[A_{c_{[I_j]}}]$ transferred to the actor at this input multiplied by the double of the calculated average path communication-time delay per token T_{av} (4). The average path delay per token is doubled to compensate the time delay of the two paths $\mathcal{P}_{[PE_W,MFM_f]}$ and $\mathcal{P}_{[MFM_f,PE_R]}$. The total estimated communication time is the sum of all estimated communication-time delays of all inputs:

$$T^e_{cm}[C_{A_c,p}] = \sum_{j=1}^{|\mathbb{I}_c|} 2 \times T_{av} \times NT_{total}[A_{c_{[I_j]}}] \quad (16)$$

Note that the adopted model of computation forbids the transfer of tokens in between actors (running on PEs) directly without passing through a FIFO (allocated in a memory module MFM_f). Hence, tokens produced by the writer actor (running on PE_W) will pass through two paths ($\mathcal{P}_{[PE_W,MFM_f]}$ and $\mathcal{P}_{[MFM_f,PE_R]}$) before arriving to the reader actor (running on PE_R). The exact number of tokens passes through both paths while considering same average path delay per token T_{av}. So, the average path delay per token is doubled in (16).

When adopting ALTD method, the estimated communication-time per input j is equal to the total number of input tokens $NT_{total}[A_{c_{[I_j]}}]$ transferred to the actor A_c through this input multiplied by the sum of all average communication-time delay per token $T_{av}[l]$ for each link l constructing the paths which the input tokens use to reach the processing element running the actor A_c. The total estimated communication time will be the sum of all estimated communication-time delays of all inputs:

$$T^e_{cm}[A_c] = \sum_{j=1}^{|\mathbb{I}_c|} \left(\sum_{i=1} T_{av}[l_i] \right) \times NT_{total}[A_{c_{[I_j]}}] \quad \ni l_i \in \left\{ \mathbb{L}_{\mathcal{P}_{[PE_W,MFM_f]}} \cup \mathbb{L}_{\mathcal{P}_{[MFM_f,PE_R]}} \right\} \quad (17)$$

In addition, the estimated computation time $T^e_{cp}[C_{A_c,p}]$ of the moved actor A_c is determined depending on the recorded computation time of the moved actor A_c during the previous mapping $T_{cp}[A_c]$ and the estimated total speed-up ratio $SU^e_{total}[C_{A_c,p}]$, which is achieved when moving A_c to p:

$$T^e_{cp}[C_{A_c,p}] = T_{cp}[A_c] \times SU^e_{total}[C_{A_c,p}] \quad (18)$$

such that

$$SU^e_{total}[C_{A_c,p}] = \frac{\mathcal{A}_{A_c}[p]}{\mathcal{A}_{A_c}[looser]} \times \frac{f[looser]}{f[p]} \quad (19)$$

where $f[p]$ is the operating frequency of processor p (Table II) and $\mathcal{A}_{A_c}[p]$ is the acceleration enhancement ratio of the moved actor A_c when running on processor p (Table I).

Note that for all mapping combinations, the period of the *looser* processor is modified when an actor A_c is supposed to be mapped to another processor p. Hence, it is updated by subtracting the actual communication time $T_{cm}[A_c]$ and the actual computation time $T_{cp}[A_c]$ of the moved actor A_c:

$$Period^e_{looser}[C_{A_c,p}] = Period_{max} - T_{cm}[A_c] - T_{cp}[A_c] \quad (20)$$

For each mapping combination, the manager determines the maximum estimated period $Period^e_{max}[C_{A_c,p}]$ which denotes the maximum period among all processors when actor A_c is mapped to processor p. Fig. 10 demonstrates an example of finding $Period^e_{max}[C_{A_c,p}]$. The figure considers the example illustrated in Fig. 9. Three actors are mapped to the *looser* processor PE_1. The candidate list $\mathbb{C}$ includes three actors: A_1, A_2 and A_3. Six mapping combinations are illustrated: C_{A_1,PE_2}, C_{A_1,PE_3}, C_{A_2,PE_2}, C_{A_2,PE_3}, C_{A_3,PE_2} and C_{A_3,PE_3}. The figure shows how to find the maximum estimated period $Period^e_{max}[C_{A_c,p}]$ for each mapping combination. It is shown in the figure that both the estimated communication time and estimated computation time of the same actor differ when mapped to different PEs.

These computed new periods are then used to find the performance gain related to each mapping combination:

$$Gain^e_{per}[C_{A_c,p}] = Period^e_p[C_{A_c,p}] - Period_{max} \quad (21)$$

b) Estimated migration cost: The migration cost of an actor is the required time to transfer its binary code into the local memory of the new hosting processing element. It depends on the size of the binary data required to be transferred and the communication-time delay in the network. The sizes of the binary codes of all actors are considered to be known by the manager in terms of number of flits. Accordingly, the migration cost of the moved actor is estimated by the manager using the estimated NoC communication-time delay. When adopting APTD method, the estimated migration cost related to the moving of actor A_c to processor p is calculated as expressed in (22):

$$Cost^e_{mig}[C_{A_c,p}] = size_{bin}[A_c] \times T_{av} \quad (22)$$

where $size_{bin}[A_c]$ is the size of the binary code of actor A_c and T_{av} is the average path communication-time delay per token (4). When ALTD method is adopted, the migration cost of the moved actor A_c is determined by (23):

$$Cost^e_{mig}[C_{A_c,p}] = size_{bin}[A_c] \times \left(\sum_{i=1} T_{av}[l_i] \right) \quad \ni l_i \in \mathbb{L}_{\mathcal{P}_{[BCM,p]}} \quad (23)$$

c) Estimated total gain: The manager computes the total gain estimated to be achieved for all mapping combinations by finding the difference between the estimated performance gain $Gain^e_{per}[C_{A_c,p}]$ and the estimated migration cost of the actor $Cost^e_{mig}[C_{A_c,p}]$.

$$Gain^e_{total}[C_{A_c,p}] = Gain^e_{per}[C_{A_c,p}] - Cost^e_{mig}[C_{A_c,p}] \quad (24)$$

The moving of an actor would lead to permanent performance gain and the migration cost is paid once. However, the estimated performance gain takes the cost of migration into account in order to aggravate the probability of enhancing the overall performance directly after applying the move (in the next observation window). In fact, the variation of the input data and its corresponding effects on executing the

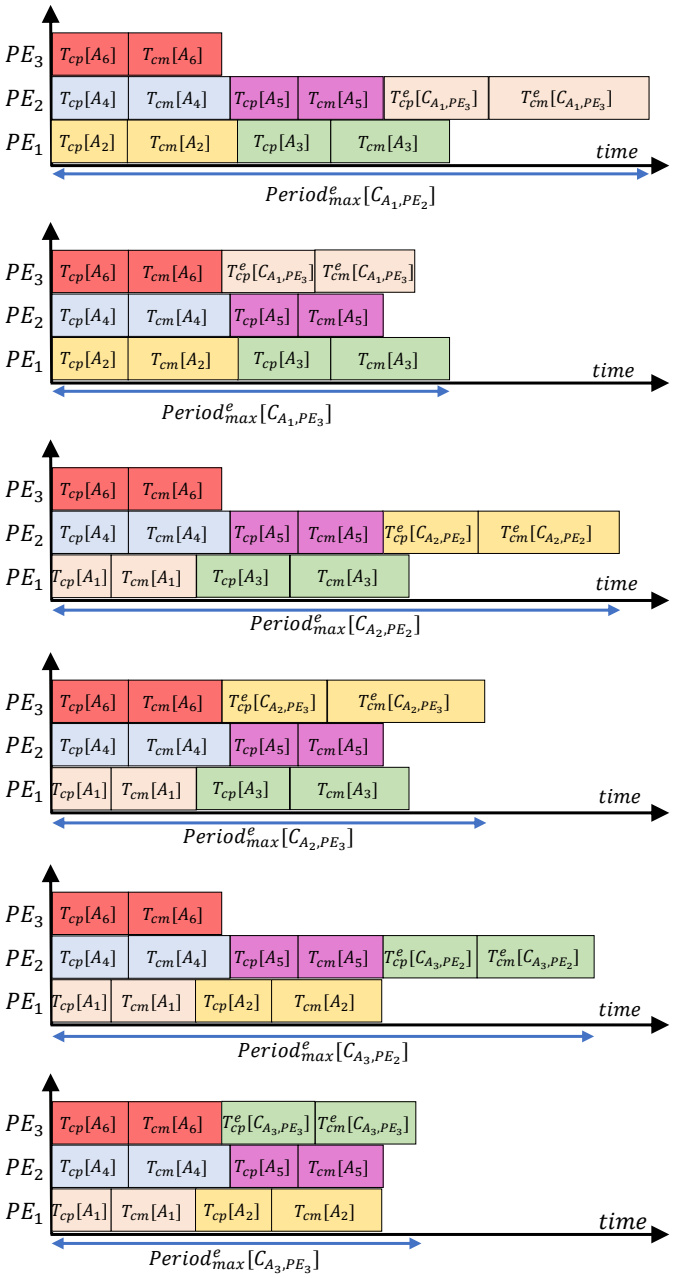


Fig. 10. An example of finding the maximum periods for each mapping combination

involved actors incites to consider worst case (severe) decision where the performance enhancement should be guaranteed once moving the actor.

Then, the manager finds the maximum achieved total gain among all mapping combinations and accordingly specifies the actor to be moved and the *gainer* processing element.

F. Moving the actor to the gainer processor

The PE, after finishing the execution of the current running actor, retrieves the new mapping information and sends directly a confirmation packet so that the manager processor manages the transfer of the object code corresponding to the new mapped actor. Before running the moved actor, the PE checks the availability of the object file corresponding to the actor in its cache memory. Note that for the initial mapping, the manager generates and sends packets to all PEs in charge of executing actors. Whereas, after executing the RR algorithm, the manager informs only the *gainer* and *looser* processors. This procedure reduces the traffic in the network and maintain the processing performance since the PEs that are not affected by remapping process are not disturbed. In fact, the manager informs first the *looser* processor about the new mapping information. Then, it waits until the *looser* processor confirms the well reception. The *looser* processor sends a confirmation packet to the manager whenever it finishes the execution of the moved actor. When the manager receives the confirmation packet, it sends the new mapping information to the *gainer* processor. Later, the *gainer* sends a confirmation packet to the manager that directly manages the transferring of the object code of the mapped actor from the shared memory into the cache memory of the *gainer* processor by making use of BCPs described in subsection III-D2b. This guarantees that the actor is executed by only one PE in the whole platform and ensure better controlling of the traffic while migrating the binary codes. In fact, the manager sends a CTP (subsection III-D1h) which includes the ID of the *gainer* processor, the ID of the moved actor and the size of the BCPs (capacity) as described in subsection III-C1. After receiving the CTP, the MAM module, which is integrated into the NI of the BCM (subsection III-C1), manages retrieving the binary code from the shared memory and dividing it into sections according to the capacity specified by the manager. The generated BCPs will be transferred to gainer processor. In our work, we consider that the gainer processor can start executing the actor once at least 256 bytes, which construct 8 lines of the L1-I cache, are received and stored to the *gainer* processor local memory. The hierarchy of the PEs' local memories includes L1 and L2 caches. L1 cache is broken up into to halves, instruction (L1-I) and data (L1-D) each of 32KB. L2 cache size is of 256KB and is used for instructions and data.

Algorithm 4 Run-time Remapping (RR)

Step 1: *Calculate the period of each PE*
Step 2: *Find PE with Max. period and assign it as looser*
Step 3: *Find the total gain (performance - migration cost)*
for each move **do**
  a- *Find the performance gain*
    . find the period for each PE
    . find the maximum period
  b- *Find the migration cost*
  c- *Calculate the total gain*
end for
Step4: *Choose the move with Max. positive total gain*

G. RR Algorithm Complexity

The devised algorithm consists of several steps summarized in Algo. 4. The complexity of each step is illustrated to determine the overall complexity. The complexity of Step1,

the step of finding the period of each processor, is $O(|\mathbb{P}|)$. Then, Step2, the step of finding the processor with maximum period has the complexity of $O(|\mathbb{P}|)$. The complexity of Step3, estimating the total gains corresponding the move of the candidate actors to all PEs rather than the looser processor, is $O((|\mathbb{P}|-1).|A_c \in \mathbb{C}|)$. The complexity of Step4, choosing the best move, is $O(|A_c \in \mathbb{C}|.(|\mathbb{P}|-1))$. If we consider a well balanced distribution of actors among the processors at initiation ($|A_c \in \mathbb{C}| \approx \frac{|\mathbb{A}|}{|\mathbb{P}|}$), the overall complexity becomes $O(|\mathbb{P}|) + O(|\mathbb{A}|)$ knowing that $\frac{|\mathbb{P}|-1}{|\mathbb{P}|} \approx 1$. Note that the algorithm is computed when all monitoring data is collected, so the maximum rate is once per execution of the whole data flow, and in practice can be tuned to be slower. With respect to the complexity and the execution rates of actors, this complexity is extremely low.

V. Experiments and Results

A. Application Model

In this work we target the multimedia application domain. We adopt the well-known MPEG4 part 2 Simple Profile video decoder (MPEG4-SP). This multimedia application is typically used in de-compression of encoded video digital data. Fig. 11 presents the structure of decoder as described in Reconfigurable Video Coding framework (RVC) [3] [33].

MPEG4-SP is specified with heterogeneous dataflow MoCs and includes up to 40% of dynamic actors [34]. It is composed of 41 actors and 70 FIFOs specified in RVC-CAL language. The ORCC tool is utilized for compiling and software synthesis [3] and we make use of the generated C-code for multi-core platforms. We also use the structure of the software FIFO presented in Fig. 1-b), which is generated by ORCC.

A FIFO may have several reader actors but only one writer actor. It opts an indexing mechanism such that a specific index is assigned to each reader or writer actor. These indexes are used to determine the number of available tokens corresponding to each reader actor and the free space in a FIFO. The number of available tokens ($T_f[R_i]$) in a FIFO (f) is the difference between the reader index ($I_f[R_i]$) and the writer index ($I_f[W]$). The free space in a FIFO is the number of memory addresses that contain no more needed data from all reader actors. In other words, it is the subtraction of the maximum available tokens from the total FIFO size ($Size_f$).

Each actor has its input and output ports and includes one or several actions. An action describes a specific functionality and is executed (fired) when a set of conditions, so-called firing rules, are satisfied. As an example, a firing rule consists of checking if the number of available tokens in the input FIFO is greater than the required number for computation, and that the output FIFO has sufficient empty room to store the produced tokens. In MPEG4-SP, the number of reader actors ranges from 1 (at least) to 6 (at most).

MPEG RVC defines RVC-CAL applications as dynamic dataflow applications, where the uncertainty of computing due to data-dependency prevents from any static scheduling. They are based on dataflow process network (DPN) model [6]. In such model, the actor executes when at least one of its firing rules is satisfied. For cases where several firing rules

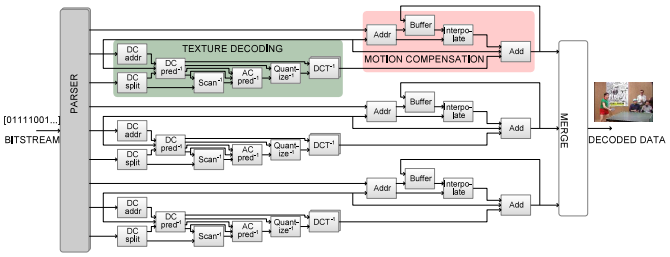


Fig. 11. MPEG4 part 2 SP decoder [33]

are satisfied simultaneously, only one is selected according to its priority. Consequently, its corresponding satisfied action is fired. Each firing consumes input tokens and produces output tokens. The number of the consumed or produced tokens may be fixed or variable.

B. Experimental framework and setup

In order to assess the feasibility of our proposed run-time remapping method, we developed a real-time simulator. The simulator is described in SystemC TLM model [35]. The devised simulator models a MPSoC platform using NoC concept for interconnecting embedded modules. The platform incorporates heterogeneous processing elements (Table I), memory blocks, and the manager. The simulator platform has been designed with hierarchical modules that can work concurrently and intercommunicate via ports using simple or complex communication channels. SystemC features have been exploited to mimic the accurate functionality of the modules described in section III.

The adopted NoC-based architecture, presented in section III, is implemented in the devised simulator platform. In order to accurately model the adopted application, all involved actions are functionally simulated to determine their execution-timing features and generate the real data exchanged by actors during video decoding. The SystemC model adopted in the simulation platform is cycle accurate at the level of the NoC and the network interfaces. The timing of all corresponding action executions on PE is compensated in the simulation according to the profiling data extracted while running the application on a reference computer. Profiling data provides, for each involved action, the mean value of the number of cycles required to execute it. In this work, profiling data has been extracted using on a desktop computer (i7-2620M CPU@2.7 GHz and 8GB memory). We consider that the NoC operating frequency f is 500 MHz. The clock cycle in each PE is determined according to Table II. During SystemC simulations, for each fired action, its corresponding execution time determined in profiling is mapped according to the processor frequency and used as time delay to compensate the real execution time. In addition, several benchmark video sequences with different formats from [36] have been encoded. The selected video sequences have different manner in changes between successive frames. This guarantees to evaluate the performance of the proposed algorithm for different data-dependent behaviors. The resultant data has been used as input to the decoder. These same encoded videos have been

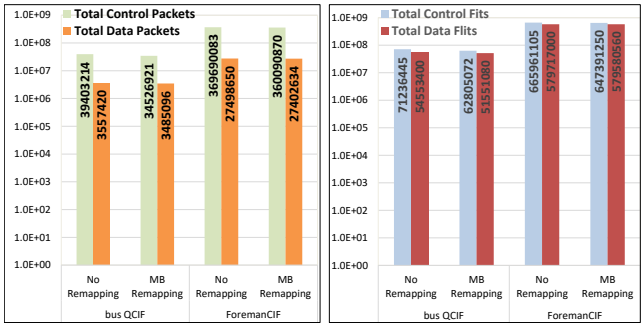


Fig. 12. Classification of transported packets and flits

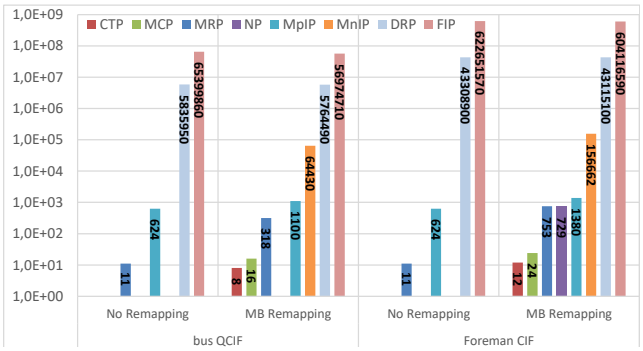


Fig. 13. Classification of control flits according to their types

decoded on a desktop computer and the FIFO contents have been traced over the decoding period. To verify the proper functionality of each actor, the contents stored in the FIFOs in the simulator have been compared to the traced FIFO data. Also, the output data of the simulator have been reconstructed into visual video in order to verify the functionality of the devised simulator. The video sequences have been decoded without applying remapping targeting the same NoC-base architecture and the obtained results have been compared to that obtained when the video sequences are decoded adopting the MB remapping algorithm applying ALTD and APTD for estimating the communication time delay while considering an observation window of $N_f = 10$.

C. Experimental Results

1) Transported data: The number of packets that travel through the network during the decoding of the video sequences, and their corresponding flits are recorded in the case of applying the MB remapping and the case of decoding the video without remapping. Fig. 12 presents the number of transported packets and flits in logarithmic scale during decoding the Foreman video with CIF format and Bus video with QCIF format for the case of adopting MB remapping algorithm and the case of ordinary decoding. The packets and flits are classified into control and data categories. The figure shows that the flits of control packets form about 53% of all transported flits in the two cases.

Furthermore, investigating thoroughly the types of transported control flits illustrates that 93% of control flits belong to FIP. This refers to the MoC adopted in dataflow applications which requires checking the firing rules (availability of input data and output buffer space). Fig. 13 shows in logarithmic scale the number of each type of control flits transported while decoding the Foreman video sequence in CIF format and Bus video sequence in QCIF format for the case of MB remapping and the case of ordinary decoding.

Also, Fig. 13 shows that additional flits are transported in the network due to the remapping. In fact, applying remapping induces additional control and data packets. In order to evaluate the effect of applying the MB remapping algorithm on the traffic in the network, the transported flits are classified into two main categories. The first category includes the flits which are used basically for dataflow. This category encompasses the flits which occupy the payload of

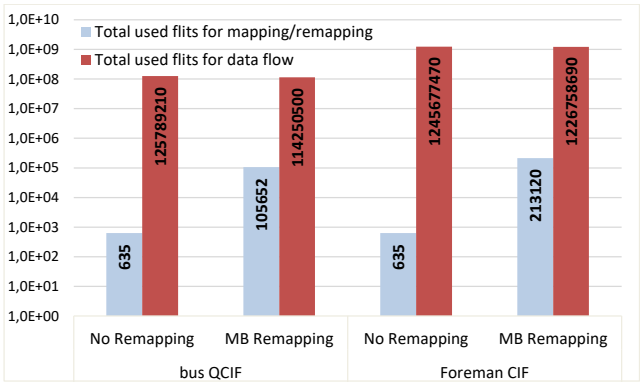


Fig. 14. Number of transported flits

TABLE V
PERCENTAGE OF FLITS TRANSPORTED IN BCP FROM TOTAL FLITS

Video		Remapping Algorithm	
Sequence	Format	MB-ALTD	MB-APTD
Foreman	CIF	0.0044%	0.0067%
Bus	CIF	0.0008%	0.0125%
Ice	4CIF	0.0027%	0.0019%
Bus	QCIF	0.0348%	0.0272%

all FIP, DRP and DFP. The second category includes the induced flits by applying the remapping algorithm. Hence, the second category compromises the flits listed in the payloads of NP, MRP, MCP, M_nIP, M_pIP, CTP, and BCP. Note that both categories include data and control packets. Fig. 14 illustrates the comparison summary in terms of the number of transported flits of both categories. In the figure, the number of transported flits, which is obtained while processing the Foreman video with CIF format and Bus video with QCIF format, is presented in logarithmic scale for both cases (decoding while applying remapping algorithm and ordinary decoding). The comparison shows that the additional flits induced by applying the MB algorithm forms less than 0.02% from total transported flits. In addition, Table V shows the percentage of flits transporting the binary code of migrated actors from the total number of transported flits in the network while decoding several video sequences. The presented percentages illustrate that the impact of actor migration on the traffic is negligible.

2) Packet time-delay: The packet time-delay is recorded while decoding the video sequences, following the procedure explained in subsection IV-B. Fig. 15 presents the variation

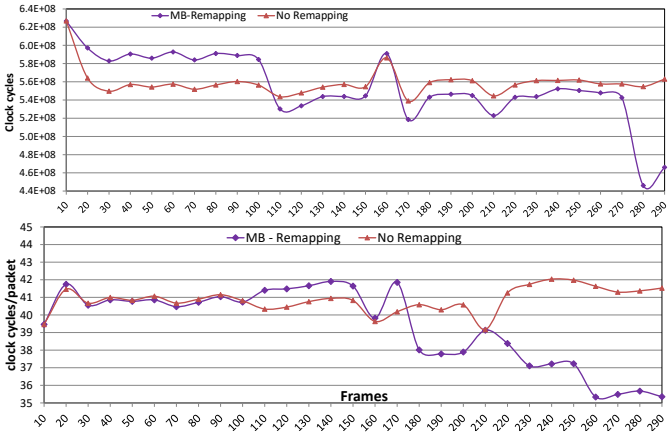


Fig. 15. Sum of packet time-delays (top) and average packet time-delay (bottom) while decoding Foreman video with CIF format [36]

of the sum of packet time-delays throughout the observing windows during the decoding of the Foreman video sequence with CIF format when adopting the MB remapping technique. It is noticed that applying the MB remapping algorithm affects the time-delay of the packets. In addition, the figure shows the comparison with the case of ordinary decoding. The comparison illustrates that using MB remapping decreases gradually the total packet time-delay. Note that the task moves occur after processing 80, 100, 160, and 270 frames. Fig. 15 shows that the total packet delay decreases after the conducted moves. This refers to the fact that task remapping contributes in distributing the tasks on PEs that are nearer to the memory modules accommodating the input and output FIFOs. Also, Fig. 15 presents a comparison in terms of average time-delay of packets transported during the decoding of the Foreman video with CIF format when adopting the MB remapping technique and when using ordinary decoding. The comparison confirms that the use of MB remapping technique contributes significantly in reducing the time-delay.

3) Timings: Fig. 16 presents the recorded total communication time and total computational time throughout the observing windows during the decoding of the Foreman video with CIF format when adopting the MB remapping technique and when using ordinary decoding. It shows that the communication time represents 90% of the total execution time in both cases. Hence, the total execution time is affected more by the variation of the total communication time. Also, Fig. 16(a) shows that the total communication time is almost not changing among observation windows in the case of ordinary decoding. Whereas, when MB technique is applied, the communication time varies significantly and tends to follow a decreasing manner as shown in Fig. 16(b). This illustrates that reducing the time-delay achieved by MB remapping has a direct impact on the communication time.

The communication time of each processing element is investigated through the decoding of all video frames. It is noticed that when applying the MB remapping technique, the variation between communication times of all involved PEs is reduced. The communication time values of all PEs converges gradually to a specific interval as shown in Fig. 17.

4) Performance results: Multiple simulations have been conducted to decode several benchmark video sequences from [36]. Fig. 18(a) presents the achieved throughput in terms of frames per second (FPS) when decoding Foreman video (CIF format) and using ALTD and APTD respectively for estimating the NoC communication time delay. The figure also shows the achieved throughput when decoding the Foreman video (CIF format) without remapping. The letter "M" shown on the curves represents when an actor move occurs. Fig. 18(a) shows that using MB results in significant performance enhancement. In addition, the figure illustrates that adopting ALTD for estimating the NoC communication time delay, while decoding Foreman video sequence with CIF format, increases the achieved enhancement ratio. Other similar simulations have been conducted targeting other video sequences with different formats (CIF, 4CIF and QCIF). The selected videos are of diverse characteristics to ensure that the proposed remapping algorithm is not related to specific formats or video content. The obtained results confirm that adopting MB algorithm ensures enhanced performance when compared to decoding the video without remapping. Also, the results demonstrate that adopting ALTD rather than APTD leads to additional performance enhancement.

D. Discussion and Comparison

In order to determine the relevancy of the devised algorithm, it is compared to the STM method introduced in [31]. To achieve fair comparison, the STM method has been modeled and implemented on our devised NoC-based architecture. We have also implemented the exact method presented in [20] for the initial mapping, with two differences: we have used constraint programming instead of ILP, and the objective function is the maximum period, Eqn. 12, as it is our optimization goal. The workload used for the computation time of the actors is based on the profiling of Foreman video. Simulations have been conducted while running the MPEG4 decoder to process real-life videos.

1) Performance enhancement of MB remapping: The results presented in Fig. 18 show that for Foreman video sequence with CIF format (Fig. 18(a)), the use of MB remapping algorithm when adopting ALTD leads to a maximum performance enhancement of 38.2% (frame 280) and adopting MB-APTD leads to a maximum performance enhancement of 14.8% (frame 210) when compared to the results of processing the video without remapping. For Ice video sequence with 4CIF format (Fig. 18(b)), maximum performance enhancement of 56% (frame 450) and 16.5% (frame 250) are recorded when applying the MB algorithm adopting ALTD and APTD respectively. Furthermore, the use of MB algorithm adopting ALTD and APTD leads to a maximum performance enhancement of 10.92% (frame 120) and 7.6% (frame 50) for Bus video sequence with QCIF format (Fig. 18(c)) and Bus video with CIF format (Fig. 18(d)). For Grandma video with QCIF format (Fig. 18(e)), maximum performance enhancement of 33.2% (frame 170) and 23.9% (frame 190) are recorded when applying the MB algorithm adopting ALTD and APTD respectively. The simulation results show that the link level

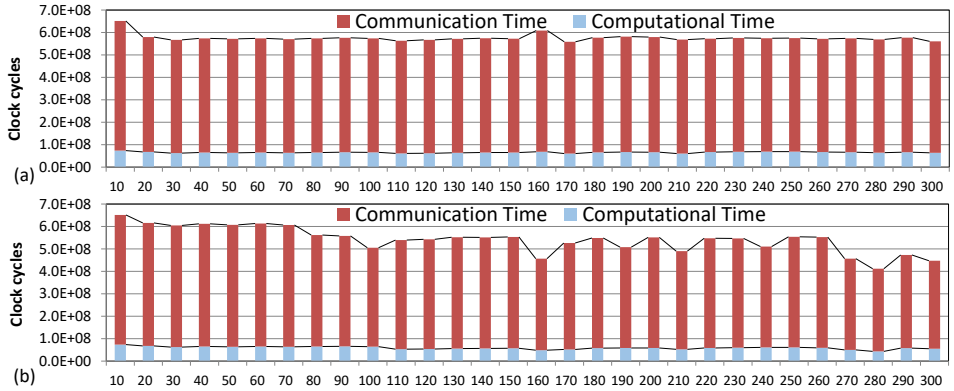


Fig. 16. Total communication and computational times recorded throughout the observing windows during the decoding of the Foreman video with CIF format [36]; when adopting (a) ordinary decoding and (b) MB remapping

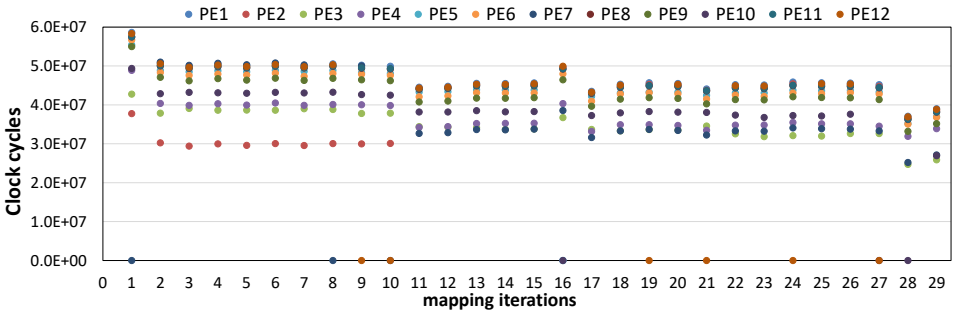


Fig. 17. PE communication time in terms of FPS of decoding Foreman video with CIF format [36] using MB remapping algorithm

estimation of ALTD is more accurate and usually leads to better performance compared to APTD. However, in some cases APTD performs better such as for some observation windows of Grandma video (Fig. 18(e)). This refers to the fact that the heuristic is data-dependent and the link level prediction depends on the monitoring information collected during the previous data which may not match with the that of the current processed data.

2) MB remapping in comparison to STM remapping: Fig. 18 shows a comparison between our proposed remapping and the STM algorithm in terms of throughput (FPS). The results shows that the MB remapping outperforms STM remapping technique when considering either APTD or ALTD for estimating the NoC communication time delay.

Besides, the graphs in Fig. 18 show that in some cases the STM method leads to deterioration in the performance. In fact, the STM method selects critical task to be moved in each observation window without estimating the resulting total performance gain. Moving the task without determining its effects on the whole system performance degrades the overall performance. While in our proposed algorithm, the maximum achieved total gain among all mapping combinations is first determined as explained in subsection IV-E2c. Accordingly, a task is specified to be moved if the estimated maximum total gain is positive. It is noticed that in some observation windows no tasks are moved when the proposed algorithm is applied. A move is indicated by letter "M" in Fig. 18(a). In these cases, the estimation shows that no performance enhancement will be achieved for all mapping combinations.

3) MB remapping in comparison to optimal mapping: Fig. 18 also shows the results obtained from the mapping approach proposed in [20]. Note that the "optimal" mapping corresponds to the best mapping found based on the profiling of Foreman video after a time out of one hour (like the original paper), and the optimality is not proven. The results show that the MB algorithm, starting from a random mapping (without significant initial delay), performs better that the optimal with no remapping for Foreman video sequence in CIF format (Fig. 18(a)). As the optimality is searched for the Foreman profile, we used the optimal mapping as a starting point for the MB algorithm, and the results show that it further improves the throughput. As expected, the optimal mapping for Foreman does not perform good for the Ice video sequence in 4CIF format (Fig. 18(b)) and Grandma video sequence in QCIF format (Fig. 18(e)). But surprisingly, it performs good for the Bus video in QCIF format (Fig. 18(c)) and Bus video in CIF format (Fig. 18(d)). The so-called optimal method cannot be used for two reasons. First it introduces an unpractical initialization delay without guaranty of optimality. Secondly, a static solution is not appropriate to data-dependent applications since a solution can be good for one data-stream and inefficient for another one and more importantly the efficiency of a mapping varies over time.

4) Comparison summary: Table VI summarizes the comparison of average FPS achieved when processing multitude video sequencing while adopting different remapping tech-

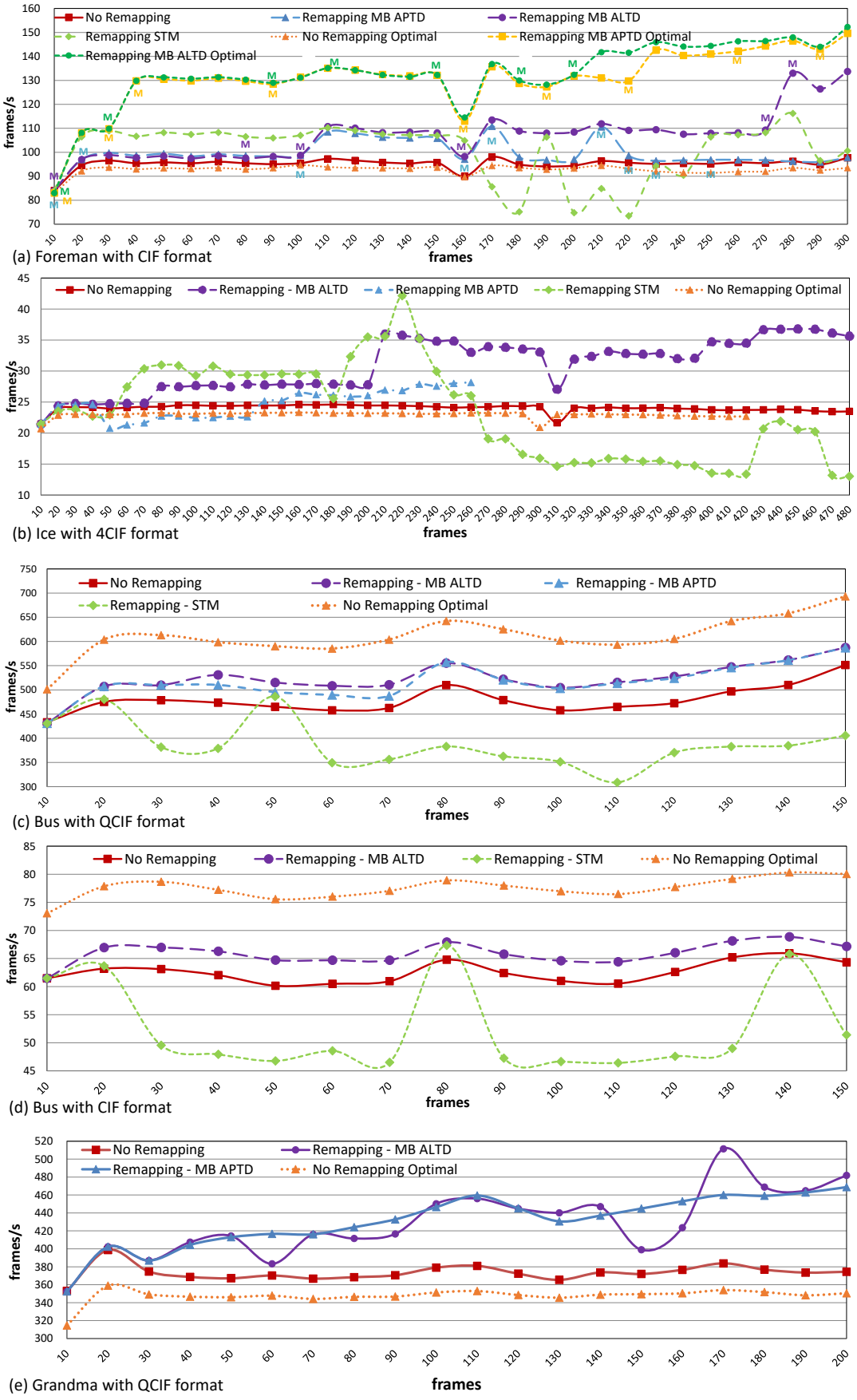


Fig. 18. Throughput in terms of FPS when decoding video sequences [36] using MB and STM remapping algorithms

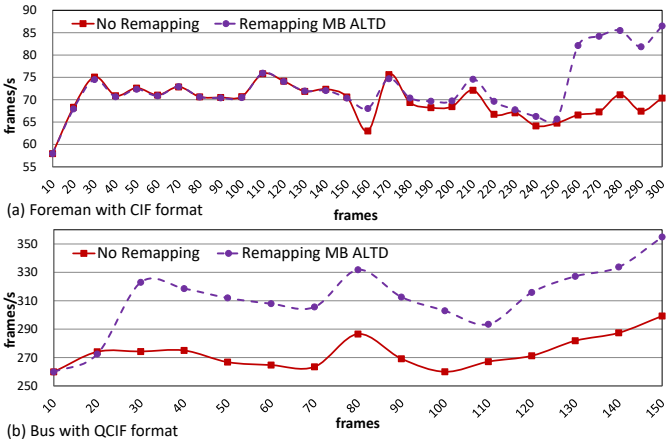


Fig. 19. Throughput in terms of FPS when decoding video sequences [36] using MB remapping algorithms targeting 4×6 NoC

TABLE VI
ACHIEVED RESULTS ADOPTING DIFFERENT REMAPPING TECHNIQUES

Video		Remapping Algorithm		
Sequence	Format	MB-ALTD	MB-APTD	STM
Foreman	CIF	11.4%	5%	4.1%
Bus	CIF	5.4%	5.4%	−17.7%
Ice	4CIF	26.1%	2%	−13.04%
Bus	QCIF	9%	8%	−20%
Grandma	QCIF	14.91%	14.11%	NA

niques. The table shows that the MB algorithm achieves the maximum average performance enhancements of 26% and 14.11% when adopting ALTD and APTD respectively compared to the achieved throughput of processing the frames without remapping. Whereas, remapping using STM algorithm achieves a maximum average enhancement of 4%.

E. Scalability and generality

The scalability of our approach relies first on a negligible extra payload in the context of actor-level dataflow models, which intrinsically require a large amount of small control packets. For example, when decoding the Foreman video sequence the extra flits imposed by remapping (including the flits holding the binary codes of moved actors) constitute less than 0.02% of the flits used for dataflow. The proposed remapping method enhances the performance by exploiting the NoC structure and the characteristics of the available resources. The results show that our method positively impacts the NoC performance. Table VII illustrates the reduction percentages of packet hops when decoding different video sequences adopting the proposed MB remapping compared to ordinary decoding without remapping. The comparison shows that the proposed remapping method reduces the packet hops. The percentage of reduction is more than 20%. Secondly, the method includes the migration cost and so limits the number of moves.

Fig. 19 shows the results obtained for a 4×6 NoC, for Foreman and Bus video sequences, starting from a random mapping. The results show that our approach can also improve the throughout for a larger NoC. On average, the throughout is improved by 13.5% and 4% for Bus QCIF and Foreman CIF videos respectively.

TABLE VII
REDUCTION OF PACKET HOPS WITH MB-ALTD AND MB-APTD

Video		Remapping Algorithm	
Sequence	Format	MB-ALTD	MB-APTD
Foreman	CIF	20.94%	12.64%
Bus	QCIF	3.24%	5.29%
Grandma	QCIF	14.18%	8.33%

VI. CONCLUSION

This paper presents an original *Move*-based algorithm and NoC-based architecture to map the tasks of dataflow application during run-time. The method monitors the performance and intercommunication, takes the proper mapping decision and applies the required mapping configurations. The algorithm and the devised architecture are thoroughly presented. The best way to verify the effectiveness of a run-time mapping, which is by definition data dependent, is to simultaneously execute the target application. However such demonstrations are complex, time consuming and so ignored in the literature. In this paper we address this issue by conducting a SystemC simulation of the MPEG4-SP decoder with several real-life video sequences. The obtained results demonstrate that the proposed algorithm significantly enhances the performance. In addition, the proposed algorithm outperforms the available run-time mapping technique. Future work will consider the implementation of integrated module in the NIs and estimating the overhead in terms of area and energy.

REFERENCES

[1] W. A. Najjar *et al.*, "Advances in the dataflow computational model," *Parallel Computing*, vol. 25, no. 13, pp. 1907 – 1929, 1999.
[2] K. J. M. Martin *et al.*, "Notifying memories: a case-study on data-flow applications with NoC interfaces implementation," in *Proc. of the Design Automation Conf. (DAC)*, June 2016.
[3] H. Yviquel *et al.*, "Orcc: Multimedia development made easy," in *Proc. of the ACM Int. Conf. on Multimedia*, ser. MM '13. New York, NY, USA: ACM, 2013, pp. 863–866.
[4] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proc. of the IEEE*, vol. 75, no. 9, pp. 1235–1245, Sep. 1987.
[5] Y. Lesparre *et al.*, "Evaluation of synchronous dataflow graph mappings onto distributed memory architectures," in *Proc. of Euromicro Conf. on Digital System Design (DSD)*, Aug. 2016, pp. 146–153.
[6] E. A. Lee and T. Parks, "Dataflow process networks," in *Proc. of the IEEE*, 1995, pp. 773–799.
[7] C. Wang *et al.*, "Dynamic application allocation with resource balancing on NoC based many-core embedded systems," *J. Syst. Archit.*, vol. 79, no. C, pp. 59–72, Sep. 2017. [Online]. Available: https://doi.org/10.1016/j.sysarc.2017.07.004
[8] J. Henkel *et al.*, "Dynamic resource management for heterogeneous many-cores," in *Proc. of the IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD)*, 2018, pp. 1–6.
[9] S. Kaushik *et al.*, "Computation and communication aware run-time mapping for NoC-based MPSoC platforms," in *Proc. of IEEE Int. SOC Conf.*, Taipei, Taiwan, 2011, pp. 185–190.
[10] T. Maqsood *et al.*, "Dynamic task mapping for network-on-chip based systems," *J. of Systems Architecture*, vol. 61, no. 7, pp. 293 – 306, 2015.
[11] H. R. Mendis *et al.*, "Dynamic and static task allocation for hard real-time video stream decoding on nocs," *Leibniz Transactions on Embedded Systems*, vol. 4, no. 2, pp. 01:1–01:25, Jul. 2017.
[12] M. Rapp *et al.*, "Neural network-based performance prediction for task migration on S-NUCA many-cores," *IEEE Transactions on Computers*, pp. 1–1, 2020.
[13] S. Paul *et al.*, "Adaptive task allocation and scheduling on NoC-based multicore platforms with multitasking processors," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 1, Dec. 2020.

[14] ——, "A hybrid adaptive strategy for task allocation and scheduling for multi-applications on NoC-based multicore systems with resource sharing," in *Proc. of Design, Automation Test in Europe Conf. Exhibition (DATE)*, 2021, pp. 1663–1666.
[15] Z. Li *et al.*, "Chordmap: Automated mapping of streaming applications onto CGRA," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2021.
[16] D. Huff *et al.*, "Clockwork: Resource-efficient static scheduling for multi-rate image processing applications on FPGAs," in *Proc. of IEEE Annual Int. Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021, pp. 186–194.
[17] T. D. Ngo *et al.*, "Move based algorithm for runtime mapping of dataflow actors on heterogeneous MPSoCs," *J. of Signal Processing Systems*, vol. 87, no. 1, pp. 63–80, Apr. 2017.
[18] A. Singh *et al.*, "Mapping on multi/many-core systems: Survey of current and emerging trends," in *Proc. of the Design Automation Conf. (DAC)*, May 2013, pp. 1–10.
[19] P. K. Sahu and S. Chattopadhyay, "A survey on application mapping strategies for Network-on-Chip design," *J. of Systems Architecture*, vol. 59, no. 1, pp. 60–76, 2013.
[20] K. Huang *et al.*, "A scalable and adaptable ILP-Based approach for task mapping on MPSoC considering load balance and communication optimization," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 9, pp. 1744–1757, Sep. 2019.
[21] C. Rubattu *et al.*, "Pathtracing: Raising the level of understanding of processing latency in heterogeneous mpsocs," in *Drone Systems Engineering and Rapid Simulation and Performance Evaluation: Methods and Tools*, ser. DroneSE and RAPIDO '21. NY, USA: ACM, 2021, pp. 46–50. [Online]. Available: https://doi.org/10.1145/3444950.3447282
[22] C. Chou and R. Marculescu, "Run-time task allocation considering user behavior in embedded multiprocessor networks-on-chip," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 1, pp. 78–91, 2010.
[23] E. L. d. S. Carvalho *et al.*, "Dynamic task mapping for MPSoCs," *IEEE Design Test of Computers*, vol. 27, no. 5, pp. 26–35, 2010.
[24] M. Fattah *et al.*, "Adjustable contiguity of run-time task allocation in networked many-core systems," in *Proc. of Asia and South Pacific Design Automation Conf. (ASP-DAC)*, 2014, pp. 349–354.
[25] A. K. Singh *et al.*, "Resource and throughput aware execution trace analysis for efficient run-time mapping on MPSoCs," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 1, pp. 72–85, 2016.
[26] C. Ykman-Couvreur *et al.*, "Linking run-time resource management of embedded multi-core platforms with automated design-time exploration," *IET Computers & Digital Techniques*, vol. 5, pp. 123–135(12), Mar. 2011.
[27] W. Quan and A. D. Pimentel, "A scenario-based run-time task mapping algorithm for MPSoCs," in *Proc. of the Annual Design Automation Conf. (DAC)*, New York, NY, USA, 2013.
[28] A. K. Singh *et al.*, "Accelerating throughput-aware runtime mapping for heterogeneous MPSoCs," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 18, no. 1, Jan. 2013.
[29] S. Skalistis and A. Simalatsar, "Near-optimal deployment of dataflow applications on many-core platforms with real-time guarantees," in *Proc. of the IEEE Design, Automation Test in Europe Conf. Exhibition (DATE)*, 2017, pp. 752–757.
[30] H. Yviquel *et al.*, "Towards run-time actor mapping of dynamic dataflow programs onto multi-core platforms," in *Proc. of the Int. Symposium on Image and Signal Processing and Analysis (ISPA)*, 2013, pp. 732–737.
[31] W. Quan and A. D. Pimentel, "A hybrid task mapping algorithm for heterogeneous MPSoCs," *ACM Trans. on Embedded Computing Systems (TECS)*, vol. 14, no. 1, pp. 14:1–14:25, Jan. 2015.
[32] J.-P. Diguet *et al.*, "Networked power-gated MRAMs for memory-based computing," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 12, Dec. 2018.
[33] S. S. Bhattacharyya *et al.*, "Overview of the MPEG reconfigurable video coding framework," *Journal of Signal Processing Systems*, vol. 63, no. 2, Dec. 2011.
[34] M. Wipliez and M. Raulet, "Classification of dataflow actors with satisfiability and abstract interpretation," *Int. J. of Embedded and Real-Time Communication Systems (IJERTCS)*, vol. 3, no. 1, pp. 49–69, 2012.
[35] T. Grotker *et al.*, *System Design with SystemC*. Springer, 2002.
[36] Xiph.org video test media. [Online]. Available: http://media.xiph.org/video/derf/

Mostafa Rizk received his Maitrise degree in Electronics, M.Sc in Biomedical Physics, and M.Sc in Signal, Telecom, Image, and Speech from the Lebanese University in 2007, 2008 and 2010 respectively. He received his Ph.D. degree in Sciences and Technologies of Information and Communication from Telecom Bretagne, France in 2014 and a Ph.D degree in Electronics and Telecommunication from the Lebanese University in 2015. Dr. Rizk has been a post-doctoral researcher at UBS University and Lab-STICC laboratory CNRS, France. Dr. Rizk has been an associate professor at LIU, Lebanon and associate researcher at IMT-Atlantique, France. Currently, Dr. Rizk is a researcher at Lab-STICC laboratory CNRS, Brest, France. His general research interests include both algorithm development and corresponding hardware/software implementations and digital circuit design; NoC design and new MPSoC architectures based on emerging non-volatile memory technologies; embedded machine learning; embedded intelligence and embedded computer vision.

Kevin J. M. Martin received a M.S. degree in electrical and computer engineering in 2004 and a PhD in computer science in 2010 from the Université de Rennes, France. He is since 2011 an associate professor at Université Bretagne-Sud in Lorient, France, in the Lab-STICC. His research interests stand at the crossing point between architecture, methods and tools, including but not limited to: custom processors, CGRA, multi-processor platforms, high-level synthesis, computer-aided design tools, compilers and software engineering.

Jean-Philippe Diguet is a CNRS director of research. He has been A/Prof. at UBS University, research visitor at IMEC/Belgium and UQ/Australia, invited Prof. at Tohoku Univ./Japan and USP/Brasil. At Lab-STICC he has led the team method and tools for SoC and embedded system (MOCS) from 2008 to 2016 and then the ICT and Drones program. Since 2021 his is the director of CROSSING, an International CNRS Lab in Adelaide, Australia dedicated to Human/Automous-Agents teaming. His research work focused initially on various aspects of SoC and embedded system design including self-adaptation and now addresses different levels of embedded and distributed intelligence.