Programmable routers for efficient mapping of applications onto NoC-based MPSoCs

Manel Djemal, François Pêcheux, Dumitru Potop-Butucaru, Robert de Simone, Franck Wajsburt, Zhen Zhang

Programmable routers for efficient mapping of applications onto NoC-based MPSoCs

Manel Djemal
INRIA, France
Email: manel.djemal@inria.fr

François Pêcheux
UPMC-Lip6, Paris, France
Email: francois.pecheux@lip6.fr

Dumitru Potop-Butucaru
INRIA, France
Email: dumitru.potop@inria.fr

Robert de Simone
INRIA, France
Email: rs@sophia.inria.fr

Franck Wajsburt
UPMC-Lip6, Paris, France
Email: franck.wajsburt@lip6.fr

Zhen Zhang
UPMC-Lip6, Paris, France
Email: zhen.zhang@lip6.fr

***Abstract*—We extend the state-of-the-art DSPIN network-on-chip architecture by defining programmable NoC routers that can establish effective static scheduling and routing of data packets as demanded by the application. Router programs are the result of a general compilation process which targets the NoC and the computing cores altogether. The objective is to reduce NoC contentions, improving speed and timing predictability. We consider the range of applications of such an approach and provide results on two of them (a simple embedded controller and an FFT).**

I. Introduction

Modern computer architectures are increasingly relying on multi-processor systems-on-chip (MPSoCs), with data transfers between cores and memories managed by on-chip networks (NoC). This reflects in part a convergence between embedded, general-purpose PC, and high-performance computing (HPC) architecture designs.

Efficient compilation of applications onto MPSoCs remains largely an open problem, with the issue of best mapping of computation parts (threads, tasks,...) onto processing resources amply recognized, while the issue of best use of the interconnect NoC to route and transfer data still less commonly tackled. In the most general case, dynamic allocation of applications and channel virtualization can be guided by user-provided information under various forms, as in OpenMP, CUDA, OpenCL and so on. But then there is no clear guarantee of optimality, and first attempts by non-experts often show poor performances in the use of available computing power. Conversely there are consistent efforts, in the domains of embedded and HPC computing, aiming at automatic parallelization, compile-time mapping and scheduling optimization. They rely on the fact that applications are often known in advance, and deployed without disturbance from foreign applications, and without uncontrolled dynamic creation of tasks. Our proposed work makes most sense in this "static application mapping" case.

An optimal use of the NoC bandwidth should authorize data transfers to be realized according to (virtual) channels that are temporarily patterned to route data "just-in-time". Previous works have identified the need for Quality of Service (QoS) in "some" data connections across the network (therefore borrowing notions from macroscopic networks, say internet and its protocols). But our main claim here is that NoC optimal usage should result from a *global* optimization principle, as opposed to a collection of local optimizations of individual connections. Indeed, various data flows with distinct sources and targets will nevertheless be highly concerted, both in time and space, like in a classical pipelined CPU, where the use of registers (replaced in our case with a complex NoC) is strongly synchronized with that of the functional units.

The purpose of the current paper is to investigate how the underlying architecture should offer the proper infrastructures to implement optimal computation and communication mappings and schedules.[1] Our thesis is that optimal data transfer patterns should be encoded using simple programs configuring the router nodes (each router being then programmed to act its part in the global concerted communication scheme).

We concretely support our proposed approach by extending the DSPIN 2D mesh network-on-chip (NoC) [1] developed at UPMC-LIP6. In this NoC, we replace the fair arbitration modules of the NoC routers with static, micro-programmable modules that can enforce a given packet routing sequence, as specified by small programs. We advocate the desired level of expressiveness/complexity for such simple configuration programs.

We justify our choice by its use in reducing communication time (and therefore global execution time), and in reducing contentions in two case studies: A simple embedded control application and an implementation of the Fast Fourier Transform (FFT). These two examples provide a good illustration of how abstract dataflow communications between compute operations have to be organized according to crossroad traffics at routers, once computations have been mapped to processing elements.

a) Outline: Section II reviews related work. Section III presents the generalist NoC-based MPSoC architecture we used as base for our work. Section IV explains why static packet orders can improve NoC utilization, and how to enforce them. Section V gives our results on the larger FFT example. Section VI concludes.

II. Related work

Our work has drawn significant influence, and is close in intent to three lines of existing work:

- The design of generalist MPSoCs, like the DSPIN-based ones [1], allowing simple programming using classical concepts and tools.
- MIT's RAW architecture [2], which significantly develops NoC router programmability as part of so-called *scalar operand networks*.
- The design of networks-on-chips with resource reservation or QoS mechanisms, of which a good review is provided in Stefan *et al.* [3], Harrand and Durand [4] or Kakoee *et al.* [9].

In RAW, the objective is to allow the MPSoC-wide use of compilation techniques that exploit Instruction Level Parallelism [5] and a very

[1] For space reasons, our results on the synthesis of such schedules are only briefly mentioned here.

fine grain, very efficient scheduling of computations and communications. The main difference in our case is that we aim for a coarser level of control in both the NoC hardware (transmission of packets instead of mere scalar values), and the software control of the NoC (which is performed through components such as cache controllers and DMA units). While losing in NoC routing flexibility and timing precision, our approach allows the use of a classical shared memory programming model, general-purpose development tools, and existing applications. It also reduces the complexity of NoC programs, but makes timing analysis more difficult.

More generally, our intent of allowing NoC resource reservations to improve temporal (or other) properties parallels that of existing work on a variety of NoC architectures. Closest to our work are NoC architectures where reservations are based on time division multiplexing (TDM) [6], [7], [8]. In such architectures, each router performs data transfers according to its own TDM table, which can be seen as a simple program assigning transmission slots of given position in time to the various transmission sources. The main difference with respect to our work is that in all these architectures the objective of slot allocation is to attain bandwidth and latency objectives, which can usually be done with small TDM tables. Our objective is to strongly synchronize computations and communications, allowing, for instance, to start a communication as soon as the computation of the data is finished, and allocating all bandwidth to this transmission (as opposed to using just a percentage of the bandwidth) for a fixed time duration. Doing this at the application scale makes for long communication patterns which require efficient encoding with counters, resulting in more elaborate programs, as discussed in Section IV. Another difference here is that we follow a packet-level arbitration policy, as opposed to the TDM approach which may split larger packets into flits at TDM slot barriers. In our case, packets synchronize both computations and NoC data transfers and arbitration.

We also mention here the development of programmable network switches (but at a level equivalent to an MPSoC tile) [10], and the work on reconfigurable data paths, one example of which is present *inside* the tiles of the P2012 platform [11].

From a different point of view, our work can be seen as a contribution towards the development of the Precision Timed machines [12] and their programming.

Finally, our work can be seen as an exploration of what is needed for efficiently implementing high-performance computing algorithms such as the FFT. In this sense, work on FFTW [13] and Spiral [14] were a basis for our work.

III. Tiled MPSoC architectures in SoCLib

Our work in this paper is based on the tiled MPSoC architecture built upon the SoCLib hardware library [15]. As pictured in Fig. 1, such an MPSoC is composed of a rectangular set of tiles connected through a state-of-the-art proprietary 2D mesh network-on-chip (NoC), called DSPIN [1]. Each tile has its own local interconnect. The local interconnect of a tile is linked to the DSPIN NoC through a Network Interface Controller (NIC). The other IPs of the tile are linked to the local interconnect. Typically useful IPs provided by SoCLib library are CPU cores (MIPS32, PPC405, ARM7 or ARM9 with the associated instruction and data caches), memory banks (RAM or ROM), I/O units, DMA controllers which allow for fast data transfers, timers, interrupt control units, or TTYs (useful for debugging).

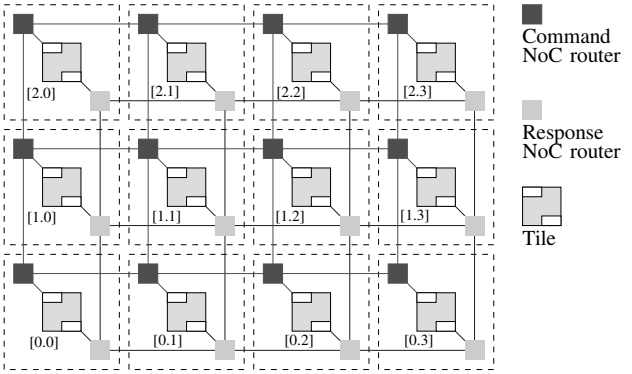

Fig. 1. A typical DSPIN NoC based MPSoC architecture. Dark rectangles are the routers of the command sub-network. Light rectangles are the routers of the response network.

A. Distributed shared memory

The MPSoC follows a *distributed shared memory* model where all memory banks are mapped to a single address space. To this shared memory space are also mapped the programming registers of all peripheral devices, allowing a uniform programming paradigm and the use of general-purpose development tools (the GNU compiler suite, in our case). All inter-tile communication in the MPSoC is performed through memory accesses produced by the CPUs or the CPU-programmed DMA controllers. Inside each tile, specific components allow interrupt triggering in the CPUs, but the programming of the interrupt generators is also done through memory accesses.

The implementation of the distributed shared memory is based on the VCI/OCP protocol [16]. This protocol follows a master/slave model. Masters are called VCI initiators, and slaves are called VCI targets. A CPU is a typical master, while a RAM is a typical slave. Some components, such as DMAs, can act both as VCI initiators (for the transfers themselves) and VCI targets (for the transfer configuration by the CPU).

The VCI/OCP protocol organizes communications into *transactions*. A transaction takes place between a VCI initiator and a VCI target, which exchange *packets* through one or more layers of interconnect, according to the locations of the initiator and target. A transaction consists of a command packet issued by the VCI initiator, followed by the corresponding response packet emitted by the VCI target. The commands and the responses are transmitted through distinct input and output ports and through distinct physical networks (the command subnetwork and the response subnetwork). This helps avoiding deadlock conditions between commands and responses and allows transaction pipelining. This also adds complexity to our objective of programming the NoC, a problem we explicitly address in Section IV-C.

For instance, a CPU triggering a direct memory access (DMA) transfer involves 3 transactions. In the first one, the CPU issues a command to the DMA by sending a command packet to write its memory-mapped registers. The DMA unit acknowledges the command through a response packet. In the second transaction the DMA sends a read command to the source RAM bank. The response of the RAM consists of the required data. Finally, in the last transaction the DMA unit sends a write request towards the target RAM bank, along with the data to write. It receives an acknowledge.

B. The DSPIN NoC

DSPIN is a typical packet-switching network using a wormhole [17] routing paradigm. A DSPIN packet is a sequence of *flits*[2]. As pictured in Fig. 2, the first flit of a packet contains the destination tile coordinates (X and Y), and the last one the end-of-packet (EOP) flag. The default routing algorithm implemented in the command subnetwork is X-First. With this routing algorithm, the command packets are first routed on the X direction, and then on the Y direction. The response packets follow a Y-first route on the response subnetwork, which ensures that DSPIN is deadlock-free.

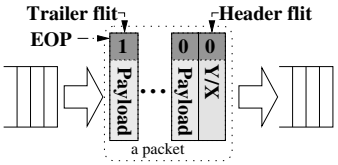

Fig. 2. A general DSPIN packet.

To implement this routing policy, each of the routers pictured in Fig. 1 has the general structure of Fig. 3. The router is composed of five identical modules, named North, South, East, West and Local. The first four correspond to the connections with the adjacent routers. The last one is the link with the local tile. Each module is formed of one input port and one output port. The input port, marked with RF (for *routing function*) in Fig. 3, receives incoming packets, decodes the X and Y destination coordinates, and routes the packet to the corresponding output port of the router. The output port arbitrates between outgoing packets using a Round Robin policy which ensures that no starvation can occur. Every two neighboring routers are directly connected by two FIFOs, providing a point-to-point communication channel.

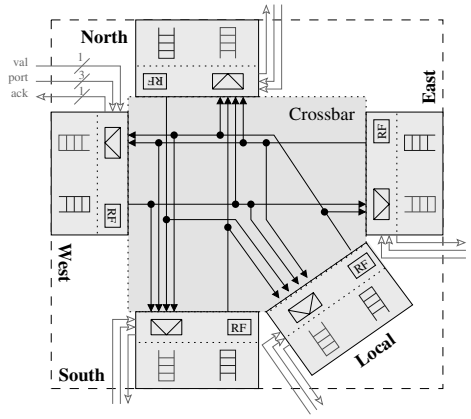

Fig. 3. A DSPIN router. The val, port, and ack wires are only added in the programmable version, presented in Section IV.

C. SystemC Simulation and execution support

Executing code over our DSPIN-based platform requires two distinct executables:

- The hardware simulator itself, compiled and linked with the SystemC and SoCLib libraries.
- The software that will run on the MPSoC platform, i.e. the multithreaded embedded software.

The component models detailed in the paper are written in SystemC and are cycle-accurate bit-accurate (CABA). The CABA modeling rules allow the (optional) use of the optimized simulation engine SystemCASS [18].

The distributed shared memory model used by the MPSoC platform allows the use of standard compiling tools (a GCC cross-compiler) for the generation of the embedded software.

[2]FLow control unITs, the data unit that can be transmitted over a logical link in one clock cycle.

IV. Static scheduling of NoC packets

A. The principle

The NoC routers use a dynamic, fair policy (Round Robin) for choosing the order of packets leaving the router through the output port in a given direction. Along with a limitation on NoC packet sizes, this ensures that NoC resources are evenly distributed among the data transmissions using them, with good NoC utilization factors and guaranteed (albeit possibly low) transmission throughputs for each transmission.

However, when programming embedded control or consumer applications the objective is usually not to improve NoC usage, but to improve application speed, power consumption, etc. The following example explains to what extent fair routing may slow down communications, and thus the overall application. Fig. 4 pictures the case where the "East" output of a DSPIN router is concurrently traversed by two bursts of data, each formed of n packets of equal length m numbered from 0 to n. Each burst transmits a single piece of data, and processing cannot start at the destination until all data have arrived. In the worst case, the first packets of the two bursts arrive at the router in the same clock cycle, and we assume that the current state of the arbiter leads to "Local" passing first. Then, the fair routing policy of DSPIN results in the interleaved transmission of the "Local" burst and the "West" burst having respectively lengths $(2 * n - 1) * m$ and $2 * n * m$. The passing order can be represented by the $(WL)^n$ regular expression.

Fig. 4. Round Robin communication interleaving: $(WL)^n$.

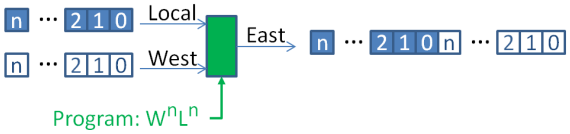

Fig. 5. Programmed communication interleaving: $W^n L^n$.

Our objective is to allow for the better packet interleaving of Fig. 5. We assume there that the "West" burst is needed by a computation located on the critical path of the application running on the NoC, so that accelerating the burst transfer will result in a faster application. Therefore, we let the entire "West" burst pass before the entire "Local" burst, even when "Local" arrives first. The transmission durations will then be respectively $2 * n * m$ and $n * m$. The passing order is represented by the $W^n L^n$ regular expression inside the multiplexer. This expression is an abstract view of the router program specifying that n packets from the "West" input port should pass

before the n packets from the "Local" input port of the NoC router. The router programs to be defined in the next section directly depend on these regular expressions.

As we shall see in the examples of the next sections, such static ordering of packets at router output ports allow speed gains and a balanced use of NoC resources, by precisely allocating free time slots on the NoC to in transit packets. It also allows for the construction of applications with very good timing predictability and enhanced determinism.

However, these gains come at a certain price. Part of this price is the need for *programmable routers*, described in Section IV-B, which increase the silicon surface of the NoC.

The second part is the need for temporal predictability. Indeed, the router programs are computed based on the expected execution order of the various operations (computations and communications). In turn, the order depends on operation durations, and better precision in computing these durations results in better routing programs. To improve the precision in computing operation durations, we build our MPSoC architecture in such a way as to reduces resource access contentions, such as NoC contentions, RAM access contentions, etc. These architectural choices, along with guidelines on how to better (re-)organize software to take advantage of them are presented in Section IV-C.

B. Programmable DSPIN

The final purpose of DSPIN router programming is to fully control the arbitration between incoming input packets at each of the router's output ports. To do this, we introduce new signals that control each router output, as shown in Fig. 3.

The 3 new signals, named VAL, PORT and ACK, are used to transmit to the router output the sequence of routing orders, using a classical FIFO protocol:

- PORT defines the input port from which the next packet will be accepted for transmission. It is set while the current packet is still transmitted, but does not affect this transmission (we do not allow packet transmission interruption).
- VAL validates the new PORT.
- ACK is used to acknowledge the current PORT and VAL. It is set when the first flit of the corresponding packet passes.

These signals are driven by the router controllers. One router controller is added to each of the outputs of a tile, as shown in Fig. 6. We therefore use 5 independent micro-programmable router controllers per tile, grouped together into a LocalRouterController component connected to the local interconnect of the tile to allow programming.

Each controller contains:

- 8 16-bit local registers named R0 to R7, which allow a compact encoding of regular expressions through the use of counters.
- 240 32-bit word local memory for micro-programs. This memory takes the largest place, and we keep its size low.
- 2 addressed registers, CMD and PC: CMD allows for commuting between the programmed and fair (RoundRobin) arbitration policies. PC is the Program Counter.

The controllers have 6 micro-instructions, whose assembly language representations are presented in Table I.

This specific architecture and instruction set allows an efficient (compact) encoding of routing patterns such as the one of Fig. 5 through the use of counters. A full example of router program for a simple application is provided in Section IV-D and Fig. 10. The use of multiple registers and general decrement and test statements

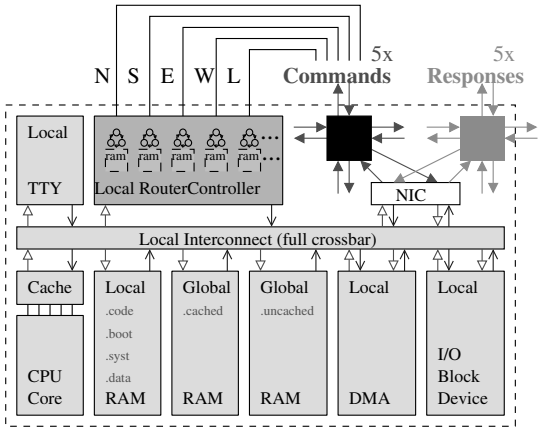

Fig. 6. Tile structure in our architecture. Local router controllers are only present in the programmable version.

Instruction	Function
NOP	Spend a cycle doing nothing.
LOADIMM REG IMM	Load a 16-bit immediate value (IMM) to register REG
WRITE PORT	Send a new arbitration value (through PORT and VAL) to the corresponding router output. The controller is blocked until reception of ACK
DEC REG	Decrement the register
BNZ REG LABEL	If the value of REG is not zero, jump to the target instruction marked by the LABEL
JUMP LABEL	Jumps to the target instruction marked by LABEL

TABLE I
THE ROUTER CONTROLLER MICRO-INSTRUCTION SET.

allows a simple encoding of complex loop nests. If the registers are made addressable in the global address space, our architecture also allows the inclusion of data-dependent control in the routers.

At the same time, router logic remains low. Even if we carried out no exact evaluation of the area overhead, we determined that the largest penalty comes from the 240-word micro-program memory. One remark here is that careful encoding of the instruction set may lead to using much less than one word per statement, on average. A second, more important remark here is that router program memory should be accounted for as program memory, and considered in view of the application efficiency (speed, power) optimizations it enables.

C. Other architectural choices

To reduce the silicon cost, we only add programmable output ports and their respective controllers on the command network part of the NoC, leaving unchanged the fair arbiters on the response network. To avoid uncontrolled contentions on the response network, all large transfers of data, represented by packet bursts, should be performed with write operations. This way, the response network only transfers 2-flit acknowledge packets with negligible contention cost attached.

To minimize NoC usage during *large* data transfers, we perform them using DMA units controlled by the CPU of the sending tile. Transferring data directly through CPU operations would mean that the packet construction and sending is controlled by the CPU cache, which generates one packet for each transmitted word. The use of DMAs also minimizes the transmission interval between successive

packets, as the transmission control is performed in hardware. Using a separate DMA for data transfers also allows for some concurrency between computations and data transfers.

Programming the NoC removes the NoC-related contentions. However, the Network-on-Chip routers are only one of the shared MPSoC resources where contentions can occur. In our experiments, the second most important source of contentions is the access to RAM banks,[3] especially when sustained DMA-driven data transfers are both reading and writing the RAM bank. As a partial solution to this problem, we decided to instantiate two RAM banks in each tile, and use them so as to avoid programming situations where concurrent read and/or write operations are performed on one single bank. More general and scalable solutions, such as programmed access to the RAM are under evaluation.

To avoid interference between code and local data reading by the CPU and the data transfers to and from other tiles, a third memory bank, accessible only within the tile, is used on each tile. This memory bank stores program code, system libraries, CPU exception handling code, as well as local program data.

To exploit the RAM structure, all data and code are explicitly placed on specific memory banks, which allows optimizations relying on data locality.

Also as an attempt to improve temporal predictability in our first analyses, our architecture does not use Inter Processor Interrupts (IPI) for synchronization. All synchronizations are currently managed by locks and active wait mechanisms. The timing penalty is negligible, because the potential gain in signaling speed is lost in subsequent instruction cache update (to read exception code).

Finally, we consider that the local interconnect of each tile is a full crossbar, so that no access conflicts arise in the interconnect. This results in a variable geometry tile structure, where certain tile components are optional, depending on the XY location of the tile in the MPSoC. For instance, I/O devices are only instantiated in a few peripheral tiles.

D. A simple example, in depth

To showcase the previous developments, we consider the dataflow program of Fig. 7. This dataflow model is a highly simplified and pipelined version of the image processing part of the platooning application for the CyCab electric car [19]. Its 3 dataflow operators (f, g, and h) respectively correspond to simple filters (image crop, Sobel filter for detecting edges, and then a histogram search for detecting dominant edges, which are then used to identify the position of the front car). As the front car moves, the crop window changes its position, so that the target car does not exit the processed area. This makes for the feedback loop, which we cut by 2 unit delay blocks, labeled Δ.

Unit delays are 1-place FIFOs that are initially full. The program represents a cyclic behavior. At each cycle, an input image (named i) is read from the exterior before f performs its computation and produces the data x. It also produces an output image (named o) with the cropped area highlighted, which is displayed to the user. Operation g can be executed at the beginning of the cycle, because it depends on the value of x and v of the previous cycle, as transmitted by two delay blocks. Once g produces z, h can be executed. It computes the correction of the crop window (v) as well as some display data (o'). Then, the next execution cycle can start.

We implement this program on the MPSoC of size 2x2 pictured in Fig. 8. To simplify, we assume that f, g, and h have the same

[3]with a single VCI target port, a RAM bank does not allow read and write operation to be performed at a time.

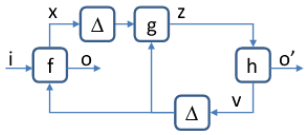

Fig. 7. Simple dataflow specification

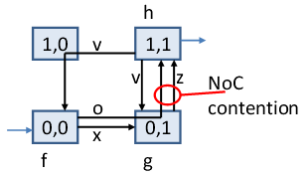

Fig. 8. Sample mapping for the simple dataflow example

duration (1000 clock cycles on the CPU). We also assume that the communication of any of x, y, or z will consume network packets whose combined length is 500 flits, and that the communication of v requires only one packet whose length is 10 flits. Assuming that the tiles and the NoC have the same clock speed (fully synchronous), transmitting x, y, or z without contentions will consume only a little more than 500 cycles (with small variations due to instruction cache state and remaining RAM access conflicts). The timing assumptions of this example are simplified for clarity, but realistic. If we use MIPS32 CPU cores with the standard caches, data transmission through the NoC is roughly 4 times faster than the computation of a simple filter, such as multiplying all data by a constant. At the same time, to balance the NoC speed, tiles usually contain more than one CPU core (e.g. 16, in Kalray's architecture).

We also assume that the data inputs i enter the MPSoC through tile (0,0) and the data outputs o and o' exit the MPSoC through tile (1,1). For this reason, f is executed on cluster (0,0) and h on cluster (1,1).

Under these conditions, our scheduling tool will map the dataflow operations to the MPSoC tiles (to their CPUs) as pictured in Fig. 8 (f on (0,0), g on (0,1), and h on (1,1)). The (simplified) C code for the CPUs to which operations have been allocated in provided in Fig. 9.

The code is divided in 3 sections, each providing the data and the main function of a tile. Global data can be written by other tiles and must be placed in the global RAM banks of Fig. 6. Private tile data, defined inside the main functions, can be stored in the private (local) RAM bank. Synchronization between tiles (for data access) is realized using lock variables. in our case, locks must allow the representation of blocking reads, but blocking writes are not needed due to the tightly synchronized nature of the example. Computation functions (f, g, h) operate only on local data. Communications are performed separately, which allows for sending larger packets of data (using the DMA units) and should facilitate execution time analysis of the functions. Such locality properties can be either ensured during synthesis of the C code from a higher-level data-flow form, or by transformations of an otherwise general C code.

Each inter-tile communication consists in two phases. The first one is the call to `dma_send`, which transfers the actual data using the DMA unit of the tile. The second one is the update of the distant lock, which is performed directly by the CPU.

As the execution of operations f and g depend on the end of operation h, their starts are tightly synchronized. As the two operations have equal durations, the transmissions of o and z exhibit the

```
// CODE AND DATA FOR TILE (0,0)
// Incoming variables have locks to implement blocking
// reads. Blocking writes are not needed in this example.
SmallDataType v_in_0_0 = v_init ;
bool v_in_0_0_lock = 0 ;

void main_0_0() {
  //Data produced locally does not need to be public.
  LargeDataType o_out ;
  LargeDataType x_out ;
  do {
    //execute f (which also reads i)
    f(v_in_0_0,&o_out,&x_out);
    //send o to (1,1) (and signal it using the lock).
    dma_send(o_out,o_in_1_1) ; o_in_1_1_lock = 1 ;
    //send x
    dma_send(x_out,x_in_0_1) ; x_in_0_1_lock = 1 ;
    //waiting for v, using the associated lock
    while(!v_in_0_0_lock) ; v_in_0_0_lock = 0 ;
  } while(1);
}

// CODE AND DATA FOR TILE (0,1)
LargeDataType x_in_0_1 ; bool x_in_0_1_lock = 1 ;
SmallDataType v_in_0_1 = v_init ;
bool v_in_0_1_lock = 1 ;
// Two copies of x are needed on (0,1) to allow
// pipelining between the writing of x by (0,0)
// and its use by g. This is private data but needs
// persistency.
LargeDataType x_delayed = x_init ;
void main_0_1() {
  LargeDataType z_out ;
  do {
    g(x_delayed,v_in_0_1,&z_out) ;
    dma_send(z_out,z_in_1_1) ; z_in_1_1_lock = 1 ;
    while(!x_in_0_1_lock) ; x_in_0_1_lock = 0 ;
    x_delayed = x_in_0_1 ;
    while(!v_in_0_1_lock) ; v_in_0_1_lock = 0 ;
  } while(1);
}

// CODE AND DATA FOR TILE (1,1)
LargeDataType o_in_1_1; bool o_in_1_1_lock = 0 ;
LargeDataType z_in_1_1; bool z_in_1_1_lock = 0 ;

// Two copies of x are needed on (1,1) to allow
// pipelining between the writing of z by (0,1)
// and its use by h.
LargeDataType z_delayed = z_init ;

void main_1_1() {
  SmallDataType v_out ;
  do{
    while(!z_in_1_1_lock) ; z_in_1_1_lock = 0 ;
    h(z_in_1_1,&v_out) ; //execute h, output o,o'
    while(!o_in_1_1_lock) ; o_in_1_1_lock = 0 ;
    dma_send(v_out,v_in_0_1) ; v_in_0_1_lock = 1 ;
    dma_send(v_out,v_in_0_0) ; v_in_0_0_lock = 1 ;
  } while(1);
}
```

Fig. 9. C code for our simple data-flow application

```
// 11 Packets from LOCAL to NORTH
LOOP:   LOADIMM R1 11
L0:     WRITE LOCAL
        DEC R1
        BNZ R1 L0
// 11 Packets from WEST to NORTH
        LOADIMM R1 11
W0:     WRITE WEST
        DEC R1
        BNZ R1 W0
        JUMP LOOP
```

Fig. 10. Assembly code for the North router of cluster (0,1) of our simple application.

phenomenon described in Section IV-A. In our case, the theoretical slowdown will amount to 450 clock cycles, a value closely matched by simulations.

This slowdown can be eliminated by choosing the order in which packets are sent from the router of tile (0,0) to the north. To determine the needed program, we assume that the maximal packet length for DMA data transmissions is 50 flits. As the transmission of z is on the critical path of the application, and not o, the best throughput is ensured by the routing sequence $(L^{11}W^{11})^*$, which allows all the 11 packets from the Local input (the transmission of z) to pass before all the packets from the West input (the transmission of o). From each direction, the first 10 packets correspond to the `dma_send` call, and the 11^{th} corresponds to the lock update. The sequence of 22 packets is repeated indefinitely, once for every computation cycle of the dataflow program. The exact controller assembly program corresponding to this routing sequence is provided in Fig. 10. This program must be placed on the controller of the North output of the (0,1) tile router. All the other router outputs of the NoC can be left non-programmed (under fair routing) because no contentions must be arbitrated.

Note that, the optimization involved no changes to the C code, which can be executed over a programmed or non-programmed NoC.

V. Case study: the FFT

While the previous example showed the details of our platform, we use a more realistic example to show its interest for larger applications. We chose the Fast Fourier Transform [13], [14] because of its widespread use in signal processing and embedded control and because its characteristics (at least in the variant we considered) raise optimization questions that go beyond the classical speed optimization criterion. More precisely, we started from the FFT algorithms proposed in [20] for execution on MPSoC architectures with 2D mesh NoC interconnect. On the default DSPIN-based MPSoC, this algorithm features a strong domination of computation over communication, and a clear organization into successive computation phases and strongly synchronizing communication phases.

An important question to ask, in this case, concerns the case where only part of a larger NoC is dedicated to the FFT computation, while the rest perform other tasks of the global application. In this case, it may be useful to let NoC traffic not belonging to the FFT algorithm traverse the FFT-dedicated NoC area. The question is whether it is possible to do so *without slowing down the FFT computation*. We shall see that programmable routers make this possible, whereas realistic architectural choices lead to slowdowns of over 30% if the standard (non-programmable) fair routers are used.

A. FFT algorithm description

We work on the first FFT algorithm proposed in [20]. It encodes a 1d FFT. We assume the data size is $N = 2^n$ and the number of tiles in the MPSoC is $M = 2^m$. We also assume data is received by tile (0,0) from outside the FFT-dedicated area, and the result is also output by (0,0). Then, the algorithm is based on a (classical) division of the N-sized data into M segments of size N/M. The segments are distributed during an initial communication stage to the M tiles. On this data, each tile computes a FFT of size 2^{n-m}. Once this computation is complete, a sequence of m "butterfly" steps begins. During each step, each tile exchanges data with exactly another one, and then performs some local computation. In the end, the result computed by each tile is gathered on tile (0,0). The data is always exchanged in segments of size N/M.

We choose, for the scope of this paper, $N = 2^{13}$ and a mesh of M=4x4=16 tiles. We pictured in Fig. 11 the direction of the data exchanges during the 6 communication stages.

B. Measures before programming

As mentioned before, in the standard SocLib MPSoC FFT computation time dominates communication time. The duration of the

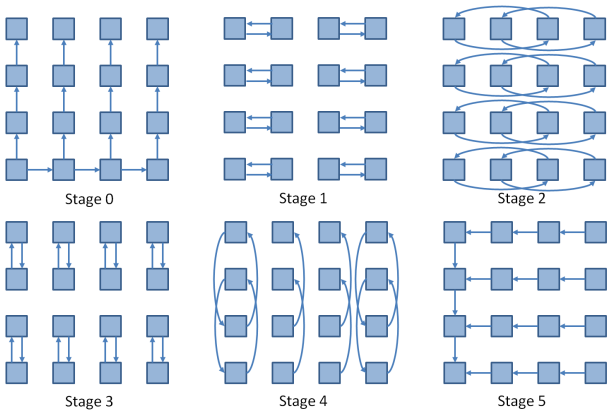

Fig. 11. The 6 FFT communication stages.

initial FFT of size N/M is more than 200 times superior to the duration of the communication of the associated data, and a ratio of 10 between computation and communication is seen in the butterfly stages. Communications are performed in bursts, meaning that the NoC resources remain largely unused, but that significant contentions do occur, in stages 2, 4, and 5 as presented in Fig. 11 in places where communication routes intersect.

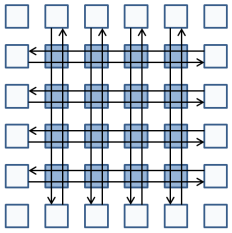

Fig. 12. Adding traffic not due to the FFT itself.

We simulated traffic traversing the FFT-dedicated NoC by using a 6x6 NoC where the border tiles function as traffic generators, and the middle 4x4 tiles execute the FFT algorithm. Traffic is simulated through sustained data transmissions between border tiles, in the fixed directions pictured with arrows in Fig. 12. This form of traffic simulates the worst-case effect of East-West and North-South transfers. We believe the approach is realistic due to the bursty nature of the FFT communication, which means that the same slowing effects, described below, can be obtained with much less traffic (but occurring during peak NoC use by the FFT).

As expected, traffic injection in the absence of reservation by programs does slow down the execution of the FFT. To evaluate the effect of architectural variability, we computed this slowdown for the default SocLib/DSPIN platform, but also on platforms where the tiles clocks are speeded up by factors of 2, 4, 8, and 16 with respect to the NoC clock. Tile speedup seems the best way of modeling the effect of using more powerful tiles with more or faster CPU cores (as in Kalray's or p2012 architectures [4], [11]) or, alternately, slower NoCs (in either clock speed or flit size).

The results, pictured in Table II, show that NoC contentions due to traffic not belonging to the FFT become important as the tiles speed up. Figures are given in NoC clock cycles, to make clear the speedup of the whole MPSoC as the speed of the tiles increases. For instance, a 16x tile speedup results in a 8.3x speedup in the FFT computation. But the same tile speedup only results in a 5.75x speedup if external traffic is injected, which amounts to an impressive 33% slowdown

tile speedup	FFT alone		FFT+traffic		Slowdown
	NoC cycles	speedup	NoC cycles	speedup	
1x	646961	reference	668272	0.96x	3.29%
2x	342684	1.9x	368653	1.75x	7.58%
4x	193737	3.3x	222146	2.91x	14.66%
8x	120557	5.5x	148965	4.34x	23.56%
16x	83978	8.3x	112408	5.75x	33.85%

TABLE II
SLOW-DOWN DUE TO TRAFFIC INJECTION (IN NOC CLOCK CYCLES).

tile speedup factor	Non-programmed	Programmed
1x (no speedup)	0.52%	1.32%
2x	0.98%	2.44%
4x	1.78%	4.26%
8x	3.05%	6.75%
16x	4.74%	9.58%

TABLE III
NETWORK USE BY FFT-GENERATED PACKETS, IN % OF ALL TRANSITING PACKETS

of the FFT computation speed when compared to the case with no traffic injection. Note that this happens while the FFT still uses only a small part of the NoC bandwith (as explained below, in relation with Table III). However, the fact that this use is concentrated in short, highly synchronized bursts means that contentions at those points in time have a very significant effect, and show the importance of reserving not only bandwidth, but bandwidth at specific points in time, as our architecture allows.

C. Adding programming to the NoC

Our objective here was to show that NoC programming allows us to maintain FFT speed while allowing permeability to traffic originating outside the FFT-dedicated MPSoC area. We did achieve this, *i.e.* programming the NoC as explained below maintains FFT speed while allowing external traffic to traverse the FFT-dedicated MPSoC part. As a quantitative measure of permeability, Table III provides figures showing the percentage of FFT-generated packets transiting the NoC over the global number of packets transiting it. We can see that programming reduces permeability, but not significantly. In the worst case, which corresponds to the biggest tile speedup, more than 90% of all transiting packets do not belong to the FFT. In all cases, the programmed NoC allows the FFT to run as if no external traffic existed.

The main difficulty we had was the construction of NoC programs. Work on scheduling tools capable of synthesizing efficient schedules is under way, and we are already able to automatically synthesize the code for specifications that follow a classical data-flow form (like the first example). However, we are not yet able to exploit the regularity properties that make for an efficient FFT implementation. Therefore, our objective here is not the synthesis of schedules from scratch, but merely the preservation of the speed (and therefore of the schedule) of the FFT operations, as for an FFT executed in isolation.

As pictured in Fig. 13, the approach we propose is based on the use of an execution trace of the FFT in isolation. This execution trace includes the routing operations performed by all of NoC's routers, including their starting dates and durations. As we want to preserve these dates and durations unchanged, we see this execution trace as a reservation table showing for each NoC route when it is free from FFT traffic. These time intervals can be used to transfer other packets. Therefore, we can statically schedule a maximal number of

external packets in these intervals. The resulting scheduling table, when projected on each router output, provides the program of the router. Collectively, these programs ensure the preservation of the FFT speed.

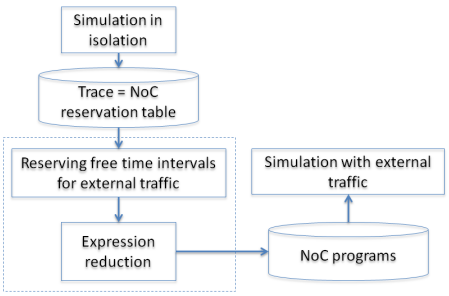

Fig. 13. NoC program generation flow.

We have written a tool that automatically transforms execution traces of the FFT in isolation into programs with maximal statically scheduled communications from outside the FFT-dedicated area. This tool allowed us to verify by simulation that the resulting system preserves the FFT speed. One critical aspect here is the optimization of the generated router programs by identification of repetitive patterns in the orders provided by the previously-defined process. Indeed, a small router program memory is a key point in reducing the area impact of NoC programmability. In the case of the FFT, these programs remain short.

VI. Conclusion

We have advocated the fact that efficient parallel executions on a NoC-based MPSoC require better integration of CPU core computations with NoC data traffics, which could be obtained by compile-time static scheduling of both computations and communications. In turn, this means that global compiling processes should target together the processing elements and the programmable NoC routers. While this may remain an utopia for some time at final user level and general-purpose applications, it is realistic to consider for basic library functions and intensive data-computation functions, regular enough to benefit from high parallel streaming throughput. Typical regular applications as found in scientific and signal-processing parallel computing of course fall into that range. It remains to be seen how compiling directives such as expressed in languages such as OpenMP, CUDA, OpenCL, and so on, could be efficiently used in some cases to help build predictable routing patterns to be uploaded as configurations to our programmable routers.

b) Future work: We are currently working on several extensions of this work: The main effort is dedicated to the automatic synthesis of efficient static schedules for given dataflow specifications. We use direct static (table-based) scheduling, following an approach previously advocated for in the context of real-time scheduling, or in compilation for superscalar and VLIW microprocessors. Another research direction aims at finding a good balance between NoC router complexity and efficiency gain, seen in a broad sense. Solutions here range between an all-hardware solution where the routers incorporate more and more features such as software-defined routes or multi-cast, and a full software solution where software protocols realize all these functions. For instance, we need protocols for interfacing between statically-programmed areas of the NoC and areas that still use fair arbiters.

c) Acknowledgements: The authors would like to thank Alix Munier, Alain Greiner, and Daniela Genius for interesting discussions on this work.

References

[1] I. Panades, "Conception et implantation d'un micro-réseau sur puce avec garante de service," Ph.D. dissertation, Université Pierre et Marie Curie, 2008.

[2] E. W. *et al.*, "Baring it all to software: The raw machine," *IEEE Computer*, vol. 30, no. 9, pp. 86 –93, sep 1997.

[3] R. Stefan, A. Molnos, and K. Goossens, "daelite: A tdm noc supporting qos, multicast, and fast connection set-up," *IEEE Transactions on Computers*, 2012.

[4] M. Harrand and Y. Durand, "Network on chip with quality of service," United States patent application publication US 2011/026400A1, Feb. 2011.

[5] M. B. T. *et al.*, "Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams," *SIGARCH Comput. Archit. News*, vol. 32, no. 2, Mar. 2004.

[6] C. Paukovits and H. Kopetz, "Concepts of switching in the time-triggered network-on-chip," in *Proceedings of the IEEE international conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Kaohsiung, Taiwan, 2008.

[7] M. Schoeberl, F. Brandner, J. Sparso, and E. Kasapaki, "A statically scheduled time-division-multiplexed network-on-chip for real-time systems," in *Proceedings of the 6th International Symposium on Networks-on-Chip*, Lyngby, Denmark, May 2012.

[8] K. Goossens, J. Dielissen, and A. Radulescu, "Æthereal network on chip: Concepts, architectures, and implementations," *IEEE Design & Test of Computers*, vol. 22, no. 5, 2005.

[9] M. Kakoee, V. Bertacco, and L. Benini, "ReliNoC: A reliable network for priority-based on-chip communication," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, march 2011, pp. 1 –6.

[10] H. de Freitas, , and P. Navaux, "On the design of reconfigurable crossbar switch for adaptable on-chip topologies in programmable noc routers," in *Proceedings GLVLSI*, 2009.

[11] L. Benini, "Programming heterogeneous many-core platforms in nanometer technology: the p2012 experience," Presentation in the ARTIST Summer School, Autrans, France, Sep 2010, online at: http://www.artist-embedded.org/artist/Videos.html.

[12] S. A. Edwards and E. A. Lee, "The case for the precision timed (pret) machine," in *Proceedings of the 44th annual conference on Design automation*. SESSION: Wild and crazy ideas (WACI), June 2007.

[13] S. G. Johnson and M. Frigo, "Implementing FFTs in practice," in *Fast Fourier Transforms*, C. S. Burrus, Ed. Rice University, Houston TX: Connexions, September 2008, ch. 11. [Online]. Available: http://cnx.org/content/m16336/

[14] P. Milder, F. Franchetti, J. Hoe, and M. Püschel, "FFT compiler: From math to efficient hardware," in *IEEE International High Level Design Validation and Test Workshop (HLDVT)*, 2007.

[15] LIP6, "SoClib: an open platform for virtual prototyping of multi-processors system on chip," 2011, online at: http://www.soclib.fr.

[16] V. Alliance, "VCI: Virtual Component Interface Standard (OCB 2 2.0)," online at: http://www.vsi.org.

[17] L. Ni and P. McKinley, "A Survey of Wormhole Routing Techniques in Direct Networks," *Computer*, vol. 26, no. 2, pp. 62–76, 1993.

[18] R. Buchmann, F. Pétrot, and A. Greiner, "Fast cycle accurate simulator to simulate event-driven behavior," in *Proceedings of the International Conference on Electrical, Electronic and Computer Engineering*, 2004, pp. 35–38.

[19] C. Pradalier, J. Hermosillo, C. Koike, C. Braillon, P. Bessière, and C. Laugier, "The CyCab: a car-like robot navigating autonomously and safely among pedestrians," *Robotics and Autonomous Systems*, vol. 50, no. 1, 2005.

[20] J. H. Bahn, J. Yang, and N. Bagherzadeh, "Parallel fft algorithms on network-on-chips," in *Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on*, april 2008.