Reconciling performance and predictability on a many-core through off-line mapping

Thomas Carle, Manel Djemal, Daniela Genius, François Pêcheux, Dumitru Potop-Butucaru, Robert De Simone, Franck Wajsbürt, Zhen Zhang

Reconciling performance and predictability on a many-core through off-line mapping

Thomas Carle*, Manel Djemal*, Daniela Genius‡, François Pêcheux‡, Dumitru Potop Butucaru*,
Robert de Simone*, Franck Wajsbürt‡, Zhen Zhang†
*INRIA, Rocquencourt, France, Email: firstname.name@inria.fr
†IRT SystemX Saclay, France, Email: firstname.name@irt-systemx.fr
‡ LIP6, Université Pierre et Marie Curie, Paris, France, Email: firstname.name@lip6.fr

***Abstract*—We start from a general-purpose many-core architecture designed for average-case performance and ease of use. In particular, its distributed shared memory programming model allows the use of a code generation flow based on the (unmodified) gcc compiler chain. We modify this architecture and extend the code generation flow to allow the construction of efficient hard real-time systems starting from dependent task specifications. We rely on a static (off-line) real-time scheduling paradigm well-adapted to embedded control and signal processing applications with regular control structure.**

We modify the architecture (and in particular the on-chip network) to allow the implementation of static schedules with very high (clock cycle) temporal precision. On the software side, we define application mapping rules ensuring that the timing precision provided by the hardware is not lost. These mapping rules include requirements on the allocation of data variables to specific RAM banks and on the use of locks to ensure the absence of contentions during access to shared resources. Applications complying with these rules can be written manually or automatically obtained using a new mapping tool that takes all the allocation and scheduling decisions. Compilation of the resulting C code is still done using the (unmodified) gcc compiler chain. The resulting platform provides good performance, and at the same provides very high timing precision, as shown by two case studies (an embedded controller and an implementation of the FFT).

We conclude our paper with a presentation of some ongoing work on the subject: A case study (an implementation of the H.264 decoder) meant to test the limitations of our method.

I. Introduction

The number of transistors in micro-processor chips upholds today its historic trend of exponential growth, known as Moore's law [1]. Until the years 2000, this growth mostly translated into micro-architectural changes aimed at improving mono-processor performance. However, performance increase by micro-architectural advances alone follows the (empirical) Pollack's rule [2] which states that the performance increase is roughly proportional to the square root of the increase in complexity. As the last decade brought the end of the fast operating frequency increases [3], mono-processors were no longer able to cover the performance needs of increasingly complex applications.

This led to an industry-wide shift towards parallel computing. While parallel architectures already existed, mainly in the high-performance and embedded computing fields, parallelism now entered the mainstream of general-purpose computing under the form of multi-core, and then many-core architectures. We are focusing in this paper on *tiled many-core architectures* characterized by:

- *Large numbers of simpler processing cores*, ranging from a few tens to a few hundreds in production architectures. The cores are divided among a set of identical *computing tiles*.
- Tiles are linked together through one or more *networks-on-chip (NoCs)* with regular structure (*e.g.* mesh, torus). Such NoCs provide better performance/scalability trade-offs than classical buses and crossbars.
- *Novel memory architectures* that can deliver higher bandwidth access through the use of multiple memory banks localized near the processors (in the computing tiles). Data localization often requires that the memory hierarchy is exposed, at least in part, to the programmer.

Tiled many-cores are commercially proposed today for general-purpose, high-performance, and embedded/real-time applications [4], [5], [6]. However, the use of such architectures in a real-time context not only depends on the performance, predictability and cost of the hardware, but also on the availability of development tools such as compilers, scheduling and schedulability analysis tools, WCET analysis tools, *etc.*

a) Contribution: To reconcile performance and predictability, all the hardware, software, and mapping aspects of our design flow are considered in a unified way and optimized to fit a single real-time scheduling paradigm: *table-based off-line real-time scheduling*. This paradigm is well adapted to our target application class: periodic embedded control and signal processing applications specified using data-flow synchronous languages like those used in safety-critical embedded systems design.

On the hardware side, the extensions presented in Section III allow the efficient implementation of global scheduling tables covering both CPUs and NoC resources. Computations and communications can be tightly synchronized, and NoC resources can be statically allocated with the same precision and flexibility as CPU time (as specified with *communication programs*).

The global scheduling tables are the result of a compilation process that follows a global optimization approach to define the allocation and scheduling of all computations and communications. To ensure that the timing precision provided by the hardware is well exploited by the software, we require that the allocation, the scheduling, and the software code generation

follow a set of rules meant to ensure the absence of contentions during access to shared resources.

Furthermore, *we achieve this with limited modifications of a general-purpose many-core platform*. This allows existing tools for optimized code generation (the gcc compiler) and for timing analysis to be used unmodified, and thus reduces the overall cost of the approach.

We have presented in previous publications details of our approach, namely the NoC architecture allowing static scheduling of communication [7] and the global off-line scheduling algorithm [8]. In this paper, we emphasize the importance of a global approach where (limited) modifications are made at all levels of the development environment, in both hardware and software, and for both scheduling and code generation.

b) Outline.: The remainder of the paper is organized as follows: Section II reviews related work. Section III presents the original many-core platform and the modifications we brought in both the NoC and the computing tiles. Section IV defines the rules we impose on code generation to ensure performance and predictability. Section VI presents ongoing work on the H.264 case study, Section V presents some experimental results, and Section VII concludes.

II. Related work

Our work had 3 main sources of inspiration. Previous work on off-line real-time scheduling [9], [10] has shown that table-based techniques allow an efficient allocation of resources in multi-processor architectures, especially if classical optimization techniques such as software pipelining are used [11]. The main difference with respect to this work is that we take into account architectural aspects that are specific to many-cores, such as the NoC-based communication system based on wormhole routing and with limited buffering capabilities.

Our second source of inspiration is previous work on the RAW many-core architecture [12] and on the StreamIt mapping tool for RAW [13]. The RAW architecture was the first to employ communication programs to control on-chip communications. There are several fundamental differences between this work and the work presented in this paper: The objective of RAW is to allow the many-core-wide use of compilation techniques that exploit Instruction Level Parallelism [14] and a very fine grain scheduling of computations and communications. We aim for a coarser level of control in both the NoC (transmission of packets instead of RAW's scalar values), and the software control of the NoC (which is performed through standard components such as caches and DMA units). Our approach allows the use of a classical shared memory programming model, general-purpose development tools, and existing applications. The StreamIt compiler uses scheduling tables as an internal compiler representation, but does not aim for real-time implementation. This is why it does not take into account timing interferences due to the mapping itself (which we do).

Our third major source of inspiration was previous work on the design of *general-purpose many-core* architectures using the SoCLib virtual prototyping library [15] and the DSPIN NoC [16]. Our work extends this NoC to provide good support to off-line real-time scheduling. Our changes retain the general-purpose character of the architecture by preserving its simple programming model and the ability to use general-purpose development tools.

Besides these 3 direct inspiration sources, our work is closely related to previous results from several fields. The idea of organizing a development environment and flow around a scheduling paradigm is not new. We already saw that static table-based scheduling was the basis of the RAW/StreamIt approach. We know of two other attempts. The first one is the *CompSoC* platform [17], which relies on a compositional scheduling and timing analysis approach where applications are assigned latency and throughput budgets on the computation and communication resources (such an approach can be generalized towards the use of full-fledged real-time calculus, like in [18]). The respect of these budgets is enforced using time division multiplexing (TDM) mechanisms on the various resources, such as the NoC, but the fine-grain synchronization between these TDM mechanisms is not required, nor used during timing analysis (only the latency/throughput budgets are used). By comparison, our approach allows the tight synchronization of computation and communication schedules, which improves timing precision and guaranteed performance, but requires a more static execution model than CompSoC. The second approach is based on the use of a priority-preemptive scheduling paradigm [19]. However, the target application class is even farther from the one we consider that the application class of CompSoC. Indeed, the cited paper considers the case of independent tasks, whereas our main focus is on dependent task systems.

More generally, our work is also closely related to previous work on the design of NoCs with support for real-time and safety-critical applications [20], [21], [22], [23] and on application mapping onto many-core architectures [24], [25], [26], [27], [28], the difference being given by the integrated approach we use and by the statically scheduled NoC communications which ensure high timing precision and efficiency for the chosen class of applications.

III. Hardware platform

A. Tiled many-cores in SoCLib

The SoCLib virtual prototyping library [15] allows the definition of tiled many-cores following a *distributed shared memory* paradigm where all memory banks and component programming interfaces are assigned unique addresses in a global address space. All memory transfers and component programming operations are represented with memory accesses organised as command/response transactions according to the VCI/OCP protocol [29]. To avoid interferences between commands and responses (which can lead to deadlocks), the on-chip interconnect is organized in two completely disjoint sub-networks, one for transmitting commands, and the other for responses.

There are two types of transactions: read and write. In write transactions the command sub-network carries the data to be written and the target address, and the response sub-network

carries a return code. In read transactions, the command sub-network carries the address and size of the requested data, and the response sub-network carries the data.

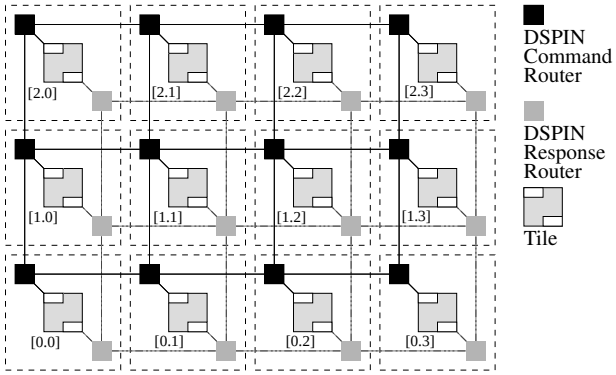

Fig. 1. General organization of our tiled many-core

As pictured in Fig. 1, the tiled many-cores of SoCLib are composed of a rectangular set of tiles connected through a state-of-the-art 2D synchronous mesh network-on-chip (NoC) called DSPIN [16]. The NoC is formed of a command NoC and a response NoC which are fully separated. Each tile has its own local interconnect, linked to the NoC and to the IP cores of the tile (CPUs, RAMs, DMAs, etc.).

SoCLib currently contains simulation models for a number of processors cores such as PowerPC 405, Sparc 7, ARM7, NIOSII and MicroBlaze and can be extended to take into account future cores.

B. Modifications of the tile structure

To improve timing predictability and worst-case performance, we modify both the tiles and the NoC of the SoCLib-based many-core. However, we retain the global organization of the many-core, and in particular its distributed shared memory model which allows programming using general-purpose tools. Fig. 2 pictures the structure of the computing tile in the original SoCLib many-core, and Fig. 3 its modified version.

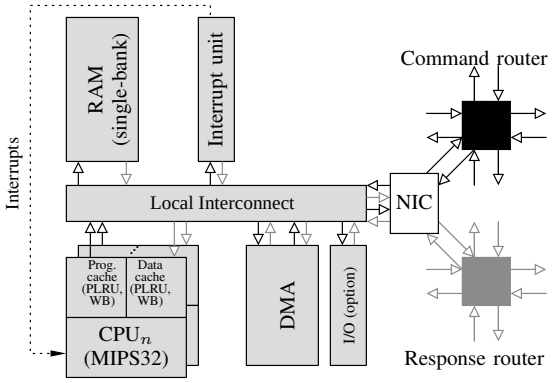

Fig. 2. The computing tile of the original many-core architecture

c) The memory subsystem.: Our objective here is to improve timing predictability by eliminating contentions. In our experiments with the original SoCLib-based many-core, the second most important source of contentions (after the NoC) is the access to the unique RAM bank of each tile. To reduce these contentions, we decided to follow the example of existing industrial many-core architectures [30], [20], and replace the single RAM bank of a tile with several memory banks that can be accessed independently.

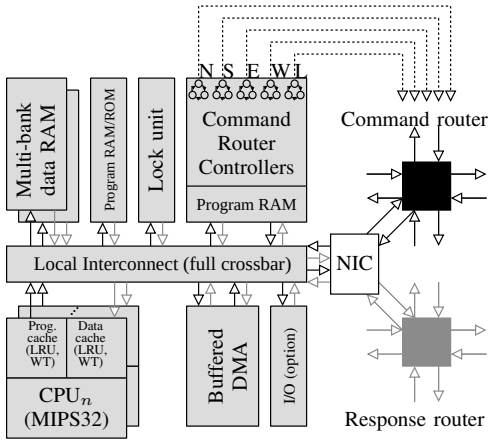

Fig. 3. Modified computing tile of our architecture

To facilitate timing analysis, we separate data (including stack) and program memory. One RAM bank is used in each tile to store the program of all the CPUs of the tile. Data and stack are stored on a *multi-bank RAM*. Each bank of the data RAM has a separate connection to the local interconnect. RAM banks of a tile are assigned contiguous address ranges, so that they can store data structures larger than a single tile. Explicit allocation of data onto the memory banks, along with the use of lock-based synchronization and the local interconnect presented below allow the elimination of contentions due to concurrent access to memory banks.

Note that the use of a multi-bank data RAM also removes a significant performance bottleneck of the original architecture. Indeed, a single RAM bank can only serve 4 CPUs (placing more than 4 CPUs per tile result in no performance gain because the RAM access is saturated). Having multiple RAM banks per tile removes this limitation. Our test configurations use a maximum of 16 CPU cores per tile and two data RAM banks per CPU core, for a maximum of 4Mbytes of RAM per tile.

d) The local interconnect: It is chosen in our design so that it cannot introduce contentions due to its internal organization. Contentions can still happen, for instance, when two CPUs access concurrently the program memory. However, accesses from different sources to different targets never introduce contentions. Interconnect types allowing this are the full crossbars and the multi-stage interconnection network [31] such as the omega networks, the delta networks, or the related logarithmic interconnect [32]. The experiments of this paper use a full crossbar interconnect.

e) The CPU core: The original architecture uses a single-issue, in-order, pipelined implementation of the MIPS32 ISA with no speculative execution. We did not change this, as it simplifies timing analysis and allows small-area hardware implementation. However, significant work has been invested in designing a cycle-accurate model of this core inside a state-of-the art WCET analysis tool [33].

f) The caches: They have been significantly modified. The original design featured caches with a pseudo-LRU (PLRU) replacement policy and with a writing policy that is intermediate between write-through and write-back.[1] Memory accesses from the data and instruction caches of a single CPU were multiplexed over a single connection to the local interconnect of the tile. All these choices are known to complicate timing analysis and/or to reduce the precision of the analysis [34], [35], and thus we revert to more conservative choices: We use the LRU replacement policy, a fully write-through policy, and we let the instruction and data caches access the local tile interconnect through separate connexions. Note that the use of a write-through policy reduces the processing speed of each CPU. This is the only modification we made on the architecture that decreases processing speed.

g) Synchronization: To improve temporal predictability, and also speed, our architecture does not use interrupt-based synchronization. Interrupt signaling by itself is fast, but handling an interrupt usually requires accesses to program memory which take supplementary time. Furthermore, arrival date imprecision and modifications of the cache state result in supplementary imprecision during static timing analysis. To avoid these performance and predictability problems, we replace the interrupt unit present in each tile of the original architecture with a hardware lock component. These components allow synchronization with very low overhead (1 non-cached local RAM access) and without modifications of the cache state. The lock unit follows a simple request/grant protocol.

h) Buffered DMA: The traditional DMA unit used in the original architecture requires significant software control to determine when a DMA operation is finished so that another can start. This is either done using interrupt-based signaling, which has the problems mentioned above, or through polling of the DMA registers, which requires significant CPU time and imposes significant constraints on CPU scheduling.

To avoid these problems, we use DMA units allowing the buffering of transmission commands. A CPU can send one or more DMA commands while the previous DMA operation is not yet completed. Furthermore, the DMA unit can be programmed so that it not only sends out data, but also signals the end of the transmission to the target tile by granting a lock, as described in Section IV. Thus, all inter-tile communication and synchronization can be performed by the DMA units, in parallel with the data computations of the CPUs and without requiring significant CPU time for control.

C. Modifications of the NoC

The DSPIN network-on-chip [16] is a classical 2D mesh NoC. It uses wormhole packet switching and a static routing scheme[2]. Each router of the command or response NoC has the internal structure of Fig. 4. Each NoC router is connected

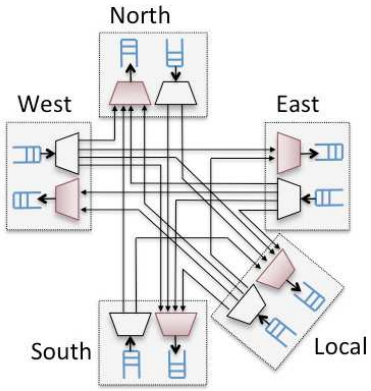

Fig. 4. Structure of a DSPIN NoC router

through FIFO links with the 4 neighboring routers (denoted with North, South, West, East) and with the local computing tile. Each of these connections is realized through one demultiplexer and one multiplexer. The demultiplexer ensures the routing function (X-first/Y-first). It reads the headers of the incoming packets and, depending on the target address, sends them towards one of the multiplexers of the router. The multiplexer ensures the arbitration (scheduling) function. When two packets arrive at the same time (from different demultiplexers), a fair Round Robin arbiter is used to decide which one will be transmitted first. Once the transmission of a packet is started, it cannot be stopped.[3]

The fair arbitration scheme is well-adapted to applications without real-time requirements, for which it ensures a good use of NoC resources. But when the objective is to provide real-time guarantees and to allocate NoC resources according to application needs, it is better to use some other arbitration mechanism. In our case, the objective is to provide the best possible support for the implementation of static computation and communication schedules. Therefore, we rely on a *programmed arbitration* approach where each router multiplexer enforces a fixed packet transmission order specified under the form of a *communication program*.

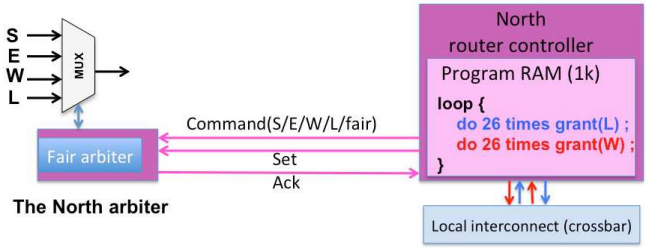

Fig. 5. Programmed arbitration in a NoC router multiplexer

Enforcing fixed packet transmission orders requires the use

[1]Consecutive writes inside a single cache line are buffered.

[2]X-first routing for the command network and Y-first for the response network.

[3]Unless a two-level virtual channel mechanism is used, as described in [16].

of new hardware components called *router controllers*. These components are present in the tile description of Fig. 3 and their interaction with the NoC router multiplexers is realized as pictured in Fig. 5. Each multiplexer of the command network is controlled by its own controller running a separate program. The program is loaded onto the component through the local interconnect. The interface between router controller and the local interconnect is also used to start or stop the router controller. When the controller is stopped, the fair arbiter of DSPIN takes control. In Fig. 5, the arbitration program will cyclically accept 26 packets from the Local connection, then 26 packets from the West connection.

More details on the implementation and properties of our programmed arbitration mechanism can be found in [7].

IV. Software organization

On the hardware architecture defined above, we use non-preemptive, statically-scheduled software with lock-based inter-processor synchronization. To ensure both performance and predictability, we require that software follows the organization rules detailed in this section.

i) Data locality: We require computation functions to only operate on local tile data. If a computation needs data from another tile, this data is first transferred to the local data RAM. Under this hypothesis, the timing (WCET/BCET) analysis of computation functions can be performed without considering the NoC.

j) Inter-tile data transfers: They are only performed using the DMA units. The CPUs still retain the possibility of directly accessing RAM banks of other tiles, but they only do so during the boot phase (which follows the standard protocol of the MIPS32 ISA), or for non-real-time code running on many-core tiles allocated to such code. Traffic generated directly by CPUs and their caches has very small grain (usually a single data word per memory write access), and it is difficult to accurately predict its timing. Thus, not allowing it to traverse the NoC largely simplifies the timing analysis of both NoC transfers and CPU code [36].

Inter-tile data transfers and synchronizations are only performed through write transactions performed by the DMA unit of the sending tile. Thus, the response NoC only carries 2-flit acknowledge packets, so that contentions on the response NoC are negligible even in the absence of programmed arbitration. This is why router controllers are only used for the command NoC multiplexers, leaving unchanged the fair arbiters on the response network.

k) Allocation of the tile memory: The memory allocation scheme we used for automatic code generation and for the case studies makes several assumptions. First, we assume that the programs of all CPUs in a tile are stored in the local program memory. This amounts to either assuming that this memory is a non-volatile one, or that the loading of the program is performed during a boot phase, so that the NoC only transfers data.

Second, we allocate one of the data RAM banks for the stacks of all the CPUs of the tile. Using only one RAM bank for all the stacks is possible because our applications only make little use of the stack (most data is explicitly allocated by our tool on the other memory banks).

Allocating all programs (respectively all stacks) on a single memory bank means that the cost of a cache miss due to a program (resp. stack) memory access can be very high, due to interference from the other CPU cores of the tile. However, the (relatively) small size of the programs (resp. stacks) means that misses seldom occur. For applications with large programs or with significant use of the stack, other memory allocation approaches can be used.

All data RAM banks except the stack-dedicated one are allocated to data variables. To each data variable we associate a contiguous memory region with statically-defined start address and length. The length of the region must be equal to the worst-case size of the data type of the variable (which must be provided by the programmer).

A. Communication and synchronization

The allocation of memory regions to the data variables and the synchronization between computations and communications must ensure that no data RAM bank is accessed from two sources at the same time. Under the previously-made assumptions, a RAM data bank can be accessed from 3 sources: local tile CPU caches, local tile DMA, and incoming NoC data sent by the DMA of some other tile.

In our framework based on static real-time scheduling, ensuring exclusive access to memory banks amounts to:

- Building a scheduling table where no two computations or communications read or write the same piece of data at the same time, and
- Implementing the scheduling table (and more precisely allocating the variables to the memory banks) in a way that preserves this exclusive access property.

The off-line scheduling algorithms we use, which are presented in [8], ensure the first property. Code generation ensures the second property through the use of hardware locks.

We illustrate this with an example involving inter-tile communications. We consider the scheduling table of Fig. 6. This figure provides part of a larger scheduling table for a NoC with the topology of Fig. 1. We only included here the allocation and scheduling for one CPU of tile (1,1), one CPU of tile (2,2), and the communication resources along the path from

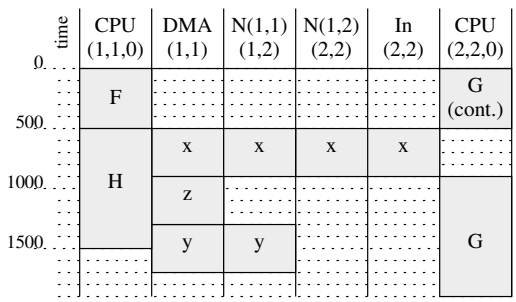

Fig. 6. Partial view of a scheduling table with inter-tile communications

tile (1,1) to tile (2,2).[4] These resources are, in order, the DMA of tile (1,1), the NoC link from the router of tile (1,1) to that of tile (1,2), the NoC link between the routers (1,2) and (2,2), and the link from the router of tile (2,2) to the tile (2,2). We assume that processor $CPU(1,1,0)$ executes the functions F and H, and processor $CPU(2,2,0)$ executes function G. We also assume that F produces data x, y, and z, and that x is needed by G, z is needed on tile (0,1), and y is needed on tile (1,2).

Each vertical lane of the scheduling table shows the time windows allocated on one resource for the execution of one operation (computation or communication). For instance, our scheduling table allocates $CPU(1,1,0)$ for the execution of F between dates 0 and 500 (time is measured in clock cycles). Scheduling tables represent cyclic execution patterns, meaning that once the pattern of the table is completed it immediately restarts. Thus, one operation can start in one cycle of the table and complete in a subsequent one (building such tables must follow the rules laid out in [11]). In our case, $CPU(2,2,0)$ is reserved for the execution of G starting at date 907 in one cycle (the date when the x produced in the current cycle is fully available on tile (2,2)) and until date 507 in the next cycle (when the first words of x of the for the next cycle arrive on tile (2,2)).

Note that the scheduling table is correct, in the sense that it ensures that the memory zone allocated to x on tile (2,2) is never accessed by both G and the transmission of x. However, the implementation of this scheduling pattern can only make use of the lock-based inter-processor synchronization mechanism defined above. In the general case, two locks are needed: one to ensure that G starts after the reception of x (produced in the current cycle), and the other to ensure that no part of x arrives on tile (2,2) before G of the previous cycle completed its execution, and the timing overhead due to these locks must be taken into account during the construction of the scheduling table through overheads added to the durations of the various operations (otherwise, the table may be unimplementable).

Another cost that must be taken into account during construction of the scheduling tables is that of DMA control by the CPUs. This cost is very small, but it is not zero (the cost of issuing a DMA command by a CPU is over-approximated at 30 clock cycles). When implementing the scheduling table of Fig. 6, $CPU(1,1,0)$ must start 3 DMA operations in every cycle. Due to the non-preemptive execution model, these operations cannot be initiated during execution of H (as pictured in the table). Given the ordering constraints, the commands must be issued at the end of F and before H starts. The scheduling of operations on processors must take into account the timing overhead due to these operations.

V. Results

We have evaluated our mapping and code generation method on a platooning automotive application described in [37], and on a parallel Cooley-Tukey implementation of the integer 1D radix 2 FFT over 2^{14} samples [38]. We chose these two applications because they allow the computation of tight lower bounds on the execution cycle makespan and because for the FFT the cited reference provides a mapping onto NoC-based 2D tiled MPPAs. This allows for meaningful comparisons, while no tool equivalent to ours exists to provide another basis for evaluation. Evaluation is done on the 3x4 MPPA pictured in Fig. 1, where we assume that input data arrives on *Tile*$(0,0)$ and the results are output by *Tile*$(2,3)$.

For both applications, after computing the WCET of the tasks and the WCCT of the data transmissions, the mapping tool (described in more detail in [8]) was applied to build a running implementation and to compute execution cycle makespan and throughput guarantees. Then, the code was run, and its performances measured. This allowed us to check the functional correctness of the code and to determine that our tool produces very *precise timing guarantees*. Indeed, the difference between predicted and observed makespan and throughput figures is less than 1% for both examples, which is due to the precision of our mapping algorithms and to the choice of a very predictable execution platform.

The generated off-line schedule (and the resulting code) has *good real-time properties*. For both the CyCab and the FFT, we have manually computed lower bounds on the execution cycle makespan.[5] The lower bounds computed for the CyCab and FFT examples were lower than the makespan values computed by our algorithms by respectively 8.9% and 3.4%.

For the FFT example, we have also compared the measured makespan of our code with that of a classical NoC-based parallel implementation of the FFT [38] running on our architecture. For our code, the NoC was statically scheduled, while for the classical implementation it was not. Execution results show that our code had a latency that was 3.82% smaller than the one of the classical parallel FFT code. In other words, **our tool produced code that not only has statically-computed hard real-time bounds (which the hand-written code has not) but is also faster**.

Our mapping heuristics favor the concentration of all computations and communications in a few tiles, leaving the others free to execute other applications (as opposed to evenly spreading the application tasks over the tiles). The code generated for Cycab has a tile load of 85%-99% for 6 of the 12 tiles of the architecture, while the other tiles are either unused or with very small loads (less than 7%). Using more computing tiles would bring no latency or throughput gains because our application is limited by the input acquisition speed. In the FFT application the synchronization barriers reduce average tile use to 47% on 8 of the 12 MPPA tiles. Note that the remaining free processor and NoC time can be used by other applications. Also note that these results are obtained using an allocation and scheduling heuristic whose sole objective is the optimization of speed (latency and throughput). Other heuristics may be needed when the optimization objective

[4]Naming convention: the tiles are identified by their (x,y) coordinates in the 2D mesh, and the CPU identifiers use a third integer to identify the CPU inside the tile.

[5]To compute these lower bounds we simplify the hardware model by assuming that the resources $N(i,j)(k,l)$ generate no contention (*i.e.* they allow the simultaneous transmission of all packets that demand it). We only take into account the sequencing of operations on processors and DMAs and the contentions on resources $In(i,j)$.

includes power consumption or thermal management issues.

Finally, we have measured the influence of static scheduling of NoC communications on the application latency, by executing the code generated for Cycab and the FFT with and without NoC programming. For Cycab, not programming the NoC results in a speed loss of 7.41%. For the FFT the figure is 4.62%.

VI. Ongoing work: the H.264 case study

Mapping techniques based on off-line scheduling are traditionally used on applications with little data-dependent control (represented under the form of execution modes or conditional execution). Such applications are usually at the core of safety-critical embedded control systems, and dedicated languages (such as Scade or Simulink [39]) exist for their specification. However, it is an open question what types of specification can *realistically* be mapped using off-line scheduling approaches. This question bears high practical significance today, when complex embedded systems are composed of multiple applications with different characteristics which must be mapped on the same platform.

To understand the limits of our mapping approach, and thus clarify the price to pay for the good results of the previous section, we consider a very different application class (telecommunications/video streaming), and an application involving significant data-dependent control: a decoder for the popular H.264 [40] video format. Our objective is to determine under which H.264 encoding options the decoding process can be given an efficient implementation relying on off-line scheduling.

Significant work exists on parallelizing H.264 and its successor HEVC [41], [42], and some of this work explicitly addresses the issue of statically parallelizing the decoding process. However, existing work mainly considers architectures where the frames used for *motion compensation* are stored in a central memory, known as the *frame buffer*, which the decoding threads that run in parallel access concurrently. This memory organization is used in Fig. 7, which pictures the tasks of the H.264 implementation in the *task and communication graph* formalism used by the Design Space Explorer tool [43] developed for SoCLib.

In this implementation, the H.264 data stream is read by a *traffic generator* task TG. The stream is then sent to the *Split* task, which detects the markers delimiting frame slices. These slices are then dispatched to the decoding pipelines in a round-robin manner. After decoding, the decoded image slice leave the *Decode* tasks. Slice data from the different pipelines are joined by the *Merge* task and sent on to the *RAMDAC* for display. Each pipeline is mapped onto a separate MPPA tile. The *TG* and *Split* tasks on the one hand, the *Merge* and *RAMDAC* tasks on the other hand, are mapped together on an input and output tile, respectively.

The use of a central frame buffer is known to be a bottleneck in existing decoders. However, little can be done if no restrictions are imposed on the H.264 encoding process, because in the most general case no bound is imposed on the motion vector lengths or on the distance between a frame and the reference frame(s) used for motion compensation.

The use of motion vector bounds has already been proposed in [41] to facilitate the static mapping of the H.264 decoder. What we propose is to use these bounds to allow the complete removal of the central frame buffer, which is to be replaced with small frame buffers associated with each MPPA tile. Our approach relies on representing the application under the form of a dependent task system encoded as a data-flow program written in a synchronous language similar to Scade [39]. Such a representation allows automatic parallel mapping using the tool presented above. The motion vector and reference frame bounds allow limiting the dependencies between the data-flow blocks, which in turn allows the mapping of each block (with its input buffers) in the constrained memory space of an MPPA tile.

Note that the memory organization of the original H.264 decoder is representative of a large class of parallel applications where multiple threads directly access (read and write) shared data using semaphore-based synchronization mechanisms. Our ongoing work on H.264 shows that for *some* of these algorithms it is possible to put them in a data-flow form facilitating predictable implementation.

The performance evaluation of our technique will have to:

- Compare the performance of the code generated by our approach against that of existing implementations relying on a centralized frame buffer.
- Determine the efficiency loss in H.264 encoding due to constraints imposed by the off-line mapping approach.

VII. Conclusion and Perspectives

We have shown that taking into account the fine detail of the many-core hardware and software architecture allows real-time implementation of very good precision and predictability. We have also shown that it is possible to achieve this with limited modifications to a general-purpose many-core platform, which allows us to use, unmodified, existing tools for optimized code generation and timing analysis (which largely reduces the overall cost of the approach).

In addition to these results, which mainly show the advantages of our approach, we have also presented ongoing work on the H.264 case study, which is meant to better identify its limitations. In this case, the limitations are materialized as constraints on the H.264 encoding parameters, and thus the efficiency of the encoding process.

References

[1] G. E. Moore, "Moore's law." Online http://en.wikipedia.org/wiki/Moore's_law, 1965.

[2] F. J. Pollack, "New microarchitecture challenges in the coming generations of CMOS process technologies," in *Proceedings MICRO 32*. Washington, DC, USA: IEEE Computer Society, 1999.

[3] L. J.Flynn, "Intel halts development of 2 new microprocessors." *New York Times*, May 2004.

[4] "The TilePro64 many-core architecture." www.tilera.com, 2008.

[5] "The Epiphany many-core architecture." www.adapteva.com, 2012.

[6] "The MPPA256 many-core architecture." www.kalray.eu, 2012.

[7] M. Djemal, F. Pêcheux, D. Potop-Butucaru, R. de Simone, F. Wajsbürt, and Z. Zhang, "Programmable routers for efficient mapping of applications onto NoC-based MPSoCs," in *Proceedings DASIP*, Karlsruhe, Germany, 2012.

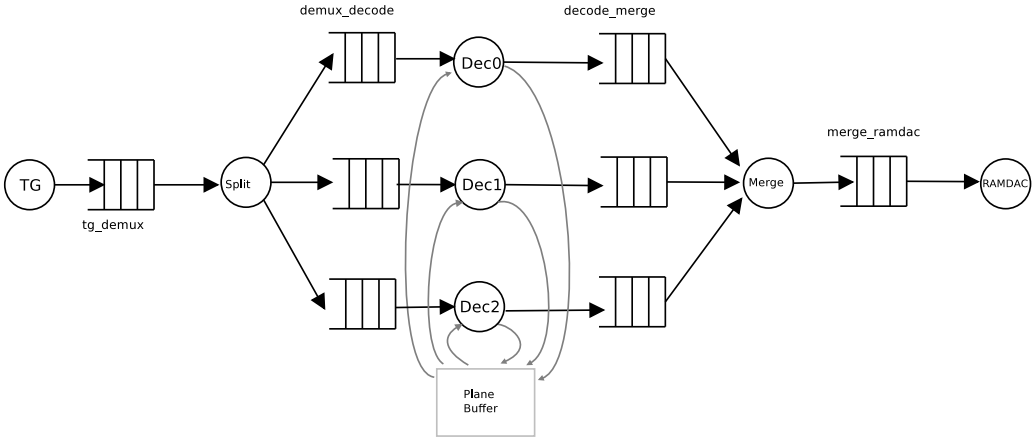

Fig. 7. Task and Communication Graph of parallel H.264

[8] T. Carle, M. Djemal, D. Potop-Butucaru, R. D. Simone, and Z. Zhang, "Off-line mapping of real-time applications onto massively parallel processor arrays," INRIA, Research Report RR-8429, Dec. 2013.

[9] G. Fohler and K. Ramamritham, "Static scheduling of pipelined periodic tasks in distributed real-time systems," in *Proceedings of EUROMICRO-RTS97*, 1995.

[10] T. Grandpierre and Y. Sorel, "From algorithm and architecture specification to automatic generation of distributed real-time executives: a seamless flow of graphs transformations," in *Proceedings MEMOCODE*, Mont Saint-Michel, France, June 2003.

[11] T. Carle and D. Potop-Butucaru, "Throughput Optimization by Software Pipelining of Conditional Reservation tables," INRIA, Research report RR-7606, Apr. 2011, to appear in ACM TACO. [Online]. Available: http://hal.inria.fr/inria-00587319

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

[13] S. Amarasinghe, M. I. Gordon, M. Karczmarek, J. Lin, D. Maze, R. Rabbah, and W. Thies, "Language and compiler design for streaming applications," *Int. J. Parallel Program.*, vol. 33, no. 2, Jun. 2005.

[14] M. B. Taylor 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.

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

[16] I. M. Panades, A. Greiner, and A. Sheibanyrad, "A low cost network-on-chip with guaranteed service well suited to the GALS approach," in *Proceedings NanoNet'06*, Lausanne, Switzerland, Sep 2006.

[17] K. Goossens, A. Azevedo, K. Chandrasekar, M. Gomony, S. Goossens, M. Koedam, Y. Li, D. Mirzoyan, A. Molnos, A. B. Nejad, A. Nelson, and S. Sinha, "Virtual execution platforms for mixed-time-criticality applications : the CompSoC architecture and design flow," in *Proceedings CRTS*, San Juan, Puerto Rico, Dec 2012.

[18] I. Bacivarov, W. Haid, K. Huang, and L. Thiele, "Methods and tools for mapping process networks onto multi-processor systems-on-chip," in *Handbook of Signal Processing Systems*. Springer, 2013.

[19] Z. Shi and A. Burns, "Schedulability analysis and task mapping for real-time on-chip communication," *Real-Time Systems*, vol. 46, no. 3, pp. 360–385, 2010.

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

[21] C. E. Salloum, M. Elshuber, O. Hftberger, H. Isakovic, and A. Wasicek, "The across mpsoc a new generation of multi core processors designed for safety critical embedded systems," in *Proceedings DSD*, Izmir, Turkey, Sep. 2012.

[22] F. Brandner and M. Schoeberl, "Static routing in symmetric real-time network-on-chips," in *Proceedings of the 20th International Conference on Real-Time and Network Systems*, ser. RTNS '12, 2012.

[23] M. Gerdes, F. Kluge, T. Ungerer, C. Rochange, and P. Sainrat, "Time analysable synchronisation techniques for parallelised hard real-time applications," in *Proceedings DATE'12*, Dresden, Germany, 2012.

[24] V. Bebelis, P. Fradet, A. Girault, and B. Lavigueur, "A framework to schedule parametric dataflow applications on many-core platforms," in *Proceedings CPC'13*, Lyon, France, 2013.

[25] D. Genius, A. M. Kordon, and K. Z. el Abidine, "Space optimal solution for data reordering in streaming applications on noc based mpsoc," *Journal of System Architecture*, 2013.

[26] J. T. Zhai, M. Bamakhrama, and T. Stefanov, "Exploiting just-enough parallelism when mapping streaming applications in hard real-time systems," in *Proceedings DAC*, 2013.

[27] S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, Eds., *Handbook of Signal Processing Systems*. Springer, 2013, 2nd edition, in particular chapter.

[28] P. Aubry et al., "Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor," in *Proceedings ALCHEMY 2013*, Barcelona, Spain, June 2013.

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

[30] D. Melpignano, L. Benini, E. Flamand, B. Jego, T. Lepley, G. Haugou, F. Clermidy, and D. Dutoit, "Platform 2012, a many-core computing accelerator for embedded SoCs: performance evaluation of visual analytics applications," in *Proceedings DAC'12*, San Francisco, CA, USA, June 2012.

[31] Y. Aydi, M. Baklouti, M. Abid, and J.-L. Dekeyser, "A multi-level design methodology of multistage interconnection network for mpsocs," *IJCAT*, vol. 42, no. 2/3, pp. 191–203, 2011.

[32] M. R. Kakoee, "Reliable and variation-tolerant interconnection network for low power mpsocs," Ph.D. dissertation, Universitá di Bologna, 2012, online at http://amsdottorato.unibo.it/4407/1/phdthesis.pdf.

[33] I. Puaut and D. Potop-Butucaru, "Integrated worst-case execution time estimation of multicore applications," in *Proceedings WCET'13*, Paris, France, July 2013.

[34] R. Wilhelm et al., "The worst-case execution-time problem overview of methods and survey of tools," *ACM TECS*, vol. 7, no. 3, May 2008.

[35] D. Hardy and I. Puaut, "Wcet analysis of multi-level non-inclusive set-associative instruction caches," in *RTSS*, 2008.

[36] R. Wilhelm and J. Reineke, "Embedded systems: Many cores – many problems (invited paper)," in *Proceedings SIES'12*, Karlsruhe, Germany, June 2012.

[37] 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.

[38] J. H. Bahn, J. Yang, and N. Bagherzadeh, "Parallel FFT algorithms on network-on-chips," in *Proceedings ITNG 2008*, april 2008.

[39] P. Caspi, A. Curic, A. Magnan, C. Sofronis, S. Tripakis, and P. Niebert, "From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications," in *Proceedings LCTES*, San Diego, CA, USA, June 2003.

[40] ISO/IEC, *Advanced Video Coding. Information technology Coding of audio-visual objects*, 2010.

[41] B. Juurlink, M. Alvarez-Mesa, C. C. Chi, A. Azevedo, C. Meenderinck, and A. Ramirez, *Scalable Parallel Programming Applied to H.264/AVC Decoding*. Springer, 2012.

[42] E. Bezati, M. Mattavelli, and M. Raulet, "Rvc-cal dataflow implementations of mpeg avc/h.264 cabac decoding," in *Proceedings DASIP'10*, 2010.

[43] N. Pouillon, “Modèle de programmation pour applications parallèles multitâcches et outil de dṕloiement sur architecture multicore à mémoire partagée,” Ph.D. dissertation, Univ. Pierre et Marie Curie, sept 2011.