Building a RTOS for MPSoC Dataflow Programming

Yaset Oliva, Maxime Pelcat, Jean-Francois Nezan,
Jean-Christophe Prevotet
IETR, INSA Rennes, CNRS UMR 6164, UEB
20, Av. des Buttes de Coesmes, 35708 Rennes
Email: yaset.oliva, maxime.pelcat, jean-francois.nezan,
jean-christophe.prevotet@insa-rennes.fr

Slaheddine Aridhi
Texas Instruments
06271 Villeneuve Loubet, France
Email: saridhi@ti.com

***Abstract*— Multiprocessor Systems-on-Chip (MPSoC) are becoming the standard high performance Digital Signal Processing (DSP) systems. Hardware complexity abstraction is needed to enable efficient MPSoC programming. A major challenge of MPSoC programming is efficiently handling the combination of new features necessary in a MPSoC operating system: load balancing and efficient use of the parallel resources, with the more traditional features of Real-Time Operating Systems (RTOS): resource sharing between applications, task priorities and reactivity to events. This paper presents a method to combine dataflow methods and RTOS features. The resulting system prototypes an RTOS for symmetric multiprocessing MPSoCs whose inputs are dataflow graphs of applications. The prototype is built on the μC/OS-II RTOS. Experimental results are given on a 3GPP Long Term Evolution algorithm executed on a 4-core MPSoC.**

I. Introduction

In [1], Edward Lee shows that programming with threads is an error-prone operation and proposes several alternatives, including using process networks, to make software behavior more predictable. The dataflow process network [2] Model of Computation (MoC) models an algorithm by concurrent and independent modules known as actors which communicate ordered tokens (data quanta) through First-In First-Out channels. A set of firing rules defines when an actor executes. Dataflow models have been shown to favor parallel algorithm description as they favor data locality and reduce multi-core scheduling constraints to data dependencies [3]. Thus, dataflow models are well suited for use with signal processing algorithms

MPSoC systems used for signal processing are increasingly complex to program. Tools that ease MPSoC programming are thus more and more needed. This paper outlines a method of using dataflow graphs instead of thread declarations as inputs to an RTOS. In this case, the RTOS is then able to dispatch the actors which compose the dataflow graph to the cores of an MPSoC. These actors are directly executed by the MPSoC RTOS, instead of requiring the programmer to program primitives for task migration and synchronization. This method ensures the synchronization of actors and the management of actor input and output data.

This paper presents experiments performed with a MPSoC RTOS prototype incorporating dataflow model management into the μC/OS-II kernel. The first target application is the uplink data decoding algorithm executed in base stations supporting the 3GPP Long Term Evolution (LTE) telecommunication standard.

Section II presents related works on multi-core runtime management and dataflow MoCs. Section III introduces the MPSoC RTOS structure. Section IV explains the dataflow management part and Section V considers the multi-core scheduling part. Experimental results of the prototype are shown in Section VI.

II. Related Works

A. RTOS and Runtime Management Systems

In [4], Nollet, et al. present an overview of runtime management systems for MPSoCs. The overview covers both industrial and academic systems. Commonalities between systems are identified: their structure can be divided into two parts, the quality manager and the resource manager. The quality manager tries to optimize the Quality of Service (QoS) of the system, i.e. to find the best application configuration; the resource manager offers mechanisms to allocate cores, communication media and memory.

The runtime systems reviewed in [4] consider only sequential applications which share cores of an MPSoC. The MPSoC RTOS presented in this paper differs, as it partitions each application between available cores. Parallelism of each application is explicitly stated using a dataflow graph. Such partitioning aims at ensuring good load balancing between cores and at reducing the computation latency of each application. In this way, the MPSoC RTOS may be considered to be equivalent to a quality manager of low granularity. The StreamIt [5] runtime system has similar goals than the MPSoC RTOS. However, it processes a specific streaming language while the MPSoC RTOS reuses C/C++ legacy code for the actors and combines it with a parameterized dataflow coordination language.

The MPSoC RTOS prototype is built on an existing kernel: μC/OS-II [6]. This choice was made through consideration of the small footprint, the simplicity and the available source code for this kernel. Since the original μC/OS-II doesn't implement

978-1-4577-0672-1/11/$26.00

any multiprocessor mechanism, its sources are modified to manage MPSoCs.

B. Modeling Applications with a Parameterized Dataflow Model

Many dataflow MoCs have been introduced in the literature, each offering a tradeoff between compile-time predictability and capacity to model dynamic run-time variability. It may be seen that the most obvious difference is their firing rules. Signal processing applications, such as telecommunication or video processing, are based on a loop which repeats a pattern of execution on a sequence of input data. Describing the repeated pattern with a dataflow MoC consists of dividing the computation into actors that exchange data only through input and output data queues without sharing any state.

In this paper, applications are described using parameterized dataflow modeling [7]. The chosen model of Parameterized Cyclo-Static Directed Acyclic Graph (PCSDAG) states that the graph can be totally reconfigured once before starting a new execution. This model is a parameterized and acyclic version of the Cyclo-Static Dataflow (CSDF) model [8]. The reconfiguration enables dynamic behavior of the algorithm, i.e. computation strongly depending on the input data. The PCSDAG model has been shown to be suitable for describing uplink and downlink data processing algorithms of 3GPP LTE base stations [9].

III. Structure of the MPSoC Model-Based RTOS

The RTOS obtained is divided into two modules: dataflow management and RTOS scheduling (Figure 1). The dataflow graph is described with C++ objects and compiled within the dataflow management module. At each OS clock tick, the dataflow graph is parameterized, i.e. new parameter values are retrieved, and a temporary graph of execution is obtained. For each actor within the temporary graph, an RTOS task is created. The task corresponding to an actor must manage actor data and call actor code. Flags are automatically generated to synchronize actors based on their dependencies. RTOS scheduling is completed by assigning each actor to a core.

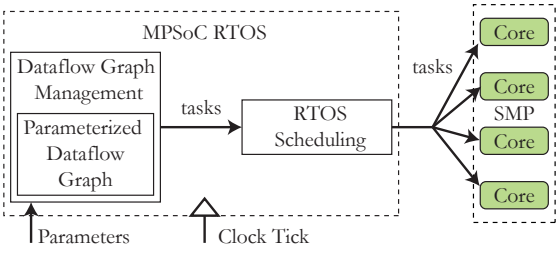

Fig. 1. Structure of the MPSoC Model-Based RTOS

Next sections detail the dataflow management and RTOS scheduling modules.

IV. Dataflow Graph Management

The test application used is the uplink decoding of 3GPP LTE base stations. The PCSDAG model was specifically designed for this application but is likely to be also suitable for video encoding and decoding algorithms. Managing the graph consists of performing an expansion. This is the process where the dataflow graph is transformed into the temporary graph in which each actor is executed only once in a graph iteration. The temporary graph depends strongly on parameter values that fluctuate between consecutive iterations.

Figure 2 illustrates the application and the expansion. Uplink decoding consists of retrieving data sent simultaneously by nb_{user} users. This data is divided into Code Blocks (CB) of variable size. Both the number of users nb_{user} and the number of code blocks nb_{CB} for each user vary every millisecond. nb_{CB} is different for each user and may be represented by a cyclic pattern $nb_{CB} = nb_{CB_{user0}}, nb_{CB_{user1}}, ...$ The computation is divided into four phases: Multiple Input Multiple Output (MIMO) decoding, symbol to bit conversion, bit processing, and Cyclic Redundancy Check (CRC) [9]. Each millisecond, the graph is parameterized and transformed into a single rate directed acyclic temporary graph (srDAG) in which production and consumption rates of each FIFO queue are equal. Figure 2 shows 4 possible configurations. The expansion phase has been demonstrated to be executable in real-time while respecting the 3GPP LTE constraints in [9].

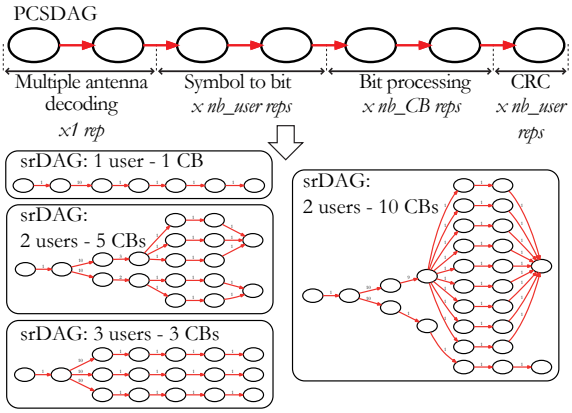

Fig. 2. Expanding the input dataflow model.

After the expansion phase, an RTOS task is created for each actor in the srDAG graph. These tasks are then scheduled on the multiple cores of the MPSoC architecture. It may be noted that an alternate algorithm model to PCSDAG can be chosen provided that the srDAG graph can still be produced.

V. Multi-core Scheduling

RTOS scheduling is divided into two phases: the master/slave phase and the symmetric phase. In the master/slave phase, the core designated as master executes the dataflow graph management and posts actors to the slave cores and to itself. When the scheduling enters the symmetric phase , each core is part of a pair and calls successively the common μC/OS-II scheduler to execute the highest priority task. The objective of this division is to limit the master/slave phase that naturally puts the master in the position of the bottleneck. The

symmetric phase also enables preemptions and passive wait of events.

A. Master/Slave Phase

The master/slave phase is illustrated in Figure 3. Each core has a 1-place queue to receive an order of task execution. After an OS timer tick, the previously described dataflow graph management is performed and one task per actor is created. The master processor then dispatches the ready tasks with highest priority to the available slave processors. The dispatching process is performed by adding a message into the 1-place queue of the selected slave. The message contains the address of the shared memory location where the code of the task has been placed as well as input/output buffer management information. The dispatching process finishes when all ready tasks have been mapped or when there are no more available slave processors. When all slave processors are busy and there are still ready tasks, then the master processor executes the next task itself.

At the same time as the master executing, each slave processor waits for a message to be placed into its 1-place queue. When the message arrives, the slave processor places its program counter to the designed address and executes the task. After this first task dispatching is done, the symmetric phase starts and all cores become pairs.

B. Symmetric Phase

During the symmetric phase (Figure 4), each core can access the schedule function. This common access requires the use of mutex semaphores provided within the architecture support library. As in a typical RTOS, the scheduler function consists of identifying the highest-priority task that is ready and running it. The scheduler is called if either the task execution is preempted by a higher execution task or if the task execution is completed.

After the execution of the schedule function, and if no task is ready to be executed by the core, the scheduler saves the current task's context state and gets returned to its private memory. The slave is now ready to receive another message from the master processor.

Using RTOS scheduling, several independent applications can share the MPSoC with each actor with its own priority. Moreover, passive wait of events is allowed. This is very important in the case of systems with co-processors. In the cases where a programmer wishes to offload a costly operation such as turbo-coding on a co-processor, an actor can wait for turbo-coding completion passively on a core and can be preempted by another ready actor while turbo-coding is running.

The next section gives experimental results of the prototype.

VI. Experimental Results

The experimental prototype is executed on an Altera Cyclone II FPGA integrating 4 NIOS cores [10] and a shared memory. The memory footprint of the system is shown in Figure 5. As can be seen, the operating system data occupies more than 50% of the 510kb of total system memory. This expensive cost is largely due to the size and number of task stacks statically allocated by the kernel (64 in this case).

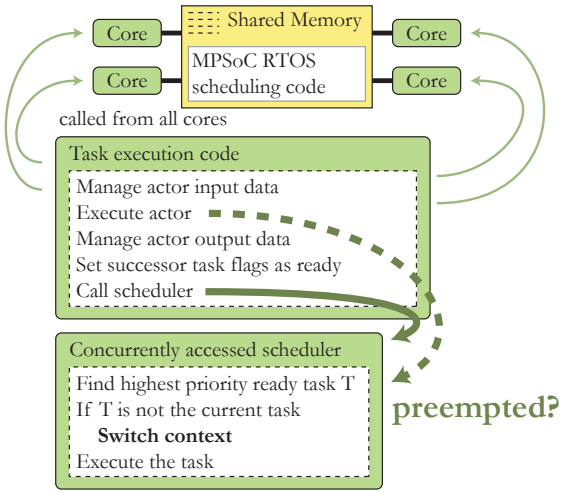

Fig. 4. Symmetric Scheduling Phase: Self-Organizing Execution

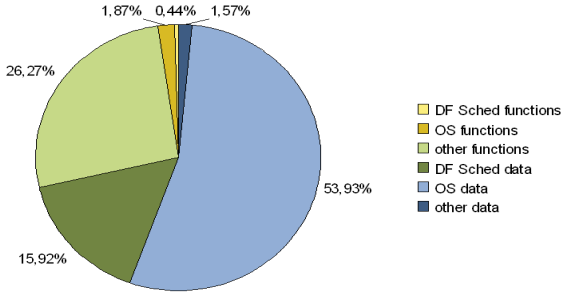

Fig. 5. Memory Usage

The prototype has been run for six iterations. At the end of each iteration, the application parameter values are modified to generate a different number of actors, as shown in Table I.

Iterations	1	2	3	4	5	6
nb_{user}	10	3	6	8	10	2
$max\ nb_{CB}\ per\ user$	5	4	3	1	2	5
$max\ nb_{CB}$	14	10	13	5	8	14
nb_{actors}	39	31	43	27	30	36

TABLE I
ITERATION PARAMETERS

Figure 6 shows the system performance in terms of execution time, against the number of cores. The iterations correspond to the ones presented in Table I. These curves confirm that the system performances improve as the number of cores increases. However, the concurrent shared memory accesses also increase and become a bottleneck. This is a limitation of the current system. The performance improvement naturally depends on the shape of the srDAG because the exposed parallelism depends on the graph parameters.

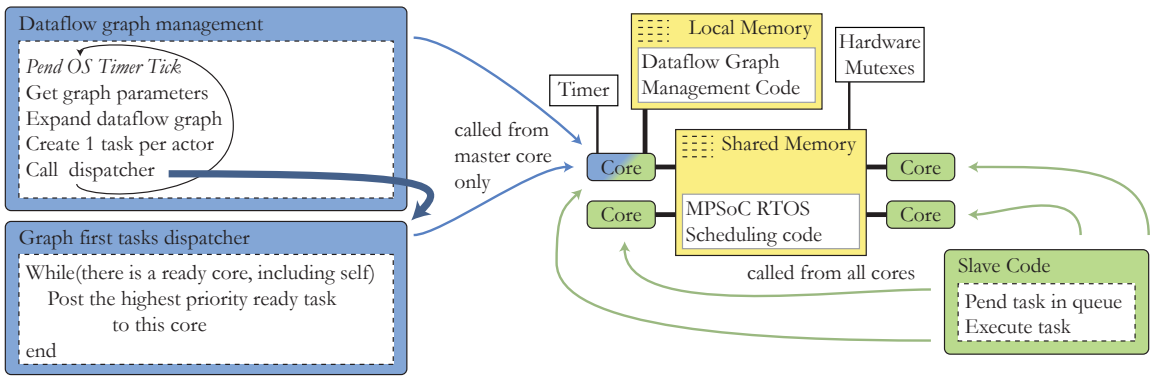

Fig. 3. Master/Slave Scheduling Phase: Expanding Dataflow Graph and Launching Slave Cores.

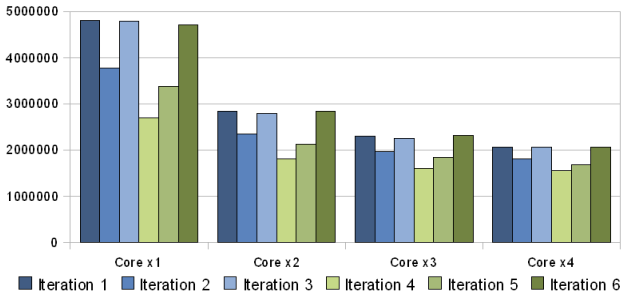

Fig. 6. Execution time performances

VII. Future Works

The current prototype may be improved both in terms of latency and in terms of memory. Speed can be gained by exploiting quiescent points [11] of the single rate DAG. Quiescent points are points between two actors' execution when no context needs to be maintained during rescheduling. This case corresponds to the scheduler call at the end of the "Task execution code" in Figure 4. This call does not require context switch and the task stack can even be flushed.

From a memory viewpoint, the task stacks can be significantly reduced. μC/OS-II associates independent stacks to each task. Having a pool of tasks and reusing tasks associated to finished actors will automatically reduce task creation time and stack memory. It is planned to test the code on a Texas Instruments TMS320TCI6486, a 6-core SMP DSP with a 500MHz clock. Each core of this platform also has a private memory. Thus, application data and code can be removed from the shared memory and managed automatically using dataflow graph information. The TMS320TCI6486 enables research towards a RTOS for non-SMP MPSoC architectures.

VIII. Conclusion

In this paper, we detailed an RTOS for Symmetric Multiprocessing MPSoC combining ideas from dataflow models and from commonly-used RTOS. The advantages of such a system are numerous. Dataflow graphs bring automatic parallelization and ease of application description. The traditional difficulty of manually synchronizing tasks disappears. Moreover, the actors themselves can be written in C or C++ code, so legacy code can be easily reused. The system obtained is portable to SMP architecture with considerable number of cores.

The MPSoC RTOS obtained goes successively through a master/slave phase that parameterizes the dataflow graph and a symmetric phase that permits any core to access the RTOS scheduler. This allows the reactivity to events of an RTOS to be combined with the automatic partitioning and portable execution of an algorithm described in a parameterized dataflow graph. Experimental results are shown on a 3GPP Long Term Evolution algorithm running on a 4-core MPSoC. The MPSoC RTOS prototype is built on the μC/OS-II RTOS.

Based on the present prototype of the MPSoC RTOS, further research will be conducted to identify the dataflow models best suited to signal processing applications. Additional scheduling methods and architectures will also be tested to exploit a prior knowledge on graph execution.

References

[1] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, p. 33–42, 2006.
[2] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, p. 773–801, 1995.
[3] W. B. Ackerman, "Data flow languages," *Computer*, vol. 2, 1982.
[4] V. Nollet, D. Verkest, and H. Corporaal, "A safari through the MPSoC Run-Time management jungle," *Journal of Signal Processing Systems*, vol. 60, no. 2, p. 251–268, 2010.
[5] W. Thies, M. Karczmarek, M. Gordon, D. Maze, J. Wong, H. Hoffmann, M. Brown, and S. Amarasinghe, "StreamIt: a compiler for streaming applications," *Technical Report MIT , Cambridge, MA*, 2001.
[6] J. J. Labrosse, *MicroC/OS-II: the real-time kernel*. Newnes, 2002.
[7] B. Bhattacharya and S. Bhattacharyya, "Consistency analysis of reconfigurable dataflow specifications," in *Embedded processor design challenges*, 2002, p. 308–311.
[8] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-static data flow," in *icassp*, 1995, p. 3255–3258.
[9] M. Pelcat, J. F. Nezan, and S. Aridhi, "Adaptive multicore scheduling for the LTE uplink," in *Adaptive Hardware and Systems (AHS), 2010 NASA/ESA Conference on*, 2010, p. 36–43.
[10] Altera, "Literature on NIOSII software embedded processor," http://www.altera.com/devices/processor/nios2/ni2-index.html.
[11] S. Neuendorffer and E. Lee, "Hierarchical reconfiguration of dataflow models," in *Formal Methods and Models for Co-Design, 2004.*, 2004.