A Hierarchical LST-based Task Scheduler for NoC-based MPSoCs with Slack-Time Monitoring Support

Marcelo Ruaro[1,2], Guilherme Madalozzo[1], Fernando G. Moraes[1]
[1]PUCRS University, Computer Science Department, Porto Alegre, Brazil
[2]SETREM, Computer Science Department, Três de Maio, Brazil
marceloruaro@setrem.com.br, guilherme.madalozzo@acad.pucrs.br, fernando.moraes@pucrs.br

***Abstract*—Emerging large-scale MPSoCs can have hundreds of PEs (Processing Elements), and scalable real-time support is necessary. Current proposals in MPSoCs scheduling have static behaviors or lack accurate validation, from a clock cycle model of the system. This paper proposes a hierarchical scheduling algorithm. At the bottom level, each PE executes a local LST-based scheduler algorithm with extended features to handle with inter-task communication and interruption overheads. At the top level, a global scheduler manages at run-time task mapping and real-time adaptation by using task migration and monitored information. The run-time adaptation is supported by a slack time monitoring that notifies the global scheduler the slack time of the PEs. The monitoring data, combined with traditional real-time metrics, provide a powerful real-time management that, as demonstrated by a clock cycle validation, can be implemented in large scale MPSoCs.**

Keywords—MPSoC; Real-time; Scheduling; Slack time; Monitoring.

I. INTRODUCTION

The increasing number of PEs (Processing Elements) in modern MPSoCs (Multiprocessor System on Chip) increases the resource sharing among the system components. For this reason, scheduler algorithms are fundamental to manage the processors' usage while satisfies RT (real-time) constraints of applications.

Differently from uniprocessor systems, a scheduling algorithm for multiprocessor system rises in complexity [1] because the scheduling problem is expanded from '*when'* to '*when and where'* execute a given task. According to the state-of-the-art section, several proposals have addressed scheduling algorithm for multiprocessor systems. However, the proportion of these targeting MPSoC remains low and still have limitations. Besides the multiprocessor scheduling be a more complex problem, MPSoCs systems increase this complexity due its limited resources.

This paper proposes an original real-time scheduling algorithm for NoC-based MPSoC, called *HQoS*. The contributions of the proposed scheduling algorithm can be summarized as follows:

Clustered and Hybrid-based Scheduling organization: the MPSoC is organized in clusters. Each cluster contains one manager processor and a set of slave processors, providing scalability [2]. The hybrid term is related to the traditional scheduling classification [1]. The algorithm has a global part, implemented at each cluster manager, and a partitioned part implemented at each slave processor.

LST (Least Slack Time) based scheduling: HQoS is based on LST algorithm [3]. The novelty is to use LST for MPSoCs, and assuming task preemption, task migration, and inter-task dependency.

Support to periodic, sporadic and aperiodic tasks: HQoS is a fully adaptive run-time scheduling algorithm. It can support variable task periods, soft deadlines and execution times. To enable this support, a dynamic RT constraint method is proposed, which enable the application's developer to program its RT task constraints to change, on demand, its RT constraints according to the task workload variation.

***Hierarchical slack time processor's monitoring*:** as HQoS uses the LST locally at each slave processor, the processor's slack-time (or idle times) can be measured and send to each cluster manager. This information is a rich metric to enable others run-time adaptation, as DVFS, clock gating, and employ novel concepts as the core activation patterns [4].

***Extended scheduler timer*:** HQoS is a priority-driven scheduler. Instead using a constant scheduler timer, as in most of the traditional schedulers [3], it dynamically computes the next scheduler call time based on the RT task status and its slack time. This approach enables to reduce the scheduler execution overhead.

II. STATE-OF-THE-ART

The literature related to task scheduling is rich, with a large number of proposals focusing on multiprocessor systems. However, few works address scheduling for MPSoCs.

Pfair is a state-of-the-art hard real-time scheduler for multiprocessor systems [5]. Park et al. [5] propose HPGP, a Hybrid scheduler for MPSoCs based on *Pfair*. A partitioned version runs the *Pfair* algorithm. The proposal considers only periodic task with deadlines equal to their periods and constant execution times. Besides, the evaluation is carried out only over two cores, what not enable to validate the algorithm in large MPSoCs.

Tafesse et al. [6] propose two scheduling algorithms for MPSoCs: Performance Driven Scheduling (PDS), for bus-based MPSoCs, and Traffic Aware Scheduling (TAS), for NoC-based MPSoCs. The PDS algorithm is based on the Simulated Annealing (SA) technique and searches for the best RT task mapping considering temperature thresholds. The TAS algorithm is a design-time algorithm that performs the task scheduling and mapping also using the SA technique: the tasks are randomly mapped into PEs until they achieve a temperature threshold without violating task deadlines.

The LST algorithm is proved non-optimal to multiprocessor systems [3]. Hwang et al. [7] propose the LSTR, a scheduling algorithm based on LST with additional features to be optimal for multiprocessor systems. The Authors enable task migration and preemption. The LSTR was designed to support only periodic tasks, do not considering task migration and task preemption overheads and requires a high frequency of task migration. Those features are not adequate to generic purpose MPSoCs, where applications are downloaded on-the-fly and can change its real-time constraints at run-time according to external inputs for example.

To the best of Authors knowledge, this paper is the first proposal that addresses a scheduling algorithm for MPSoC with the originalities detailed in the Introduction combined with an evaluation employing a clock cycle accurate platform model.

III. System Model

Section III.A describes the MPSoC Model. Section III.B describes the task model, a contribution of this work.

A. MPSoC Model

Fig. 1 presents the distributed memory MPSoC overview. An MPSoC consists of a set of homogeneous PEs interconnected by a given network topology. This work adopts a cluster-based MPSoC, with manager and slave PEs. A Cluster Manager (CM) PE does not execute user tasks. It executes a management µkernel and the top-level part of the HQoS algorithm. A specialized version of CM called Global Manager also implements the access to an application repository that simulates the insertion of a new application. The Slave PE executes user tasks. It runs a µkernel with the bottom level part of the HQoS scheduling algorithm.

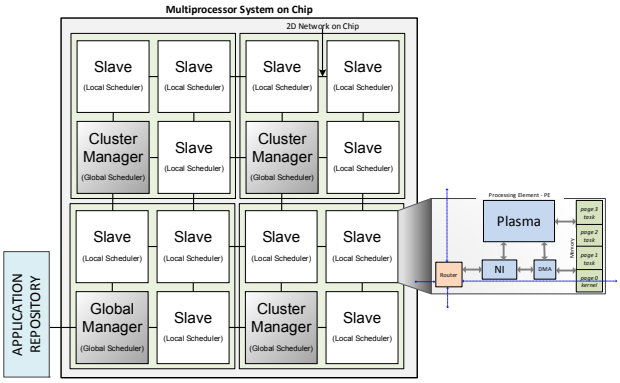

Fig. 1 - Cluster-based MPSoC architecture, with HQoS Scheduler.

Processors may handle two external interruptions: NoC interruption: signalizing a received packet from the NoC; Scheduler interruption: signalizing the end of scheduler timer. The manager PEs handle only NoC interruption. Slave PEs handle both interruptions. In the occurrence of an interruption, the executing task is preempted, and the µkernel handles the interruption.

B. Task Model

A directed task graph models the applications. Task communicate using *Send* and *Receive* MPI-based primitives. Send is non-blocking, and Receive is blocking. A given task can be at four different states: *waiting*, *ready*, *running*, and *sleeping*. Waiting state mean that the task is blocked, waiting for a producer task to send it a message. Ready state means that task already achieved its release time, and it is ready to be scheduled. Running state mean that the task is executing in the CPU, only one task can stay in running state. Sleeping state (for RT tasks only) means that task already achieved its execution time and its period does not end yet, so the task must be suspended.

The system supports two task types: *best-effort* (BE) and *real-time* (RT). BE tasks have not time bound and explore the slack time of RT tasks executions. RT task has temporal requirements. Fig. 2 details the RT task constraints model. The time is computed in clock cycles.

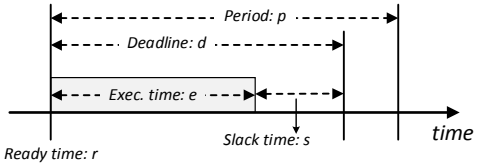

Fig. 2 – Real-time constraints model.

A given RT task has four real-time constraints: (*i*) period, (*ii*) deadline, (*iii*) execution time, (*iv*) utilization. The period is the time between the task's release times. The execution time is the amount of time that task must execute at each period. The deadline mean the time that task have to complete it execution time, for simplicity, this work handle with absolute deadlines [3]. The utilization is the percentage that task uses the processor. It is computed by the following equation: *utilization = (execution time * 100)/period.*

To make the system aware of an RT task, the task must call a system call (*syscall*) named *RealTime*. Fig. 3 presents an example of an RT task code that configures RT constraints dynamically. At line 3 the *taskA* calls the *RealTime* syscall, notifying to the µkernel its RT constraints. At lines 4-8 the task executes some code. Due previously RT constraints configuration, the scheduler can execute *taskA* according to the real-time loop requirements. Next, at line 10, the task changes its RT constraints due a new RT workload profile that will be executed in lines 11-15. The task then calls the *RealTime* syscall again to notify to the scheduler its new constraints. As observed, this system call enables the task to change its real-time constraints dynamically, characterizing a periodic or aperiodic behavior [3].

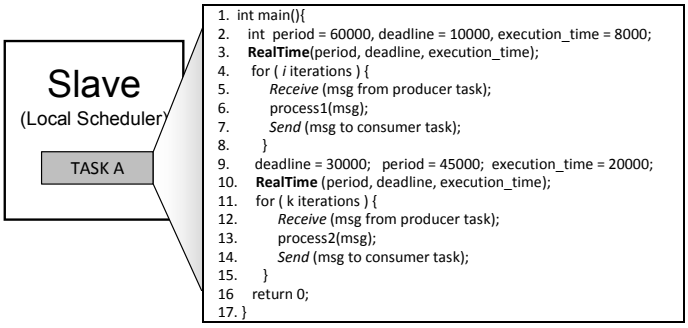

Fig. 3 - Example of a task code with run-time RT configuration. TaskA is an RT task with different constraints according its code workload. It calls 2 times the *RealTime* syscall to configure its constraints into the local scheduler (lines 3 and 10).

To handle the inter-task dependencies, this work assumes that each application's iteration fits into a hyper period, i.e., each task of the same application must configure a period that covers all its iteration periods. This hyper period includes the communication overhead observed for each application's iteration. Let take as an example the MPEG decoder in Fig. 4. Assuming in a worst case that each task is allocated into a different PE, it is possible to define the RT tasks behavior by observing the application iteration pattern. All tasks have the same period. However, each task has its individual execution time and deadline according to task dependency order. Task *input* is the first to execute at each iteration. Hence, its deadline must be the first to ends.

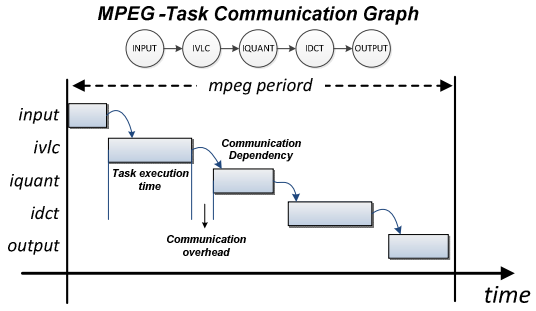

Fig. 4 – MPEG real-time constraints configuration.

IV. HQoS Scheduler

Table 1 describes the classification of the proposed scheduler according to the most adopted literature concepts [1][3][8]. The scheduler is hierarchical (Fig. 1), divided into Global Scheduler (GS), implemented at each manager PE, and Local Scheduler (LS), implemented at each slave PE.

Table 1 – Proposed HQoS scheduler classification.

Concept	Classification
Organization (Global, Partitioned)	Hybrid and clustered (Mixes Global and Partitioned)
Scheduling decision (Static, Dynamic)	Dynamic
Allocation (Clock, Table, Priority)	Priority-driven approach (LST-based)
Migration (Job level, Task level)	Task-level Migration
Processor Number (Uni., Multi.)	Multiprocessor (on chip)
Preemption (Yes, No)	Yes
Supported task	Periodic, sporadic, aperiodic
Real-time (Hard, Firm, Soft)	Soft real-time

A. Local Scheduler (LS)

Assuming the task model described in Section III.B, the partitioned LST scheduler executes at each slave PE as a traditional LST scheduler, or as some works define: assuming a partitioned behavior. In an LST scheduler, the RT tasks have scheduling priority over BE task. RT tasks are scheduled according to their least slack time. If there are two or more RT tasks with the same slack time, a Round-Robin algorithm is used to selects the next scheduled task. This same algorithm is used to schedule BE tasks if the processor does not have RT tasks or all RT tasks are sleeping or waiting.

The proposed LST-based scheduler must consider inter-task communication. As the system uses an MPI-based communication, some tasks can be in a waiting state. When this happen, BE tasks are blocked. However, when an RT task goes to the waiting state, the scheduler handles the RT task as a sleeping task, i.e., the scheduler verifies the end of task period, but not update its remaining execution time neither schedules the task. When the task received the requested message by a NoC interruption, the µkernel change the state of the task to ready and calls the LST scheduler. The scheduler then updates the slack time and the remaining execution time for all its RT ready and running tasks, scheduling the ready task with LST priority.

To enable a hierarchical scheduling support, the LS communicates with the GS. Three messages are defined: (*i*) *real-time request*, generated when a task calls the *RealTime* syscall. This message is responsible for informing the GS of the new RT task or RT parameters change; (*ii*) *deadline miss*, generated when an RT task executes beyond its deadline; (*iii*) *slack time report*, a message send periodically to update the GS's slave slack time, this message is detailed in IV.D.

B. Extended Scheduler Trigger Time

In the proposed *LS*, a hardware scheduling interruption timer is used, being dynamically programmed by using a memory mapped register. After the programming, it is decremented at each clock cycle until the configured value reaches 0. At this point, a scheduler interruption is generated to the µkernel.

When an RT task is scheduled, the appropriated scheduler timer is the RT task remaining execution time (RT_{ret}), however, other RT task can need a scheduler call before the end of the RT_{ret} with the purpose to avoid a deadline miss, caused by a lack of scheduler verification. For this reason, the scheduler needs to verify if there is any other RT task with a slack time lower than RT_{ret}. Further, the scheduling also needs to verify if there is any other RT task with the end of period lower than RT_{ret}. In the presence of a scheduling point lower than RT_{ret}, this point is selected as being the next scheduler trigger timer.

C. Global Scheduler

The *Global Scheduler* (GS) executes at each manager PE. The GS has two main functions: (*i*) schedule tasks into slave PEs (in fact it acts as a mapper); (*ii*) handle report messages from LS.

When a new application is requested to execute in the system, the global manager PE (Fig. 1) executes a heuristic to select the cluster to execute the application – *cluster selection*. The main goal of the *cluster selection* heuristic [9] is to distribute the workload evenly in the system. Once a cluster is selected, the *application description* is sent to a cluster manager (note that the global manager is also a cluster manager, and can receive the application description).

The GS starts when a new *application description* is received. To map the tasks of the new application the GS searches a slave PE (SP) according to the *GS_task_scheduling* heuristic presented in Fig. 5. The algorithm receives as input the cluster ID. Then, a set containing all slaves PEs in the cluster is created (lines 1-4). Next, a set of functions removes slaves PEs from the set, according to a group of criteria (lines 5-8). Finally (line 9), the first element of the set is returned, that makes the selected PE the PE that will receive the task. Fig. 5 uses as main information the slack time obtained by monitoring. The slack time implicitly carries the RT execution, BE execution, interruption handling overhead, scheduler overhead and switch context overhead.

```
GS_task_scheduling(cluster c_ID) {
1.    C_PE ← { }
2.    for all sp_i of c_ID do          // build the set C_PE will all SPs of the cluster
3.        C_PE ← C_PE U sp_i
4.    end for
5.    // leaves in the set C_PE the SPs with the highest average slack time
      C_PE ← avg_slack_time(C_PE)
6.    // leaves in the set C_PE the SPs with the smaller number of allocated RT tasks
      C_PE ← min_rt_task(C_PE)
7.    // leaves in the set C_PE the SPs with the highest absolute slack time
      C_PE ← abs_slack_time(C_PE)
8.    // leaves in the set C_PE the SPs with the smaller number of allocated BE task
      C_PE ← allocated_tasks(C_PE)
9.    return( first(C_PE) )            // return the first element in the set C_PE
10. }
```

Fig. 5 - Global Scheduler Heuristic to select a SP to execute a task.

After scheduling the task into the selected SP, the task is locally scheduled by the LS. During system execution, the GS handles messages sent from the LSs. For *slack time report* messages, the GS update the absolute PE slack time and average slack time. For *real-time request* messages, the GS verifies if it does not exceed 100% of the processor utilization. If this is the case, the GS start a real-time adaptation. For *deadline miss* messages, the GS directly starts a real-time adaptation.

The real-time adaptation is a heuristic that uses task migration to select an SP to receive the RT task with deadline misses. To select the SP to receive the task, a similar heuristic to Fig. 5 is used. The difference is the first criterion to remove elements in C_{PE}: *utilization*. All SPs that remain in the set C_{PE} must have a remaining utilization higher or equal than the task utilization.

Using such heuristics, the GS take advantage of monitored slack time of its slave processors, together with traditional RT metrics. Such information provides a tradeoff between processor's real-time utilization and load balancing.

D. Slack time monitoring

Slack time monitoring is implemented using a hardware timer, which at each 10 ms triggers an interruption to the SP µkernel. This interruption is handled by sending a simple message to the GS with the processor's slack time in the last 10ms.

This window monitoring is configurable at design-time and can be adjusted to provide a tradeoff between NoC communication load and slack time update frequency.

V. RESULTS

Results were obtained using an RTL SystemC model of the MPSoC (9x9, with 3x3 clusters) presented in Section III.A. The µkernel, as well as the GS and LS schedulers, are implemented in C language. The hardware implementation, specific for this proposal, corresponds to the scheduler timer counter, implemented in the SPs.

A. *Dynamic Real-Time Constraints Evaluation*

This experiment uses synthetic RT tasks. All SPs received initially an RT task consuming 40% of the processor utilization. At a given moment, a second RT task (RT_2) is allocated in one SP. RT_2 has the behavior of the task code in Fig. 3, with two calls to the *RealTime* syscall. The first one imposes 55% of processor utilization, and the second one 90%. Fig. 6 presents the latency results for each iteration of RT_2. Three scenarios are evaluated: (*i*) optimal, with the target task running alone into the system; (*ii*) RT support; (*iii*) without RT support.

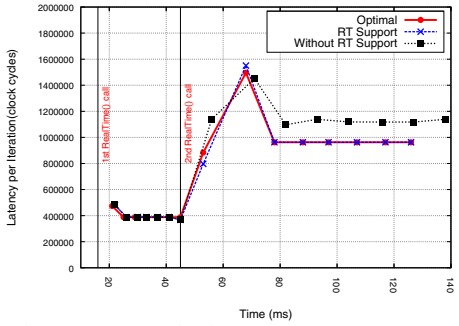

Fig. 6 - Dynamic RT support evaluation.

Note in Fig. 6 that the three scenarios have the same behavior up to the second vertical line. This vertical line corresponds to the second *ReakTime* call, which request more RT guarantees. This new behavior triggers a real-time adaptation (represented by the RT support scenario). After the real-time adaptation the scenario with RT support achieves exactly the same latency of the optimal scenario. This result is explained due the task migration of target task (RT_2) to a free processor. In this evaluation, it is also possible to observe the negligible impact of task migration over the target task latency, with a total task migration overhead of 10,343 clock cycles.

B. *Real-time Applications Evaluation*

In this scenario, two real applications are used: MPEG decoder, communication intensive, and Dynamic Time Warping (DTW), computation intensive. The DTW application starts its execution at the beginning of the simulation, the MPEG start at 15ms, and 3 BE applications are inserted at 25ms, 35ms, and 45ms.

The graph of Fig. 7 represents the latency for each MPEG iteration (frame decoding). This latency is collected at the last MPEG task (*output*). As can be observed, the RT support does not achieve the latency of the optimal scenario. This result is discussed later. In terms of average latency, the RT support increases the latency 45.8% in comparison to the optimal scenario. On the other hand, the RT support scenario provides a reduction of 209.3% compared with the scenario without RT support, demonstrating the high impact imposed by BE application in this experiment.

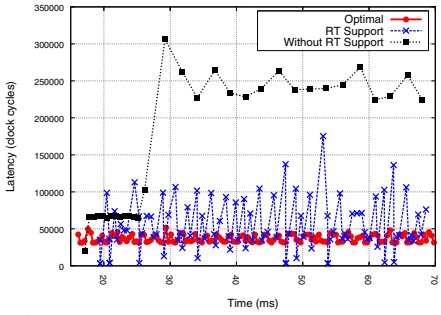

Fig. 7 - MPEG latency.

The graph presented in Fig. 8 addresses the DTW iteration latency. The effect of the RT support for DTW application is easily observed. The scenario with RT support can maintain the application requirements even with the load of the BE applications. The average latency increases only 1.43% comparing the RT support against the optimal scenario. The average latency reduction comparing the RT support against without RT is 97.7%.

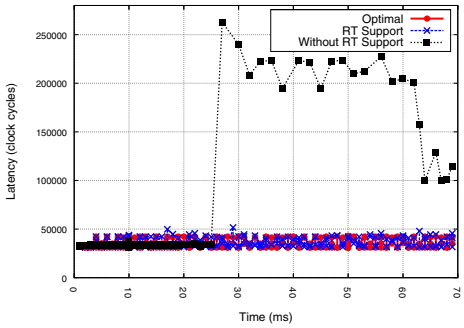

Fig. 8 – DTW iteration latency.

As shown in Figures 7 and 8, the proposed scheduler achieved a significant reduction in the latency of the benchmarks. The high impact of disturbing over the MPEG application occurs because this application is highly communicating, which allows that the disturbing caused by BE applications can be evidenced. In the case of DTW, that is computation intensive, the scheduler can prevent deadline misses even in a high computation disturbing. As the scope of the scheduler is, essentially, computation resource management, it is expected the behavior shown in both graphs. This analysis emphasizes the need to complement a real-time scheduler with a run-time communication adaptation mechanism, as proposed in [10].

VI. CONCLUSION

Most of the scheduling literature for multiprocessor is essentially theoretical or inadequate for MPSoCs. In this work, we proposed and implemented a scheduler in an MPSoC modeled at the RTL level, targeting large-scale systems. The scheduler is fully adaptive, supporting dynamic task RT constraints with slack time monitoring at the processor level. Future works include exploring the mapping heuristics and slack time monitoring results.

ACKNOWLEDGMENTS

The Author Fernando Moraes is supported by CNPq - projects 472126/2013-0 and 302625/2012-7, and FAPERGS - project 2242-2551/14-8.

REFERENCES

[1] Davis, R.I.; Burns, A. “A survey of hard real-time scheduling for multiprocessor systems”. *ACM Comput. Surv.* Article 35, 2011, 44 p.

[2] Castilhos, G.; Mandelli, M.; Madalozzo, G.; Moraes, F.G. “Distributed Resource Management in NoC-Based MPSoCs with Dynamic Cluster Sizes”. In: ISVLSI, 2013, pp. 153-158.

[3] Liu, J.W.S. “Real-Time System”. Prentice Hall, New Jersey, 2000.

[4] Shafique, M.; Garg, S.; Henkel, J.; Marculescu, D. “The EDA challenges in the dark silicon era”. In: DAC, 2014, pp.1-6.

[5] Park, S. “Task-I/O Co-scheduling for Pfair Real-Time Scheduler in Embedded Multi-core Systems”. In: EUC, 2014, pp.46-51.

[6] Tafesse, B.; Raina, A.; Suseela, J.; Muthukumar, V., “Efficient Scheduling Algorithms for MPSoC Systems”. In: ITNG, 2011, pp. 683-688.

[7] Hwang, M.; Choi, D.; Kim, P. “Least Slack Time Rate First: New Scheduling Algorithm for Multi-Processor Environment”. In: CISIS, 2010, pp.806-811.

[8] Mall, R. “Real-Time Systems: Theory and Practice”. Pearson; 1 edition, 2006.

[9] Mandelli, M.; Osts, L.; Sassatelli, G.; Moraes, F.G. “Trading-off System Load and Communication in Mapping Heuristics for Improving NoC-Based MPSoCs Reliability”. In: ICECS, 2012, pp. 544-547.

[10] Ruaro, M.; Carara, E.A.; Moraes, F.G. “Runtime Adaptive Circuit-Switching and Flow Priority in NoC-Based MPSoCs ”. IEEE Transactions on VLSI, v.23(6), pp. 1077-1088, June 2015.