Dynamic Real-Time Scheduler for Large-Scale MPSoCs

Marcelo Ruaro, Fernando G. Moraes
PUCRS University, Computer Science Department, Porto Alegre, Brazil
marcelo.ruaro@acad.pucrs.br, fernando.moraes@pucrs.br

ABSTRACT

Large-scale MPSoCs requires a scalable and dynamic real-time (RT) task scheduler, able to handle non-deterministic computational behaviors. Current proposals for MPSoCs have limitations, as lack of scalability, complex static steps, validation with abstract models, or are not flexible to enable changes at runtime of the RT constraints. This work proposes a hierarchical task scheduler with monitoring features. The scheduler is dynamic, supporting changes in RT constraints at runtime. An API enables these features allowing to the application developer to reconfigure the tasks' period, deadline, and execution time by annotating the task code. At runtime, according to the task execution, the scheduler handles the API calls and adjust itself to ensure RT guarantees according to the new constraints. Scalability is ensured by dividing the scheduler into two hierarchical levels: LS (Local Schedulers), and CS (Cluster Schedulers). The LS runs at the processor level, using the LST (Least Slack-Time) algorithm. The CS runs at the cluster level, i.e., a group of processors controlled by a manager processor. The CS receives messages from the LSs, informing the processor slack-time, deadline violations, and RT changes. The CS implements an RT adaptation heuristic, triggering task migrations according to RT reconfiguration or deadline misses. Results show a negligible overhead in the applications' execution time and the fulfillment of the applications' RT constraints even with a high degree of resources sharing, in both processors and NoC.

Keywords

MPSoC; Real-time; Scheduler; Runtime; Slack-time.

1. INTRODUCTION

The increasing number of processing elements (PEs) in modern large-scale multiprocessor systems on chip (MPSoCs) increases the resource sharing among system components [1]. For this reason, scheduling algorithms are essential for managing the processors usage while satisfying the constraints of real-time applications.

A key feature of a complex system, such as an MPSoCs, is the ability to support dynamic workloads. Applications may have moments of heavy computational load and also can have moments of a state close to the idle, waiting, for example, an external input, as a user interaction or a message from another task. For this reason, it is necessary to allow applications to tune the computational workload, avoiding unnecessary resources allocation. Aware of this challenge, this work focuses on a self-adaption technique for MPSoCs, proposing a dynamic RT task scheduler that can support runtime reconfiguration of the tasks' RT constraints. This reconfiguration starts with an API that enables the application developer to characterize the RT workload of each task at different execution points. At runtime, according to the task execution, the API triggers changes in the task RT constraints. The proposed task scheduler handles these changes at runtime to fulfill the new tasks' RT constraints.

Scalability is ensured by dividing the scheduler into two hierarchical levels: LS (*Local Schedulers*), and CS (*Cluster Schedulers*). The LS runs at the processor level, using the LST (Least Slack-Time) algorithm. The CS runs at the cluster level, i.e., a group of processors controlled by a manager processor. The CS receives messages from the LSs, informing the processors' slack-time, deadline violations, and RT changes. The CS implements an RT adaptation heuristic, triggering task migrations according to RT reconfiguration or deadline misses.

The scheduler works as closed-loop control system - Figure 1. The monitoring and notification messages (processor slack-time, deadline miss, RT changes), produced by the LSs, are the inputs for an RT manager executed in the CS, which can trigger an adaptation by generating a task migration action. This process is repeated along the execution of the applications.

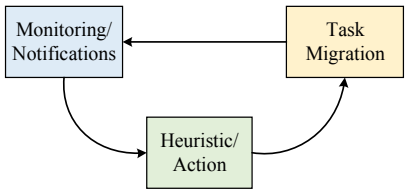


Figure 1. Scheduler support for self-adaptation at runtime.

The *focus of this work* is related to the scheduler ability to support dynamic RT reconfigurations while it satisfies the task RT constraints. Task mapping or schedulability analysis in out of this work scope. Such techniques are broadly explored in the literature [2][3][4] and can be easily combined with the proposed scheduler. Aware of the current state-of-the-art works, this work is the first to support a runtime reconfiguration of the RT task constraints. Besides, the evaluation of the proposed scheduler is executed in a clock cycle accurate description of the MPSoC.

2. RELATED WORK

The literature related to task scheduling contains a large number of proposals focusing on multiprocessor systems [1]. However, most of the works are not suitable for MPSoCs. Most MPSoC schedulers consider design-time steps integrated into frameworks [5][6][7][8]. Such works often employ a MoC (Model of Computation) such as PPN, DAG, and SDGA to model the applications at design-time, making its behavior predictable, and also enabling hard RT scheduling. Static or partial static scheduling is a conservative approach to guarantee hard RT behavior. Those proposals are only effective when the set of applications to execute in the system is fixed at design-time.

Pfair is a state-of-the-art hard RT scheduler for multiprocessor systems [4]. Park et al. [4] propose HPGP, a hybrid scheduler for MPSoCs based on *Pfair*. A partitioned version runs the *Pfair*

GLSVLSI '16, May 18-20, 2016, Boston, MA, USA
© 2016 ACM. ISBN 978-1-4503-4274-2/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2902961.2903027

algorithm together with a global scheduler that makes task mapping and task migrations. The manager selects the ready tasks to be scheduled into a PE. Each PE executes a schedulability analysis. If the task is not schedulable, the core invokes the global scheduler to perform task migration. The task mapping approach is not addressed. The proposal considers only periodic tasks with deadlines equal to their respective periods and constant execution time. Besides, the evaluation is carried out only with four cores, which does not enable validating the algorithm in large MPSoCs. This work also uses a hybrid scheduling organization. Differently from *Pfair*, in this work the scheduler supports variable tasks periods, deadlines and execution times, providing higher flexibility for RT workloads.

The LST (*Least Slack Time*) scheduler was proved to be non-optimal to multiprocessor systems [9]. Hwang et al. [10] propose LSTR, a scheduling algorithm based on LST, with additional features to be optimal for multiprocessor systems. LSTR was designed to support only periodic tasks. Besides a large number of task migrations, the Authors do not consider such overhead.

The design of an RT scheduler for MPSoCs should consider, among other factors, how to inform the RT task constraints to the system, and how to handle interruptions. As the scheduler proposed in this work is a dynamic scheduler, RT task constraints are transferred to the operating system (OS) using task code annotation, a common approach found in the literature. Theodoropoulos et al. [11] use task code annotation (called pragmas) that are used by a runtime manager to perform task mapping. Canella et al. [12] employ task annotation to implement a task migration mechanism based on task replication. The task annotation is used to guarantee inter-task communication synchronization.

The treatment of interruptions may interfere in the execution of RT tasks. Interruptions can be handled immediately using specific system routines, or can be pooled at fixed and predetermined times [13]. Another alternative is to redirect interruptions handling to free cores [4]. However, this option can only be useful when interruptions come from external devices and are not related to inter-task communications.

Concluding, there is a lack of works in the literature addressing dynamic RT schedulers for MPSoCs. To the best of the Authors knowledge, this paper is the first proposal that addresses a scheduling algorithm for MPSoC combining the originalities detailed in the Introduction, combined with an evaluation employing a clock cycle accurate platform model.

3. SYSTEM MODEL

This section describes the system model, detailing important assumption related to this work.

3.1 MPSoC Model

This work adopts the cluster-based architecture, with manager and slave PEs depicted in Figure 2(a). The MPSoC contains a set of PEs, interconnected by a NoC. It is partitioned into n clusters $C=\{c_1, c_2, \ldots, c_n\}$. Each cluster $c_i \in C$ has a set $P(c_i)=\{M, sp_1, sp_2, \ldots, sp_k\}$, where M denotes a CM, k the number of slave processors in the cluster. Only one CM has an interface to input/output devices. Note that the cluster size is not static. At runtime, it can change it size by using a reclustering protocol [14].

The platform uses a distributed memory organization. Inside each PE - Figure 2(b), there is one processor, one dual-port local memory, one NI (Network Interface), NoC router, and one DMA module (Direct Memory Access). The adoption of a DMA module improves the processor performance avoiding stalls to send and receive packets to/from the NoC.

All PEs execute a small operating system (µkernel: ~20KB). The CMs execute a management µkernel, not executing user tasks. One specific CM has access to an *application repository*. This repository contains the applications' tasks object code, and simulates the insertion of new applications into the MPSoC at runtime. SPs execute a µkernel to control the execution of the user tasks. This µkernel also implements a paging mechanism to support multitasking and an inter-task communication protocol based on MPI.

The PEs handle two external interruptions: *NI interruption*, signaling a received packet from the NoC; *scheduler interruptions*, signaling the end of the scheduler timer (only in SPs). In the occurrence of an interruption, the executing task is preempted.

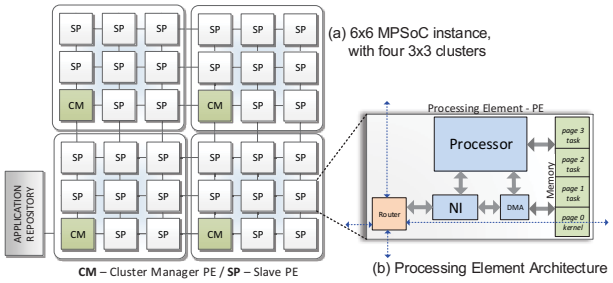


Figure 2. Cluster-based MPSoC example - a 6x6 MPSoC with four 3x3 clusters. CM: cluster manager. SP: slave processor.

3.2 Application and Task Model

An m-task application $A=\{t_1, t_2, \ldots, t_m\}$ is modeled by a task graph $G(T, E)$, with each vertex $t_i \in T$ representing a task and the directed edge (e_i, e_j), denoted as $e_{ij} \in E$, representing the communication between tasks t_i and t_j. Tasks communicate using a non-blocking *Send* and blocking *Receive* MPI-like primitives. A given task t_i can assume four states: *waiting*, *ready*, *running*, and *sleeping*. The *waiting* state implies that the task is blocked, waiting for a producer task to send it a message. The *ready* state means that the task already achieved its release time, and is ready to be scheduled. The *running* state implies that the task is executing on the processor. The *sleeping* state (RT tasks only) means that the task already finished its execution time and its period does not end yet, so the task must be suspended.

The system supports *best-effort* tasks - B_t, and *real-time* tasks - R_t. B_ts do not have time bounds and explore the slack-time of R_ts. R_ts have soft temporal requirements. Figure 3 details the R_t constraints model.

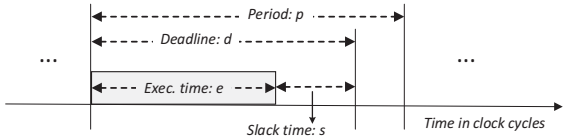


Figure 3. RT constraints model.

***R_t* definition**: R_t is a 6-tuple $\{p, d, e, u, r, s\}$ with a period (p), relative deadline [9] (d), execution time (e), utilization (u), remaining execution time (r), and slack-time (s). Utilization corresponds to u = e * 100 / p. The remaining execution time is the amount of time R_t has to finish it execution time.

To make the system aware of the constraints of a given R_t, the task must execute a system call (*syscall*) named *RealTime*. This *RealTime* syscall is the API provided to the application developer to reconfigure the RT constraints. Figure 4 presents an example of an R_t code that configures the RT constraints dynamically. At line 3, TASK A calls *RealTime*, notifying the µkernel of its RT constraints. Lines 4-8 execute some code. Due to the RT constraints configuration, the scheduler can execute TASK A

according to the RT requirements. Next, at line 10 the task calls *RealTime* again to notify the scheduler about its new constraints. The code between lines 11-15 executes according to these new constraints. As can be observed, this system call enables the task to change its RT constraints dynamically.

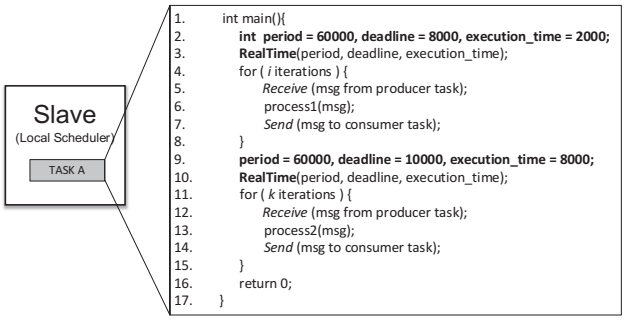


Figure 4. Example of a task code with runtime RT configuration. It calls the RealTime syscall twice to configure the constraints (in lines 3 and 10).

Note that the RT reconfiguration can be called in any task code point allowing the task can change its period (characterizing an aperiodic behavior), deadline, and/or execution time. Such behavior, where an RT task can change its RT constraints at runtime, is typical in real scenarios. For example, consider a voice recognition application. The application can assume two workloads: the first one is a listening state, where the application is waiting for the user to pronounce some sound, this state requires a moderate RT workload. The second one is the recognition state, where the task uses voice recognition algorithms. In this case, the application tasks can configure two workloads, with different RT requirements.

To handle inter-task dependencies, this work assumes that an iteration of a given application defines a *hyper period*, i.e., an RT application has all its tasks configured with the same *p*. This hyper period can handle with inter-task dependencies because it is composed of the WCET (worst-case execution time) of all tasks and the worst-case of communication between the application's tasks.

This task dependency model does not restrain the application model, accepting sequential, parallel and pipelined applications. The designer can obtain the WCET and communication latencies running the application alone in the system, representing a profiling phase.

4. THE PROPOSED SCHEDULER

Table 1 presents the classification [9][15] of the proposed scheduler.

Table 1 – Proposed scheduler classification.

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

Figure 5 presents an overview of the scheduler. It has a hierarchical organization that is divided into two levels:

- *Cluster scheduler* (CS) - runs the high-level part of the scheduler in the CMs. Its job is to perform RT adaptations if a given RT task is missing deadlines or it has changed its RT constraints and its current processor is not able to execute the task;
- *Local scheduler* (LS) - runs the low-level part of the scheduler in the SPs, executing the LST scheduling algorithm.

The LSs send messages to CS. The messages are (*i*) slack-time monitoring; (*ii*) deadline miss; (*iii*) RT change. Messages (*ii)* and (*iii)* are reactive messages, sent when a task misses a deadline, or when a task calls the *RealTime* syscall, respectively. The slack-time messages are generated periodically (slack-time monitoring – STM).

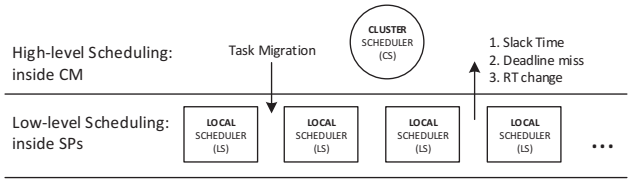


Figure 5. Hierarchical scheduler organization.

The STM provides to the CS the actual slack-time of each SP. The STM has a hardware/software implementation at each SP. The hardware part corresponds to a timer, which generates an interruption to the µkernel according to a monitoring window. This monitoring window is configurable at design time and can be adjusted to provide a tradeoff between NoC communication load and the STM update frequency. The software part corresponds to a slack-time counter and a monitoring interruption handler function, both implemented in the µkernel.

When the STM timer interrupts the µkernel, a message with the current slack-time is sent to the CS and the slack-time counter is reinitialized.

4.1 Cluster Scheduler (CS)

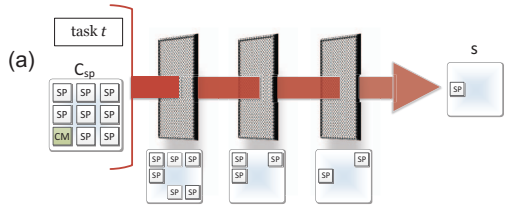
The CS executes into the CMs. The CS have the goal of handling the messages sent by the SPs and execute the RT adaptation if necessary. For the *slack-time* messages, the CS only updates the percentage of the idle state of each processor of its cluster. For *deadline miss* messages the CS executes a heuristic called *RT_adaptation*, which can select a new processor to migrate the penalized task. Finally, for *RT change* messages, the CS verify if the current processor of the task has enough utilization to execute the task, if not, as well as occurs for *deadline miss* message, the CS executes the *RT_adaptation* heuristic.

The *RT_adaptation* heuristic works as a set of decision layers applied to the SPs of the cluster - Figure 6(a). Figure 6(b) presents the heuristic pseudo-code. As input it receives the set $C_{SP} = \{P(c_i) - M\}$ (corresponding to set of SPs in the cluster), and the task *t* to be migrated into a given element of C_{SP}. As output the algorithm returns the selected processor *s*, corresponding to the new processor to receive the task *t*. If all the processors are not available to receive the task, the adaptation process is suspended, the affected task will start to miss deadlines triggering the adaptation process until an available processor can be found.

The decision functions only select the SP(s) which fulfill the function's requirement. The following decision functions are used:

- *utilization*: selects the SP(s) that have a remaining utilization enough to receive the task *t*;
- *max_avg_ST*: selects the SP(s) with the largest average slack-time, information obtained from STM;
- *min_RT_task:* selects the SP(s) with the minimum number of RT tasks allocated to it;
- *min_abs_ST:* selects the SP(s) with the largest absolute slack-time measured at the last slack-time monitoring window;
- *min_alloc_tasks:* selects the SP(s) with the minimum number of allocated tasks.

Finally, in line 6, the first SP in C_{SP} is selected to receive task *t*. Using this heuristic, the CS takes advantage of the monitored slack-time of its slave processors (lines 2 and 4), together with traditional RT metrics (line 1 and 3). The information provides a trade-off between the processor RT utilization and load balancing. After the execution of the heuristic, a task migration order in sent to the current SP of task *t* and the task is migrated to the selected processor *s*. The task migration protocol is out of this work scope.



(b) ***RT_Adaptation*** **(C_{sp}, *t*, *s*)**
Input: C_{sp}, t
Output: s
begin
1. $C_{sp} = C_{sp} \cap$ *utilization*(C_{sp}, t)
2. $C_{sp} = C_{sp} \cap$ *max_avg_ST*(C_{sp})
3. $C_{sp} = C_{sp} \cap$ *min_RT_task*(C_{sp})
4. $C_{sp} = C_{sp} \cap$ *min_abs_ST*(C_{sp})
5. $C_{sp} = C_{sp} \cap$ *min_alloc_tasks*(C_{sp})
6. s = *first*(C_{sp})
end

Figure 6. (a) layered decision flow. (b) *RT_adaptation* pseudo-code.

4.2 Local Scheduler (LS)

Assuming the task model in 3.2, the LS executes at each SP as a traditional LST scheduler [9]. The LST algorithm was chosen because it has been proved optimal for single cores, and due its support to deadlines different from the period [9], which is not supported in EDF (Earliest Deadline First). Note that in this work the multiprocessor scheduling problem can be reduced to the single core problem, due the presence of the CS, which can migrate a task R_t into an SP with resources to fulfill the task RT constraints.

The RT tasks have scheduling priority over BE tasks. RT tasks are scheduled according to its least slack-time priority. If there are two or more RT tasks with the same slack-time, a round-robin algorithm is used to select the next scheduled task. BE tasks are scheduled only by the round-robin algorithm.

As the system uses an MPI-like communication API, some tasks can be in a waiting state. In this state, BE tasks are simply blocked. However, when an R_t goes to the waiting state, the scheduler handles the R_t as a sleeping task, i.e., the scheduler verifies the end of task period, but do not update its remaining execution time neither schedules the task. When the task receives the requested message, the µkernel changes the state of the task to ready and calls the LS. The scheduler then updates the slack-time and the remaining execution time for all its RT ready tasks, scheduling the task using the LST priority.

Most schedulers use a fixed scheduler timer (S_T), or *quantum*, to schedule the tasks [9][4] (for example, EDF uses fixed time slices). This *quantum* is the interval between the scheduler calls. The proposed LS uses a variable *quantum*. Setting the appropriate S_T is challenging, because it may induce deadline misses caused by excessive scheduler executions. The method to compute S_T is executed after selecting a given task $t_s \in T$ to run. The goal is to let t_s run, minimizing scheduler interruptions, without compromising other RT tasks. Let ψ be the set of R_t allocated into a given SP, minus t_s.

The S_T value is computed applying the following steps:

1. Selection of the *first end of period* for all *sleeping* and *waiting* tasks $s \in \psi$, using Equation 1:

$$S_{tc1} = \begin{cases} C_t - first\_end\_of\_perido(s_1 \dots s_n), & n > 0 \\ t_{s(r)}, & n = 0 \end{cases} \quad (1)$$

where C_t is the current system time, and *n* is the number of tasks $s \in \psi$. The value S_{tc1} ensures that the scheduler will be called at the first end of period of a task $s_i \in \psi$. The S_{tc1} value ensures a scheduler call to awake a sleeping task or to verify if a waiting task missed a deadline. The default value, if n=0, is remaining execution time of t_s: $t_{s(r)}$.

2. Selection of the *minimum slack-time*: *minST()* for all *ready* tasks $r \in \psi$, using Equation 2:

$$S_{tc2} = \begin{cases} \text{minST}(r_1 \dots r_n), & n > 0 \\ t_{s(r)}, & n = 0 \end{cases} \quad (2)$$

where *n* is the number of tasks $r \in \psi$. The S_{tc2} value ensures the execution of t_s up to the expiration of the smallest slack-time of a task $r_i \in \psi$.

3. Selection of the scheduler timer S_T, using the Equation 3:

$$S_T = \min(S_{tc1}, S_{tc2}, t_{s(r)}) \quad (3)$$

If $t_{s(r)}$ is smaller than the previously computed values, it is adopted as the *quantum* value. After selecting S_T, tasks t_s start their execution using the S_T value as *quantum*.

5. RESULTS

Results were obtained using a clock-cycle accurate RTL SystemC model of the MPSoC. The µkernel, application code, and the scheduler was implemented in C. Results use two latency metrics:

- *Task iteration latency*, time to execute a task iteration, which can e.g. be a loop (Figure 4).
- *Application iteration latency*, time for an application to execute its hyper period.

5.1 Slack-time Monitoring

The evaluation of the STM includes: *accuracy* and *performance overhead*. The accuracy evaluation employs an 8x8 MPSoC divided into four 4x4 clusters. To estimate the SPs' slack-time, the SPs received only RT tasks. Figure 7(a) presents the annotated utilization for each SP (%). Figure 7(b) presents the monitored slack-time (%). It is possible to note that the monitored slack-time is in practice the remaining utilization of Figure 7(a), with the sum of SPs utilization with the monitored slack-time reaching 99%. In fact, the remaining 1% is related to OS overheads. *Such results demonstrate the accuracy of the monitored slack-time.*

9	11	34	7	81	77	82	10
6	29	15	10	57	5	62	14
21	55	15	44	12	25	80	24
CM	58	86	6	CM	6	12	14
16	47	12	67	69	13	66	9
50	5	10	14	29	15	14	46
9	19	9	74	81	11	12	54
CM	40	79	59	CM	27	15	46

(a) SPs utilization (**%**)

90	88	65	92	18	22	17	89
93	70	84	89	42	94	37	85
78	44	84	55	87	74	19	75
CM	41	13	93	CM	93	87	85
83	52	87	32	30	86	33	90
49	94	89	85	70	84	85	53
90	80	90	25	18	88	87	45
CM	59	20	40	CM	72	84	53

(b) Monitored SPs slack-time (**%**)

Figure 7. (a) SPs utilization using RT tasks. (b) Monitored SPs slack-time. Each square represents a PE.

To evaluate the performance overhead due to STM, a 12x12 MPSoC divided into nine 4x4 clusters is used, running a mix of RT and BE applications. The monitoring window is set to 10 ms. Figure 8 presents the STM overhead for each SP. This overhead is related to the time required to handle the STM interruption and to send the slack-time message to the CS. As can be observed, the overhead in most SPs falls between 100 and 150 *clock cycles* (cc),

with an average of 132 cc. There are a few large values, which can be explained by NoC congestion, forcing the packet to wait for the router to be released. The overhead in the CM to handle the STM packets was 1620 cc (only software execution). *Such result shows the small penalty to monitor the slack time.*

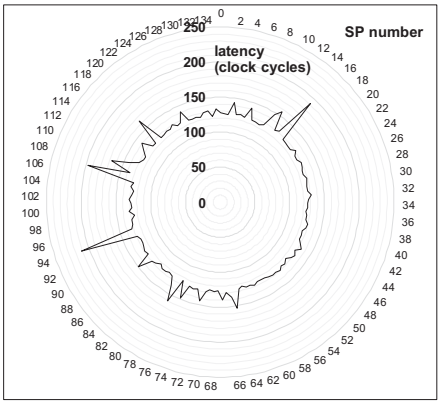


Figure 8. STM overhead for SPs in a 12x12 MPSoC.

Different STM windows were evaluated: 1, 2, 5 and 10 ms. The cost of handling a monitoring message does not change with the STM window. Reducing the monitoring window may reduce the time to adapt the system, at the cost of increased processing in CMs, due to the larger number of packets to deal with. The monitoring window is a design choice, enabling to establish a trade-off between reaction time and performance overhead in the manager processors.

5.2 RT Adaptation Support

This Section evaluates the RT adaptation support, observing the scheduler behavior when a *RealTime* syscall occurs. Figure 9 presents the CPU utilization for a given SP running two RT tasks: *t1* and *t2*. At the beginning of the execution, both tasks configure its RT constraints: $t1_{(u)}$=20%, $t2_{(u)}$=30%. Near 52 ms, *t2* changes its RT constraints, configuring $t2_{(u)}$=65%. It is possible to note that after the second *RealTime*, *t2* executes for a longer period, corresponding to the utilization configured in the second *RealTime* call.

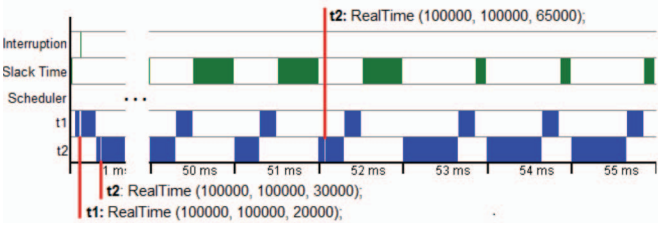


Figure 9. Change in the CPU time utilization during an RT adaptation (rectangles represent the CPU utilization).

Figure 10 presents the *task iteration latency* for *t1* and *t2*, considering the scenario presented in Figure 9. It is possible to note that the *t1* latency is not affected when the *t2* workload increases, demonstrating the capability of the LS to preserve the RT constraints. The small peak near 52 ms observed in the graph occurs due the RT adaptation process. Figure 10(b) shows that after *t2* request more CPU resources, the latency decreases in the same proportion.

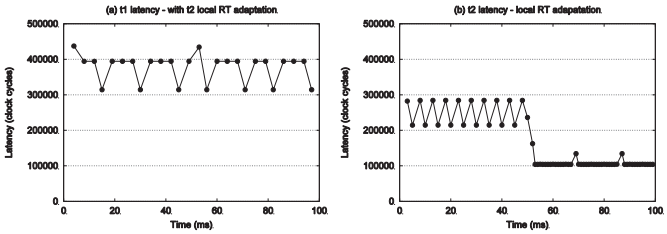


Figure 10. Task iteration latency change during an RT adaptation. (a) *t1* latency. (b) *t2* latency.

5.3 RT Adaptation with Task Migration

This Section presents an RT adaptation, Figure 11, that includes task migration. The same scenario of the previous Section is used. However, the second RT configuration of *t2* exceeds the SP utilization: $t2_{(u)}$=85%. This utilization, with $t1_{(u)}$, would result in an SP utilization equal to 105%. When CS receives the *RT request* message related to the second *t2* RT change, it executes the *RT_adaptation* (Figure 6(b)) to select an SP with enough remaining utilization.

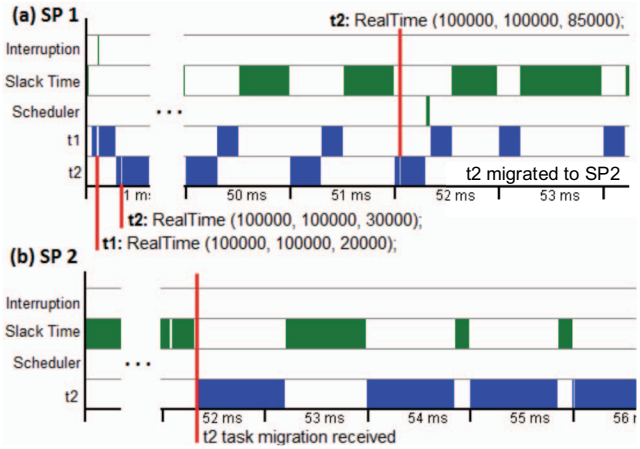


Figure 11. Change of the CPU time occupation during an RT adaptation with task migration. Task t2 start to execute in SP2 when the RT constraint changes.

Observing the chart of Figure 11(a), it is possible to note that near 52 ms *t2* call the second *RealTime*, inducing a task migration from SP 1 to SP 2 (Figure 11(b)). The total time between the start of RT adaptation until the end of task migration was 8906 cc, with 2651 cc (29.7%) required to the RT adaptation process, and 6255 cc (70.3%) required to the task migration protocol.

Figure 12(a) and Figure 12(b) present the *task iteration latency* for *t1* and *t2*, considering the scenario presented in Figure 11. It is possible to observe the negligible impact of RT adaptation even with task migrations taking place.

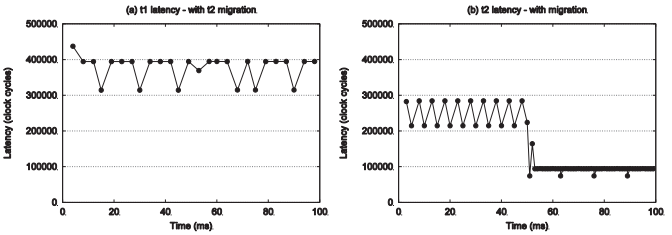


Figure 12 . t1(a) and t2(b) task iteration latency during an RT adaptation with task migration.

5.4 Impact of Disturbing Applications

This Section evaluates two RT applications: DTW (computation intensive, six tasks) and MPEG (communication intensive, five tasks). These applications – named *target applications*, were evaluated in the presence of disturbing applications (RT or BE). The goal is to observe the scheduler behavior with multiple tasks allocated in the same SP, and the impact of the disturbing applications over the target applications.

Initially, target applications execute alone in the system aiming to collect the reference RT constraints for its tasks (profiling step). In sequence, new simulations were performed aiming to insert RT interference over the target applications. According to Figure 3, the available time to execute disturbing tasks, $t_{disturb}$, corresponds to $R_t(p)$- $R_t(e)$. The experiments vary $t_{disturb}$ from 10% (0.1* $t_{disturb}$) to 90% (0.9 * $t_{disturb}$).

Figure 13(a) presents the DTW average *application iteration latency*. The first column presents the *minimal* latency, next columns present the interference of a RT disturbing application varying $t_{disturb}$ from 10% (10% RT) to 90% (90% RT), the column "BE" corresponds to a BE application interference with $t_{disturb}$ equal to 100%. The last column presents the latency when a round-robin scheduler (i.e. without RT support) is used. Is possible to observe that for all RT disturbing scenarios the DTW latency close to the minimal latency (36220.03 cc.), with an average latency increase of 2%, and a standard deviation of 316.4 cc. Such results demonstrate the scheduling ability to preserve the RT application constraints even with high resource sharing. BE disturbing applications do not influence the latency values. Disabling the RT support the latency increase 97.13% compared with the minimal latency, demonstrating that a RT scheduler is required to meet deadline constraints. The deadline miss rate for DTW with RT disturbing was 0.66%, and for the disturbing application was in average 2.7%. Figure 13(b) presents the DTW execution time. The results are similar to the latency results. The additional column "*minimal–no STM*", corresponds to the scenario to obtain the minimal latency, but disabling the STM. Note the negligible impact of STM monitoring, increasing the application execution time by 0.4% (using a 5 ms STM window).

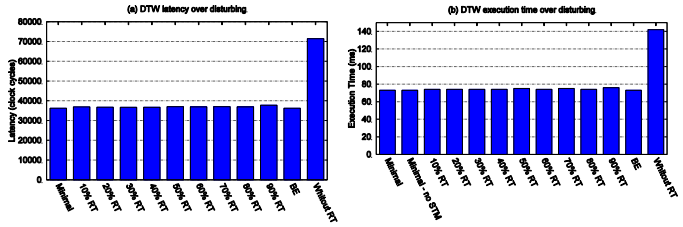


Figure 13. (a) DTW application latency over disturbing. (b) DTW execution time over disturbing.

The *application iteration latency* observed in the MPEG application is similar to the DTW application. All latencies with RT disturbing remain close to the latency of the minimal scenario (55358.5 cc.), with an average latency increase of 0.5% and standard deviation equal to 181.1 cc. The frame rate achieved in the minimal scenario was 178 frames per 100ms. With RT disturbing applications, the frame rate has presented a maximum of 2 frames decrease compared to the minimal scenario. Disabling the RT support, the frame rate drops to 81 frames per 100ms. The impact of STM monitoring in the MPEG application was 0.07% (STM windows of 5 ms). The deadline miss rate for MPEG, with RT disturbing application, was 0.18%, and for the disturbing application was equal to 2.1%.

The deadline miss difference between the disturbing application and the RT application occurs because the disturbing application is computationally intensive, which makes such application more sensitive during RT scheduling. This same observation can be used to explain the difference of deadline misses between DTW and MPEG.

6. CONCLUSIONS

This work presented a hierarchical scheduler for large-scale MPSoCs. The scheduler is completely adaptive, supporting dynamic task RT constraints and slack-time monitoring. The evaluation demonstrated a reliable ability to fulfill RT applications with soft deadlines even for communication or computation intensive applications with the interference of other RT applications. The observed worst-case application latency increase was 2% (DTW), with 90% of RT disturbing over the minimal latency. The STM presented a negligible impact on the execution time of application, with a worst execution time increase of 0.4%.

Large-scale MPSoCs have other issues to achieve QoS. One of these issues is the increasingly NoC unpredictability as the system grows in the number of cores. To cope with this, runtime QoS techniques applied to the NoC can be employed together with the proposed scheduler, to provide a full QoS support. Other future works include to integrate the scheduler with a task mapping and schedulability analysis algorithm and add energy awareness to the heuristic.

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

[2] Singh, A.K.; Shafique, M.; Kumar, A.; Henkel, J. "Mapping on multi/many-core systems: Survey of current and emerging trends". In: DAC, 2013, pp.1-10.

[3] Jung, H.; et al. "Dynamic Behavior Specification and Dynamic Mapping for Real-Time Embedded Systems: HOPES Approach" ACM Trans. Embed. Comput. Syst., vol 13(4), 2014, 26 p.

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

[5] Gangadharan, Deepak; Chakraborty, Samarjit; Zimmermann, Roger, "Quality-aware media scheduling on MPSoC platforms". In: DATE, 2013, pp. 976-981.

[6] Rosvall, K.; Sander, I. "A constraint-based design space exploration framework for real-time applications on MPSoCs". In: DATE, 2014, pp.1-6.

[7] Bamakhrama, M.; Stefanov, T. "Hard-real-time scheduling of data-dependent tasks in embedded streaming applications". In: EMSOFT, 2011, pp.195-204.

[8] Tafesse, B.; Raina, A.; Suseela, J.; Muthukumar, V., "Efficient Scheduling Algorithms for MpSoC Systems". In: ITNG, 2011, pp. 683-688.

[9] Liu, J.W.S. "Real-Time System". Printice Hall, New Jersey, 2000.

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

[11] Theodoropoulos,D.; Pratikakis,P.; Pnevmatikatos,D. "Efficient runtime support for embedded MPSoCs". In: SAMOS, 2013 pp.164-171.

[12] Cannella, E.; Derin, O.; Meloni, P.; Tuveri, G.; Stefanov, T. "Adaptivity support for MPSoCs based on process migration in polyhedral process networks". VLSI Design, 2012, Article 2.

[13] Hansson, A.; et al. "Design and implementation of an operating system for composable processor sharing". Microprocessors and Microsystems, v. 35 (2), pp. 246-260, March 2011.

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

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