Robust non-preemptive hard real-time scheduling for clustered multicore platforms

Michele Lombardi, Michela Milano, Luca Benini
DEIS - Università di Bologna
{michele.lombardi2,michela.milano,luca.benini}@unibo.it

***Abstract*—Scheduling task graphs under hard (end-to-end) timing constraints is an extensively studied NP-hard problem of critical importance for predictable software mapping on Multi-processor System-on-chip (MPSoC) platforms. In this work we focus on an off-line (design-time) version of this problem, where the target task graph is known before execution time. We address the issue of scheduling robustness, i.e. providing hard guarantees that the schedule will meet the end-to-end deadline in presence of bounded variations of task execution times expressed as min-max intervals known at design time. We present a robust scheduling algorithm that proactively inserts sequencing constraints when they are needed to ensure that execution will have no inserted idle times *and* will meet the deadline for any possible combination of task execution times within the specified intervals. The algorithm is complete, i.e. it will return a feasible graph augmentation if one exists. Moreover, we provide an optimization version of the algorithm that can compute the shortest deadline that can be met in a robust way.**

I. Introduction

Multicore platforms have become widespread in embedded computing. This trend is propelled by ever-growing computational demands of applications and by the increasing number of transistors-per-unit-area, under continuously tightening energy budgets dictated by technology and cost constraints [1], [2], [3]. Effective multicore computing is however not only a technology issue, as we need to be able to make efficient usage of the large amount of computational resources that can be integrated on a chip. This has a cross-cutting impact on architecture design, resource allocation strategies, programming models.

Scalable performance is only one facet of the problem: predictability is another big challenge for embedded multicore platforms. Many embedded applications run under real-time constraints, i.e. deadlines that have to be met for any possible execution. Predictability and computational efficiency are often conflicting objectives, as many performance enhancement techniques aim at boosting expected execution time, without considering potentially adverse consequences on worst-case execution. Hence applications with strong predictability requirements often tend to under-utilize hardware resources [4] (e.g. forbidding or restricting the use of cache memories, limiting resource sharing, etc.)

The objective of our work is *predictable and efficient non-preemptive scheduling of multi-task applications with inter-task dependencies*. We focus on non-preemptive scheduling because it is widely utilized to schedule tasks on clustered domain-specific data-processors under the supervision of a general-purpose CPU [5], [6]. Non-preemptive scheduling known to be subject to *anomalies*[1] when tasks have dependencies and variable execution times [8]. Traditional anomaly-avoidance techniques [9] rely on either synchronizing task switching on timer interrupts (which unfortunately implies preemption), or on delaying inter-task communication (or equivalently stretching execution time).

In our formulation, the target platform is described as a set of resources, namely execution clusters (i.e one or more tightly coupled processors with shared memory) and communication channels. An application is a set of dependent tasks that have to be periodically executed with a max-period constraint (i.e. deadline equal to period), modeled as a task graph where task execution is characterized by known min-max intervals and inter-task dependencies are associated with a known amount of data communication.

A preliminary heuristic allocation step assigns tasks to processor clusters targeting workload balancing and minimization of inter-cluster communication. Task migration among clusters is not allowed and partitioned scheduling is performed offline by our algorithm. The online (run-time) part of the scheduler is extremely simple: when a task completes execution on a cluster, one of the tasks for which all predecessors have completed execution is triggered.

Our main contribution can be summarized as follows. We developed a *robust* scheduling algorithm that proactively inserts *additional* inter-task dependencies only when required to meet the deadline for any possible combination of task execution times within the specified intervals. Our approach avoids idle time insertion, hence it does not artificially lower platform utilization, it does not need timers and related interrupts, nor a global timing reference. The algorithm is complete, i.e. it will return a feasible graph augmentation if one exists. We also propose an iterative version of the algorithm for computing the tightest deadline that can be met in a robust way. Even though worst-case runtime is exponential, the algorithms have affordable run times (i.e seconds to minutes) for task graphs with tens of tasks.

II. Related work

Non-preemptive real-time scheduling on multiprocessors has been studied extensively in the past. We focus here on

[1] execution traces where one or more tasks run faster than their worst-case execution time but the overall application execution becomes slower

978-3-9810801-5-5/DATE09 © 2009 EDAA

partitioned scheduling. We can cluster previously proposed approaches in two broad classes, namely: online and offline. Online techniques target systems where workloads become known only at run-time. In this case, allocation and scheduling decision must be taken upon arrival of tasks in the system, and allocation and scheduling algorithm must be extremely fast, typically constant time or low-degree polynomial time. Given that multiprocessor allocation and scheduling on bounded resources is NP-Hard [10], obviously, online approaches cannot guarantee optimality, but they focus on safe acceptance tests, i.e. a schedulable task set may be rejected but a non-schedulable task set will never be accepted.

An excellent and comprehensive description of the online approach is given in [9]. It computes a fast, heuristic allocation and schedule and performs a polynomial-time schedulability test verify that deadlines can be met. A given task graph can fail the schedulability test even if its execution would actually meet the deadline. This is not only because the test is conservative, but also because several heuristic, potentially sub-optimal decisions have been taken before the test. Recent work has focused on techniques for improving allocation and reducing the likelihood of failing schedulability tests [11], [12].

Offline approaches, like the one proposed in this paper, assume the knowledge of the task graph and its characteristics before execution starts. Hence, they can afford to spend significant computational effort in analyzing and optimizing allocation and scheduling. In this field, we distinguish two main sub-categories. One sub-category assumes a fixed, deterministic, execution time for all tasks. Even though the allocation and scheduling problems remain NP-Hard, efficient complete (exact) algorithms are known (see for instance [13] and references therein) that work well in practice, and many incomplete (heuristic) approaches have been proposed as well for very large problems that exceed the capabilities of complete solvers [14]. These approaches can also be used in the case of variable execution times, but we need to force determinism: at run-time, task execution can be stretched artificially (e.g. using timeouts or idle loops) to always complete as in the worst case. This eliminates scheduling anomalies, but implies significant platform under-utilization and unexploitable idleness [9]. Alternatively, powerful formal analysis techniques can be used to test the absence of scheduling anomalies [15]. However, if an anomaly is detected, its removal is left to the software designer. This can be a daunting task.

The second sub-category is robust scheduling. The main idea here is to build an a priori schedule with some degree of robusteness. This can be achived by modeling time variability (for example by means of min-max intervals), and inserting redundant resources [22], redundant activities [23] or more generally slack time to absorb disruptions; it is also common to add time redundancy by stretching task durations [24]. A different approach is based on building a backup schedule for the most likely disruptions [25]. Robust scheduling has been extensively investigated in the process scheduling community [16]. In this paper we leverage a technique called Precedence

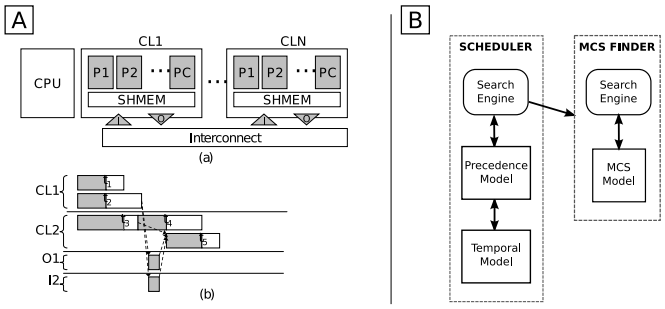

Fig. 1. (A) Target Platform; (B) Architecture of the solver

Constraint Posting (PCP) [18], [17], [21] devised for Resource Constraint Project Scheduling problem (RCPS) to obtain a robust schedule by preventing resource conflicts with added precedence constraints. Compared to existing PCP approaches, we directly exploit known time bounds on task durations and guarantee feasibility for every possible scenario; existing approaches assume fixed durations and focus on adding some degree of flexibility to the schedule, without giving any guarantee.

III. Platform & Application Models

A common template for embedded MPSoCs features an advanced general-purpose CPU with memory hierarchy and OS support and an array of programmable processors, also called *accelerators*, for computationally intensive tasks. Accelerators have limited or no support for preemption and feature private memory subsystems [5], [6], [7]. Explicit communication among accelerators is efficiently supported (e.g. with specialized instructions or DMA support), but uniform shared memory abstraction is either not supported or it implies significant overheads [2].

We model the heterogeneous MPSOC as shown in Figure 1A (a). All shaded blocks are resources that will be allocated and scheduled. For the sake of clarity, all empty blocks are not modeled in the problem formulation. Accelerators $(CL1, \ldots, CLN)$ are modeled as clusters of one or more execution engines. A cluster with C execution engines models an accelerator with multiple execution pipelines (e.g. a clustered VLIW) or a multi-threaded processor with a fixed number of hardware-supported threads. Note that the case of $C = 1$ (single-instruction-stream accelerator) is handled without loss of generality. When two dependent tasks are allocated onto the same cluster, their communication is implicitly handled via shared variables on local memory. On the contrary, communication between tasks allocated on two different clusters is explicitly modeled as an activity which uses a known fraction $b_{i,j}$ of the total bandwidth[3] available at the output communication port (triangle marked with O) of

[2]it can be emulated in software by propagating via copy operations all private memory modifications onto external shared memory [5]

[3]Note that a previous study [26] showed how modeling the communication overhead as a fraction of the bandwidth has bounded impact on task durations as long as less than 60% of the available bandwidth is used.

the cluster hosting the source task and the input port (triangle marked with I) of the cluster hosting the destination task. The master CPU is left implicit as its function is mostly related to orchestration.

The Gantt chart in Figure 1A (b) shows a simple example of a scheduled task graph. Activities, called tasks $t_i, i = 1 \dots 5$ do not have a fixed duration, but they are described by duration intervals $[dmin_i, dmax_i]$. In the Figure, the shortest duration corresponds to the shaded part of the rectangles, while the longest duration is represented by the empty part following the shaded part. Tasks t_1, t_2 are scheduled on cluster $CL1$, while task t_3, t_4, t_5 are allocated and scheduled on cluster $CL2$. Both clusters have two execution engines. Dependencies are represented by arrows. Note that inter-cluster communication is explicitly represented by activities "executing" at the same time on the input ($I2$) and output ($O1$) ports of the clusters involved in the communication.

At run-time, a partitioned schedule is described simply by a set of tasks allocated on each processor and the triggering guards of each task, which are identifiers of all predecessor tasks in the graph. The run-time semantic is straightforward: whenever a task terminates execution, a new task is started among those with completely released triggering guards (any tie-breaker rule can be used, e.g. EDF). If no task has fully released guards, the processor remains idle until a guard change triggered by a communication. Note that a single Gantt chart like the one in Figure 1A corresponds to a huge set of actual executions, one for each combination of different durations for all the tasks.

IV. Robust scheduling

Before scheduling, a heuristic allocation of the available resources (clusters) to tasks is computed by means of a multilevel graph partitioning algorithm [19]. Tasks in the graph are split in a number of sets equal to the number of clusters, trying to balance the computational load and to minimize inter-cluster communication. In particular in order to measure the computation load of a task we consider the average execution time; therefore, if S be a set of tasks, its load is:

$$\text{weight(S)} = \sum_{t_i \in S} \frac{dmax_i - dmin_i}{2}$$

The algorithm first attempts to force the difference between the maximum and minimum workload to be within a (small) specified bound, and then, as a secondary objective, minimizes the sum of the bandwidth requirements of arcs connecting tasks in different sets. Note the method is designed for homogeneous computational resources, but generalizes to heterogeneous ones with proper modifications.

Core of the presented work is the scheduling approach and model, ensuring feasibility for each possible composition of task durations; on this purpose, we associate to each task two variables: one representing the starting time of the activity and one representing its end time. In addition, we have precedence relations between pairs of activities that constrain their start and end variables. As task durations are not a priori known, we chose to provide a robust and flexible schedule as a *set of additional precedence relations among tasks added to the task graph by the solver to ensure the capacity of all resources is never exceeded, whatever are the actual task durations at execution time*. In the literature, this technique is referred to as Precedence Constraint Posting [18], [17] and it has been developed for avoiding resource conflict in resource-constrained project scheduling problems by introducing precedence constraints among conflicting activities. In our case the resources are the clusters (with capacity equal to the number of processors) and the communication links (with capacity equal to bandwidth).

The solver architecture is depicted in figure 1B. It is organized as two main components: (1) a Constraint Programming (CP) scheduler, performing search by adding precedence constraints (Precedence model), which are tranlated into task time bounds (Time model); (2) a Minimum Conflict Set (MCS) finder that exploits knowledge on resources as described later, whereas the scheduler has no direct notion of resource constraints; the reason is that usual CP algorithms connecting task time bounds and resources are designed to check consistency for at least one duration value, rather than for all durations. A MCS is a minimal set of tasks which would cause a resource over-usage in case they overlap; here minimal means that by removing any task from the set the resource usage is no longer jeopardized. Due to its minimality, a MCS is resolved by posting a single precedence constraint between two tasks belonging to the set. The search engine of the scheduler iteratively detects MCSs and solves them.

A. Precedence and temporal model

Precedence constraints are represented by means of a set of integer variables P_{ij}, with $i, j \in \{0, \dots, n-1\}$ $i < j$, such that:

$$P_{ij} = \begin{cases} 1 & \text{if } t_i \text{ runs before } t_j \\ -1 & \text{if } t_j \text{ runs before } t_i \\ 0 & \text{if } t_i \text{ and } t_j \text{ overlap} \end{cases}$$

as one can see, the P_{ij} variables are in fact equivalent to using an incidence matrix. The variables are used to represent both existing (defined in the task graph) and added (for the MCS solution) precedence relations; transitive closure is maintained via a set of constraints (not reported here for the sake of simplicity).

From a temporal perspective, for each task we introduce a start and an end variable $S_i, E_i \in 0, \dots, dl$, where dl is the value of the deadline. S_i and E_i must satisfy:

$$E_i - S_i \leq dmax_i, \quad E_i - S_i \geq dmin_i$$

where $dmin_i, dmax_i$ are the minimum and maximum durations of task t_i. Note that, in contrast to resource constraint project scheduling where a set of activity starting times is computed, we chose to provide a schedule as a set of precedence relations. Therefore, in our approach start and end variables are not assigned a fixed value during search; instead, S_i and E_i are introduced to check the satisfiability of deadline constraints. This has an important consequence on scheduler implementation: start times do not need to be triggered on timers, but they can only be conditioned to

completion of tasks, leading to an simple event-triggered scheduler as opposed to a time-triggered one.

Precedence relations and time bounds depend on each other, being connected by the following constraints:

$$P_{ij} = 1 \quad \Leftrightarrow \quad S_i + dmax_i \leq E_j - dmax_j \tag{1}$$
$$P_{ij} = -1 \quad \Leftrightarrow \quad S_j + dmax_j \leq E_i - dmax_j \tag{2}$$

constraints (1) state that if $P_{ij} = 1$ then the time bounds of t_i and t_j must allow t_i to execute before t_j, whatever are their durations at execution time. Conversely, if such a condition on the time bounds is met, a precedence relation between t_i and t_j is detected (this explains the double implication). Constraints (2) is the equivalent statement for $P_{ij} = -1$. Note that the precedence and temporal models have no notion of resources (processors and communication links): those are taken into account by the MCS finder to detect and solve conflict sets.

B. Search and MCS finder

The overall scheduling problem is solved via tree search. At each step the search engine in the MCS finder detects a MCS. It then selects a pair of tasks $t_i, t_j \in MCS$ and opens a choice point, with three outgoing branches: on the first P_{ij} is set to 1, on the second to -1 and on the third to 0 (if neither t_i can precede t_j nor the opposite, it means that t_i and t_j overlap). If a precedence constraint is posted (1 and -1 branches), the MCS is removed and a new MCS is searched and so on, until the graph contains no conflict sets. In case the 0 branch is taken, a new pair of tasks in the same MCS is selected and a new choice point is open; in case all pairs of tasks have been considered the solver fails (the problem contains an unsolvable MCS).

The MCS detection problem itself is stated as a Constraint Satisfaction Problem, whose decision variables are $T_i \in \{0, 1\}$ and $T_i = 1$ is task t_i is part of the selected MCS, $T_i = 0$ otherwise. Let R be the set of problem resources, and let $req(t_i, r_k)$ be the quantity of a resource $r_k \in R$ requested by a task t_i; $req(t_i, r_k) = 0$ is the tasks does not require the resource. The set of tasks in an MCS must exceed the usage of at least one resource:

$$\bigvee_{r_k \in R} \left(\sum_{t_i} req(t_i, r_k) T_i > cap(r_k) \right) \tag{3}$$

where and $cap(r_k)$ is the capacity of resource r_k. At least one of the inequalities between round brackets in expression 3 must be true for the constraint to be satisfied. Moreover, the selected set must be minimal, in the sense that by removing any task from the MCS, the resource over-usage no longer occurs[4]. More formally:

$$\forall r_k \in R : \sum_{t_i} req(t_i, r_k) T_i - \min_{T_i = 1} \{req(t_i, r_k)\} \leq cap(r_k)$$

that is, the capacity for all r_k is not exceeded if we remove from the set the task with minimum requirement.

Finally, a MCS must contain tasks with an actual chance of overlapping: therefore if a precedence relation exists between two tasks t_i and t_j they cannot be both part of the same MCS: $\forall t_i, t_j \mid t_i \prec t_j \vee t_j \prec t_i : \quad T_i \times T_j = 0$ the presence of such exclusion constraints makes the MCS finding problem NP-hard. A list of mutually exclusive tasks is given to the MCS finder by the scheduling search engine at each step of the solution process. The MCS finding problem is solved again via tree search. Tasks are selected according to their resource requirements, giving precedence to the most demanding ones: this biases the selection towards smaller MCS, and thus choice points with fewer branches; this often reduces the search effort to solve the scheduling problem.

C. Feasibility and optimality

The solver described so far solves the feasibility version of the problem. In fact, it finds a set of precedence constraints that added to the original task graph ensure to meet deadline constraints in every run time scenario (i.e., for each combination of task durations). A simple optimality version of the solver can easily be defined by performing binary search on the space of possible deadlines and iteratively calling the above described solver.

Initially, an infeasible lower bound (e.g. 0) and a feasible upper bound (e.g. the sum of the durations) for the deadline are computed; an optimal deadline is then computed by iteratively narrowing the interval $]lb, ub]$ by solving feasibility problems. At each step a tentative bound tb is computed by picking the value in the middle between lb and ub, then the problem is solved using tb as deadline. If a solution is found, the tb is a valid upper bound for the optimal deadline; conversely, tb is a valid lower bound. The process stops when lb is equal to ub minus 1. The worst-case execution time of both the feasibility solver and the optimality solver is clearly exponential. However, search is very efficient in practice, as discussed in details in the next section. An interesting feature of the optimality version of the algorithm is that in case it exceeds the time limit, it provides a lower and an upper bound on feasible deadlines anyway. Experimental results will show that, for the considered instances, such bounds are tight even for the largest task graphs.

		feasible		infeasible		total					
P	N	DL	T_a/T_m	DL	T_a/T_m	T_a/T_m	N_{it}	N_{pr}	TL	I/F	FL
1	20	1023	0.4/0.4	1022	0.2/0.1	4.9/5.1	11	13	0	1.00	31%
	30	1448	1.4/1.2	1447	10.5/0.2	39.4/17.3	12	22	0	1.00	35%
	40	1867	1.9/2.0	1866	0.3/0.5	34.3/32.3	12	23	1	1.00	34%
	50	1936	5.9/6.2	1890	8.3/1.0	119.5/94.8	11	23	2	0.98	32%
	60	2124	10.7/8.6	2089	9.3/1.7	192.4/185.0	11	45	3	0.98	34%
	70	2390	19.7/19.7	2306	7.0/2.0	300.5/301.2	9	50	6	0.97	34%
2	20	930	0.1/0.1	929	0.1/0.1	1.1/1.2	11	8	0	1.00	32%
	30	1360	0.2/0.2	1359	0.1/0.1	4.7/4.7	11	7	0	1.00	35%
	40	1751	0.3/0.3	1750	0.2/0.2	10.1/10.2	12	7	0	1.00	35%
	50	1758	1.0/0.8	1757	0.4/0.4	24.5/22.7	12	15	0	1.00	33%
	60	1959	1.9/1.8	1914	0.6/0.6	41.2/37.8	10	25	2	0.98	34%
	70	2108	5.1/4.9	2107	1.0/1.0	89.3/84.9	13	41	0	1.00	33%

TABLE I
FIRST GROUP OF INSTANCES (VARIABLE SIZE)

[4]Indeed, we have experimentally observed that by relaxing the minimality constraints does not affect the correctness of the solver, but speeds up the solution process.

V. Experimental results

Both the scheduler and the MCS finder have been implemented within the of the state-of-the-art ILOG Solver 6.3. Preliminary heuristic allocation is performed using the METIS [20] graph partitioning package. We forced the difference between the maximum and minimum workload to be within 6%.

We tested the system on groups of randomly generated task graphs designed to mimic the structure of real-world applications. In particular all groups contain so called structured graphs: they have a single starting task and a single terminating task, and feature branch nodes with variable fan out and a collector node for each branch; this reflects typical control/data flows arising with stuctured programming languages (such as C or C++). For all instances, the maximum task duration can be up to twice the minimum durations; data transfers within the same clusters are assumed to be instantaneous, while inter-cluster communications have finite duration (up to $1/4$ of average task execution time) and use from 10% to 90% of the port bandwidth. We consider two different platforms: the first one has 16 single-instruction-stream accelerators connected with the interconnect by ports with the same bandwidth, the second contains 4 accelerators with 4 processors each. Inter-cluster communications employ ports with the same bandwidth as in the first platform.

Furthermore, the approach was compared with a Worst Case Execution Time (WCET) scheduler, assuming the maximum duration for each task; such method is much faster, but requires time triggered scheduling and idle time, hence achieving robustness at the cost of flexibility.

The first group of experiments considers graphs with a fixed fan out (3 to 5 branches) and variable size (from 10 to 70 tasks). In each experiment we ran the optimality solver described in Section IV-C: this process requires to solve a number feasibility problems in order to identify the lowest feasible deadline and the highest infeasible deadline. All runs were performed on an Intel Core 2 Duo, with 2GB or RAM; the solver was given a time limit of 15 minutes.

Table I lists the results for the first group of experiments, related to platform type 1 and 2 (column "P"). Each row reports results for groups of 10 instances, showing the average deadline for the lowest feasible and the highest infeasible solution found (columns DL), the ratio between those two values (I/F) the number of added precedence relation in the best solution (N_{pr}). Detailed information is given about the solution time: the average and median time for the optimization process is reported in column total T_a/T_m; column TL shows the number of timed out instances (which are not considered in the average and median computation); N_{it} contains the average number of iterations performed. To give an evaluation of the feasibility version of the solver the solution time for the highest infeasible and lowest feasible deadlines is also given. Finally, the retained flexibility (FL) w.r.t. the WCET scheduler is shown: this is computed as $(CT_{WCET} - \min(CT_{PCP}))/CT_{WCET}$, where CT_{WCET} is the completion time of the WCET schedule and $\min(CT_{PCP})$ is the minimum possible completion time of the PCP schedule. Note that time for the allocation step is negligible and is omitted in the results. Similarly, the running time for the WCET scheduler is omitted, but it is always much faster then the PCP approach.

P	N	feasible DL	feasible T_a/T_m	infeasible DL	infeasible T_a/T_m	total T_a/T_m	N_{it}	N_{pr}	TL	I/F	FL
1	2-4	1765	2.2/2.0	1764	0.4/0.5	33.5/32.0	12	17	0	1.00	32%
	3-5	1573	3.9/3.8	1572	0.8/0.6	53.8/54.6	12	36	0	1.00	32%
	4-6	1551	59.4/5.4	1550	0.7/0.5	119.7/63.0	12	49	0	1.00	33%
	5-7	1406	5.6/4.9	1405	1.5/0.5	117.6/63.7	12	53	0	1.00	34%
	6-8	1318	6.6/6.2	1295	0.4/0.3	75.2/68.0	10	60	3	0.98	33%
2	2-4	1643	0.4/0.3	1642	0.2/0.2	10.7/9.8	12	8	0	1.00	32%
	3-5	1453	0.7/0.6	1452	0.2/0.2	15.0/14.5	12	14	0	1.00	33%
	4-6	1472	0.9/0.9	1471	0.2/0.2	16.5/15.9	12	21	0	1.00	33%
	5-7	1305	1.1/0.9	1304	0.2/0.2	18.6/17.4	12	23	0	1.00	34%
	6-8	1223	1.3/1.4	1218	0.2/0.2	19.7/20.4	11	28	2	1.00	33%

TABLE II
Second group of instances (variable fan-out)

Considering table I, clearly, the solution time and the number of timed out instances grows as the number of tasks increases. The very high values for the deadline ratio show how tight bounds for the optimal deadline can be computed even for timed out instances. This aspect is crucial for designing a heuristic method based on the optimal version of the algorithm proposed that provides deadline bounds at a given time limit. Differences between the average and median solution time can be used as an indicator of the stability of the solution time; in particular very close average and median values are reported for feasible deadlines, while a much less stable behavior is observed for infeasible deadlines: detecting infeasibility seems to be often an easy task, but rare complexity peaks do arise; this could be expected, being the problem NP-hard. Mapping and scheduling the considered task graphs on the clustered platform appears to be an easier task. Not only the solution time is lower, but the behavior of the solver is much more stable; this is probably due to the smaller number of inter-cluster communications, which are scheduled as resource demanding tasks, and therefore increase the actual problem size and the number of conflict sets. The shorter deadlines are also due to the reduced number of inter-cluster communications.

To investigate the solver sensitivity to graph parallelism, in the second group of experiments we considered instances with a fixed number of tasks, but variable fan out (from 2-to-4 to 6-to-8). The results for this second group of instances, mapped respectively on platform 1 and 2, are reported in table II. Again, each row summarizes results for 10 experiments. Both for platform 1 and 2, increasing the fan out makes the problem more difficult, since the higher parallelism translates into a higher number of conflict sets. For the same reason also more precedence relations have to be added to prevent the occurrence of resource over-usage. As for the previous group, mapping and scheduling on the clustered platforms easier and produces better optimal deadlines.

The last group of instances contains an aggregation of small,

P	N	feasible DL	feasible T_a/T_m	infeasible DL	infeasible T_a/T_m	total T_a/T_m	N_{it}	N_{pr}	TL	I/F	FL
1	20	929	0.1/0.1	928	0.1/0.1	1.4/1.5	11	5	0	1.00	34%
	30	1319	0.3/0.3	1318	0.1/0.1	6.7/6.6	11	15	0	1.00	36%
	40	1347	0.5/0.4	1346	0.2/0.3	14.2/14.5	12	23	0	1.00	34%
	50	1342	1.5/1.3	1341	1.6/0.4	37.8/32.4	12	44	0	1.00	32%
	60	1378	7.3/2.7	1377	21.5/6.5	166.9/82.1	12	79	0	1.00	32%
	70	1851	16.8/4.6	1849	54.1/1.4	274.2/112.6	12	113	3	1.00	35%
2	20	871	0.1/0.1	870	0.1/0.1	0.5/0.5	11	1	0	1.00	33%
	30	1238	0.1/0.1	1237	0.1/0.1	1.7/1.7	11	1	0	1.00	36%
	40	1259	0.1/0.1	1258	0.1/0.1	4.6/4.4	12	4	0	1.00	35%
	50	1205	0.3/0.2	1204	0.2/0.2	9.4/9.0	12	15	0	1.00	31%
	60	1217	0.6/0.6	1216	0.3/0.3	17.2/17.5	12	35	0	1.00	32%
	70	1624	1.7/1.5	1623	0.5/0.5	34.2/35.2	12	47	0	1.00	34%

TABLE III
THIRD GROUP OF INSTANCES (VARIABLE #SUBGRAPHS)

independent structured task graphs with low fan out (2-3). This case is of high interest as the result of the unrolling of a simple multirate system would likely look like this type of graphs. We generate composite graphs with 1 to 7 subgraphs, 10 to 70 tasks. The results for platform 1 and 2 are reported in table III, again by groups of 10 instances. The complexity of the mapping and scheduling problem on platform 1 grows as the number of tasks increases. Note however how, despite the high parallelism calls for a large number of precedence relations, this kind of graphs seems to be easier to solve than those in the first group; this behavior requires a deeper analysis, which is subject of future work. Finally, note how mapping composite graphs on platform 2 is surprisingly easy: the nature of those graph allows the allocator to almost completely allocate different subgraph on different clusters, effectively reducing inter-cluster communications.

VI. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a constructive algorithm for computing robust schedules for real-time applications with variable-duration tasks. Scheduling anomalies are eliminated by inserting sequencing constraints between tasks in a controlled way, i.e. only when needed to prevent resource conflicts which may appear for "unfortunate" combinations of execution times. We developed a solver that solves both feasibility and optimality problems. Even though run-time is worst-case exponential (as expected for complete methods on NP-Hard problems), solution times in practice are very promising: task graphs with several tens of tasks can be handled in a matter of minutes. This is competitive with state-of-the-art verification methods used to test the presence of anomalies [15]. Future work will focus on complete algorithms for allocation and scheduling and on heuristic solvers which can handle extremely large task graphs. Introducing a probability distribution on task durations is also subject of current research: that would enable more complex objectives, such as minimization of the expected execution time, by exploiting probability theory results.

ACKNOWLEDGEMENT

The work described in this publication was supported by the PREDATOR Project funded by the European Community's 7th Framework Programme, Contract FP7-ICT-216008.

REFERENCES

[1] M. Muller, "Embedded Processing at the Heart of Life and Style", IEEE ISSCC, pp. 32-37, 2008.
[2] A. Jerraya, et al. "Roundtable: Envisioning the Future for Multiprocessor SoC" IEEE Design & Test of Computers, vol. 24, n. 2, pp. 174-183, 2007.
[3] D. Yeh, et al., "Roundtable: Thousand-Core Chips", IEEE Design & Test of Computers, vol. 25, n. 3, pp. 272-278, 2008.
[4] L. Thiele et al., "Design for Timing Predictability", Real-Time Systems, vol. 28 pp. 157-177, 2004.
[5] D. Pham, et al. "Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor", IEEE Journal of Solid-State Circuits, vol. 41, n. 1, pp. 179-196, 1996.
[6] M. Paganini, "Nomadik: A Mobile Multimedia Application Processor Platform," IEEE ASP-DAC, pp. 749-750, 2007.
[7] S. Bell et al., "TILE64 Processor: A 64-Core SoC with Mesh Interconnect"', ISSCC, pp. 88-598, 2008.
[8] J. Reineke, et al. "A definition and classification of timing anomalies," International Workshop on worst-case execution time (WCET) analysis, 2006.
[9] J. Liu, Real-Time Systems, Pearson Education, 2000.
[10] M. Garey, D. Johnson, Computers and Intractability: A Guide to the Theory of Np-Completeness, Freeman 1979.
[11] N. Fisher et al., "The Partitioned Multiprocessor Scheduling of Non-preemptive Sporadic Task Systems", RTNS, 2006.
[12] N. Fisher et al., "The Non-preemptive Scheduling of Periodic Tasks upon Multiprocessors", Journal of Real-Time Systems, vol. 32, n. 1-2, pp. 9-20, 2006.
[13] M. Ruggiero et al., "A fast and accurate technique for mapping parallel applications on stream-oriented MPSoC platforms with communication awareness," ACM International Journal of Parallel Programming, vol. 36, no. 1, pp. 3-36, 2008.
[14] E. Zitzler, et al. "Performance Assessment of Multiobjective Optimizers: An Analysis and Review", IEEE Transactions on Evolutionary Computation, vol. 7, no. 2, pp. 117-132, 2003.
[15] A. Brekling, et. al., "Models and Formal verification of multiprocessor systems-on-chip"', Journal of Logic and Algebraic Programming, vol. 77, no 1-2, pp. 155-166, 2008.
[16] Z. Li, et al. "Process scheduling under uncertainty: review and challenges," Computers & Chemical Engineering, vol. 32, pp. 715-727, 2008.
[17] P. Laborie, "Complete MCS-Based Search: Application to Resource Constrained Project Scheduling", IJCAI, pp. 181-186, 2005.
[18] P. Laborie et al., "Planning with Sharable Resource Constraints", IJCAI, 1643-16, 1995.
[19] G. Karypis et al. "A fast and high quality multilevel scheme for partitioning irregular graphs", SIAM Journal on Scientific Computing, vol. 20, no. 1, pp. 359 - 392, 1999
[20] http://glaros.dtc.umn.edu/gkhome/metis/
[21] N. Policella, A. Cesta, A. Oddi, S. F. Smith, "From precedence constraint posting to partial order schedules: a csp approach to robust scheduling", AI Commun. 2007, 20:3, 163-180
[22] S. Ghosh, "Guaranteeing fault tolerance through scheduling in real time systems", PhD thesis, University of Pittsburg, 1996
[23] S. Ghosh, R. Melhem, D. Mosse, "Enhancing real time schedules to tolerate transient faults". In RTSS, 1995
[24] W. Y. Chiang, M. S. Fox, "Protection against uncertainty in a deterministic schedule", proc. of Expert Systems for Production and Operations Management, 1990
[25] M. Drummond, J. Bresina, K. Swanson, "Just in case scheduling", AAAI 94, 1098-1104
[26] L. Benini, D. Bertozzi, A. Guerri, M. Milano, "Allocation and Scheduling for MPSoCs via Decomposition and No-Good Generation", CP2005, 107-121