Robust Scheduling of Task Graphs under Execution Time Uncertainty

Michele Lombardi, Michela Milano and Luca Benini, *Fellow, IEEE*

***Abstract*—Effective multicore computing requires to make efficient usage of the computational resources on a chip. Off-line mapping and scheduling can be applied to improve the performance, but classical approaches require considerable a-priori knowledge of the target application. In a practical setting, precise information is often unavailable; one can then resort to approximate time and resource usage figures, but this usually requires to make conservative assumptions. The issue is further stressed if real-time guarantees must be provided. We tackle predictable and efficient non-preemptive scheduling of multi-task applications in the presence of duration uncertainty. Hard real-time guarantees are provided with limited idle time insertion, by exploiting a hybrid off-line/on-line technique known as Precedence Constraint Posting (PCP). Our approach does not require probability distributions to be specified, relying instead on simple and cheaper-to-obtain information (bounds, average values). The method has been tested on synthetic applications/platforms and compared with an off-line optimized Fixed Priority Scheduling (FPS) approach and a pure on-line FIFO scheduler; the results are very promising, as the PCP schedules exhibit good stability and improved average execution time (14% on average, up to 30% versus FPS and up to 40% versus the FIFO scheduler).**

***Index Terms*—Scheduling, Uncertain Execution and Communication Times, Task Graph, Multi-core, Precedence Constraint Posting, Optimization**

I. INTRODUCTION[1]

MULTICORE platforms have become widespread in embedded computing, propelled by ever-growing computational demand and by the increasing number of transistors-per-unit-area, under continuously tightening energy budgets and cost constraints. 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 on a chip. This has a cross-cutting impact on architecture design, resource allocation strategies and programming models.

Off-line mapping and scheduling is an effective approach to improve the application performance, but relies on considerable a-priori knowledge, often unavailable in a practical setting. Conversely, *approximate* time and resource usage figures are cheaper to obtain, but require either to explicitly deal with uncertainty, or to make very conservative assumptions. The issue is further stressed when the platform lacks efficient preemption support, or when predictability is a primary design goal.

Non-preemptive scheduling is widely used on clustered domain-specific data-processors under the supervision of a general-purpose CPU [1], [2]. Unfortunately, non-preemptive scheduling is known to be subject to *anomalies*[2] when tasks have dependencies and variable execution times [3], resulting in an uncontrolled performance degradation. Traditional anomaly-avoidance techniques [4] rely on delaying inter-task communication or stretching the execution times, leading to a considerable resource overhead.

Predictability is a primary issue whenever the application runs under real-time constraints. Since many performance enhancement techniques have potentially adverse consequences on worst-case execution, applications with strong predictability requirements often tend to under-utilize hardware resources [5] (e.g. forbidding or restricting the use of cache memories, limiting resource sharing, etc.).

Our objective is to perform *predictable and efficient non-preemptive scheduling of multi-task applications in the presence of uncertainty*. We assume the target platform to be described as a set of resources with finite capacity. An application is a set of dependent tasks, modeled as a graph where execution/communication time is characterized by known min-max intervals and unknown probability distribution. The primary focus is on the scheduling procedure (i.e. ordering tasks over time): we assume a resource mapping is available via some external tool.

Building over our previous work [6], [7], we leverage a hybrid off-line/on-line technique from the Project Scheduling domain, known as Precedence Constraint Posting (PCP) [8]. *A schedule in PCP consists of a set of additional arcs in the task graph*: this is implemented at run-time by delaying each task until all its predecessors (either natural or artificial) are over. As a consequence, the completion time adapts to actual task durations and unnecessary idleness is avoided. Finding an effective PCP schedule may prove very challenging: our first contribution is an efficient optimization algorithm for off-line scheduling; the computed solutions are anomaly free, guaranteed to meet real-time constraints (if any) and optimized for low average completion time. The algorithm is complete, i.e. returns a feasible graph augmentation if one exists.

Despite PCP has appealing features for embedded systems scheduling (e.g. reduced idle time, low run-time overhead, no timers and related interrupts), other techniques are traditionally employed. As a second contribution, we compare the results of our scheduling procedure with Fixed

Michele Lombardi, Michela Milano and Luca Benini are with DEIS, University of Bologna, Bologna, 40123 IT
E-mail: {michele.lombardi2,michela.milano,luca.benini}@unibo.it

[1]The work described in this publication was supported by the ARTEMIS project SMECY, ARTEMIS-2009-1-100230.

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

Priority Scheduling (optimized via tabu search) and a pure on-line FIFO scheduler. The results are very promising: the PCP schedules exhibit good stability and even improved average completion time. The comparison is performed on synthetic instances and platforms so as to isolate the contribution of the scheduling strategy to system performance.

The paper is organized as follows: Section II discusses background and related work. The problem definition is provided in Section III, the PCP optimization method is described in Section IV, while the experimental evaluation is presented in Section V.

II. Background and Related work

A. Real-time Scheduling

Scheduling on multiprocessor system in the presence of hard deadlines and limited resource has been considered in the field of real-time scheduling. As we assume here a resource allocation to be pre-computed, we focus on partitioned scheduling. We can cluster previously proposed approaches in two broad classes, namely: on-line and off-line. An excellent and comprehensive description of the on-line approaches is given in [4]; on-line 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 the respective 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, on-line approaches cannot guarantee optimality and focus on safe *acceptance tests*, i.e. a schedulable task set may be (unfortunately) rejected but a non-schedulable task set will never be accepted. More recent works have focused on techniques for improving allocation and reducing the likelihood of failing schedulability tests [9], [10].

Off-line approaches assume the knowledge of the task graph and its characteristics before execution and usually assume a fixed execution time for all tasks. Even though the allocation and scheduling problems remains NP-Hard, efficient complete (exact) algorithms are known (see for instance [11] and references therein) that work well in practice; many incomplete (heuristic) approaches have been proposed for very large problems that exceed the capabilities of complete solvers [12]. These approaches can also be used in the case of variable execution times, but they need to force determinism: at run-time, task execution can be stretched artificially (e.g. using timeouts or idle loops) to terminate as in the worst case. This eliminates scheduling anomalies [13], but implies significant platform under-utilization and unexploitable idleness [4]. Alternatively, powerful formal analysis techniques can be used to test the absence of scheduling anomalies [14]. However, anomaly removal is left to the software designer.

B. Scheduling in OR and Constraint Programming

The problem of scheduling a set of dependent tasks with fixed resource requirements (i.e. pre-mapped) is known in the Operations Research literature as *Resource Constrained Project Scheduling Problem (RCPSP)*; usually, task durations are assumed to be deterministic (e.g. fixed to the WCET[3]). Covering the vast RCPSP literature is beyond the scope of this paper: for the interested reader, work [15] represents an ideal starting point; for a recent survey, one may refer to [16].

In the field of Artificial Intelligence, so-called *Constraint Programming (CP)* has been very successfully applied to the RCPSP. Typically tasks are modeled by pairs of integer variables $\mathtt{S_i}, \mathtt{E_i}$, respectively representing the start/end of task t_i (with duration d_i). Start/end variables must satisfy $\mathtt{E_i} = \mathtt{S_i} + d_i$. Precedence relations between tasks t_i and t_j can be modeled as simple inequalities $\mathtt{E_i} \leq \mathtt{S_j}$. Resource over-usage is prevented by means of the so-called `cumulative` constraint [17].

A solution is usually found via tree search; each constraint in the model (e.g. inequalities or `cumulative`) is an *active* component, embedding a *filtering algorithm*; when the solution process narrows the domain of a variable, filtering algorithms are triggered and prune provably infeasible values in the domains of the other variables, possibly re-triggering the process; if a domain gets empty, a dead-end has been reached and backtracking occurs. The mechanism is known as *constraint propagation* and results in a huge reduction of the search tree. The availability of powerful filtering algorithms for the `cumulative` constraint (e.g. [18]) is at the base of the effectiveness of CP on scheduling problems.

C. Scheduling in the Presence of Uncertainty

Classical RCPSP and multiprocessor scheduling formulations assume complete off-line problem knowledge; most likely, in a real world setting many problem parameters will be unknown before the execution. Many elements of uncertainty can be considered: here we limit ourselves to uncertain activity durations; for a broader perspective one may refer to [19] and [20].

Following the classification in [21], we distinguish between *proactive* and *reactive* approaches. A pure proactive technique produces a fully instantiated schedule (i.e. an assignment of task start times) before the execution. Conversely, pure reactive approaches take all decisions on-line. Most practical approaches lie somewhere in-between, as they produce an initial *baseline schedule* and perform on-line revisions in case of disruptions.

Usually, some knowledge of the duration uncertainty (in the form of a probability distribution) is assumed to be known; this is used to guide search and to evaluate *robustness*. Robust schedules can be obtained through the insertion of slack time [22] or by handing over part of the decisions to an on-line *policy*, i.e. by computing a *flexible* schedule [23], [24].

When uncertain durations are considered, the *expected* completion time (in the probabilistic sense) is the most popular cost function; the exact computation of this value has exponential complexity [24], hence approximations are

[3]Worst Case Execution Time.

needed in practice: sampling is a common approach; a method based on precedence relations is adopted in [25] (to deal with process variations) and requires Normally distributed durations; reference [26] summarizes other approaches based on sampling and Markov processes and introduce a novel and efficient approximate method for the expected completion time with Fixed Priority Scheduling.

So-called Internet Computing/Area Maximization Scheduling (IC- and AM-scheduling, see [27] and [28]) have been introduced to tackle scenarios where resource availability and execution times are highly unpredictable (e.g. computation nodes over the Internet). The approaches are in fact on-line mapping and scheduling policies, aiming to maximize the number of tasks ready to be dispatched throughout graph execution. Duration information is completely disregarded and the choice of which task to dispatch is only based on structural graph properties: optimal dispatching rules for uniform durations can be obtained for specific graph structures; interestingly, they are shown to considerably outperform other widely employed scheduling policies even in case actual durations are non-uniform. The IC/AM approaches cannot provide hard real-time guarantees.

Stochastically robust resource allocation [29] considers single thread processors and no explicit inter-task communication (hence no dependencies); under such assumptions finding an optimal schedule is trivial (tasks mapped to the same processor are simply executed in sequence) and the focus is on providing an optimal mapping; probability density functions are used to describe task durations and to formulate probabilistic deadline constraints.

D. PCP and Temporal Constraint Networks

In this paper, we rely on the Precedence Constraint Posting (PCP) technique to provide a flexible schedule; the method consists in systematically removing possible resource conflicts by adding precedence constraints [8]. Possible conflicts are identified as *Minimal Critical Sets*, i.e. minimal sets of tasks causing a resource overusage in case of overlapping execution [23], [30]. Thanks to the minimality property, an MCS is wiped out (*resolved*) *by adding a single precedence constraint* between a pair of involved tasks; a MCS free graph is anomaly-free as well. Finding MCSs is a non-trivial task, since their number is exponential in the size of the graph. The set of precedence constraints to be added can be obtained via tree search [31] or by means of a heuristic [32].

PCP approaches usually rely on Simple Temporal Networks (STN, see [33]) for temporal consistency check. Simple Temporal Networks with Uncertainty (STNU, see [34]) are STNs extended to provide support for uncertain durations. A STNU has nodes, representing temporal events, and arcs, constraining the time distance between the occurrence of the source and the target event to be in an interval $[min, max]$; in particular, STNUs distinguish between *free* constraints (the distance can be decided at execution time) and *contingent* constraints (the time distance can only be observed). An STNU is said to be *dynamic controllable* if, during execution, the distance for free constraints can be decided so that the partial sequence executed so far is ensured to extend to a complete solution, whatever durations remain to be observed. Dynamic controllability can be enforced in polynomial time as described in [35].

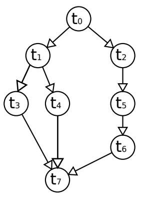
In this work, we adopt a tree-search based PCP approach, close in spirit to [31]. However, we perform MCS selection by solving a (polynomial) minimum flow problem rather than via (exponential) enumeration; as a drawback, the identified MCS may be of poorer quality when used for opening a choice point. Our flow based method is similar to the one in [36], but the algorithm we use is considerably simpler. The underlying temporal model is a STNU, where constraints and filtering replace the ad hoc consistency algorithm employed in [35]; this makes the approach particularly well-suited for implementation on off-the-shelf Constraint Programming solvers[4]. Unlike in stochastically robust resource allocation, we focus on finding high quality schedules for dependent tasks, given a fixed resource mapping; since we do not rely on probability distributions, we provide deterministic (i.e. 100%) rather than probabilistic deadline satisfaction guarantees. Compared to IC/AM-scheduling, we target a very different scenario, where platform resources are deterministically available and approximate knowledge on task durations can be a priori obtained: this information is exploited to improve the schedule quality and to provide hard real-time guarantees. An experimental comparison with IC/AM-scheduling on a simplified problem formulation is beyond the scope of this paper, but will be considered as a topic for future research.

III. Problem Definition

We start by providing an abstract definition of our target problem. The definition allows a broad range of real world platform and applications to be modeled (of course with some limitations): some examples are shown in Section III-B.

We target the problem of scheduling a set of inter-dependent tasks with uncertain duration over a set of pre-assigned hardware resources, in the presence of hard real-time constraints. Formally, the input for the optimization process is a directed, acyclic Task Graph $G = \langle T, A \rangle$, where T is the set of tasks t_i and A is the set of graph arcs (t_i, t_j), representing precedence relations. Each task t_i is annotated with a minimum and a maximum value for the uncertain duration (referred to as d_i and D_i) and with an average value $\bar{d}_i$. Each arc (a_i, a_j) is labeled with minimum and a maximum time lag (referred to as δ_{ij} and Δ_{ij}), so that the time distance between the end of t_i and the beginning of t_j is forced to be larger than δ_{ij} and smaller than Δ_{ij}. Time lags *do not* represent bounds for a

[4]Such as IBM ILOG CP Solver (http://www-01.ibm.com/software/websphere/ilog/), GeCode (http://www.gecode.org/) or Google OR-tools (http://code.google.com/p/or-tools/).



$\mathbf{R} = r_0$	$\mathbf{c_0} = 3$		
$\mathbf{rq_{0,0}} = 1$	$\mathbf{rq_{4,0}} = 2$		
$\mathbf{rq_{1,0}} = 2$	$\mathbf{rq_{5,0}} = 2$		
$\mathbf{rq_{2,0}} = 1$	$\mathbf{rq_{6,0}} = 1$		
$\mathbf{rq_{3,0}} = 1$	$\mathbf{rq_{7,0}} = 2$		

	$\mathbf{d, D, \bar{d}}$		$\mathbf{d, D, \bar{d}}$
$\mathbf{t_0}$	2, 2, 2	$\mathbf{t_4}$	1, 2, 1.1
$\mathbf{t_1}$	2, 3, 2.2	$\mathbf{t_5}$	1, 4, 1.2
$\mathbf{t_2}$	1, 2, 1.5	$\mathbf{t_6}$	1, 2, 1.3
$\mathbf{t_3}$	2, 4, 3.8	$\mathbf{t_7}$	1, 2, 1.5

$\boldsymbol{\delta} = 1$, $\boldsymbol{\Delta} = 7$ for (t_4,t_7)
$\boldsymbol{\delta} = 0$, $\boldsymbol{\Delta} = \infty$ for other arcs

Fig. 1. Task Graph and Problem Input

separation constraint with uncertainty. In detail, a *minimal* time lag is a deterministic separation constraint; there is some analogy with the latency and gap parameters in the LogP model [37], but minimal time lags are not necessarily related to inter-task communication. A *maximal* time lag is a relative deadline, i.e. the maximum allowed distance between a pair of dependent tasks.

We allow a release time (or Earliest Start Time) and a deadline (or Latest End Time) to be specified for each task and denote them as $est(t_i)$ and $let(t_i)$. The execution of t_i must take place within the interval $[est(t_i), let(t_i)]$ and the time lags must be respected, whatever the actual activity durations are. A set of platform resources R is part of the input; each resource r_k has limited capacity c_k and each task t_i requires an amount $rq_{ik} \geq 0$ of every resource r_k ($rq_{ik} = 0$ denotes no requirement).

The problem consists in computing an off-line schedule such that *temporal and resource constraints are satisfied for every possible value of task durations*; furthermore, we wish the expected completion time (or some approximation) to be minimized. Figure 1 shows an example of problem instance. Note duration variability is captured by using *bounds* (e.g. extracted via WCET analysis) and average values rather than via a probability distribution, to reduce the computational complexity. As shown in Section IV-A, this is sufficient to guarantee the satisfaction of hard real-time constraints; as a drawback, the value of the expected makespan can be only roughly approximated: this issue is discussed in Section IV-B. Minimal and maximal time lags are specified for arc (t_4, t_7).

A. Scheduling Problem Solution

Classical off-line Scheduling methods for this scenario assume worst case task durations (i.e. D_i) and specify a fixed starting time of each task (see Section II-A). Figure 2A shows an optimal fixed start schedule for the example problem; due to time lags, t_7 must be scheduled at least 1 time unit after t_4 and t_4 cannot be scheduled more than 7 time units earlier than t_7. In case actual durations are shorter than D_i, idleness is inserted to preserve the start times. However: (1) this technique requires a mechanism to insert idle time (e.g. timers, interrupts); (2) maximal time lags may be violated; (3) unnecessary inefficiency is introduced.

Ideally, one would like to be able to anticipate activities in case their predecessors last shorter. We achieve this goal by adopting a flexible scheduling approach, i.e. by computing an off-line *partial schedule*, where some decisions

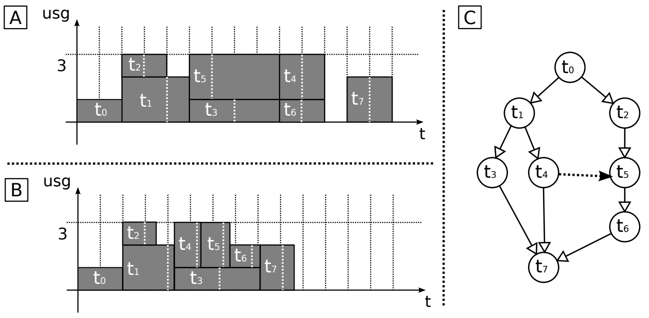


Fig. 2. A-C) Schedules for the Problem of Figure 1. B) Expected completion time approximation.

still have to be taken. In particular, the schedule is specified as a set A^* of additional synchronization constraints (i.e. precedence relations). Those may be implemented as fake data communications, or by maintaining counters for the number of completed predecessors. Figure 2C shows an example of such a partial schedule, consisting in the added arc (t_4, t_5); note the augmented graph corresponds to a huge set of possible schedule instantiations, depending on the actual duration values.

A partial schedule is completed at run-time by starting each task t_i according to an earliest start policy, namely as soon as: 1) t_i has been released and 2) all its predecessors (both actual and artificial) have completed execution. *The additional precedence relations are carefully computed so as to prevent resource contention*; formally:

Definition 1 (Partial Schedule): Given a Task Graph $G = \langle T, A \rangle$, a partial schedule is a set of additional precedence constraints A^* such that, for every possible value of the durations, the start times obtained according to the earliest start policy violate neither temporal nor resource constraints.

No resource conflicts means no scheduling anomalies, so that a partial schedule has predictable worst case behavior. In the partial schedule in Figure 2C, arc (t_4, t_5) has been added to prevent the resource conflict caused by the overlapping execution of t_4 and t_5. The *quality* of a partial schedule is evaluated in terms of expected completion time via the approximation discussed in Section IV-B.

B. Modeling examples

Tasks represent non interruptible computation units, such as non-preemptive processes. Single (dual) thread cores may be represented as resources with capacity one (two). A multi-core CPU can be modeled as a set of resources, or be abstracted as a single resource having the total number of supported threads as capacity.

Memory devices with a limited number of ports can be modeled as resources with capacity equal to the maximum number of simultaneous accesses; a shared bus can be similarly modeled as a resource, with the bandwidth as capacity[5]. Latency due to inter task communications can

[5]A previous study [38] provided evidence that 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.

be included in task durations, or modeled via minimal time lags (e.g. in case of non-blocking communication). Alternatively, communication activities can be described as artificially introduced tasks: this is a more proper approach in case of uncertain communication time (recall time lags are not related to uncertainty), or whenever performing inter-task communication requires some kind of limited resource (e.g. DMA channels, bus bandwidth).

IV. Solution Method

We solve the described problem by means of Constraint Programming (see Section II-B) and tree search; each branching node represents a *temporally feasible*, *resource infeasible* partial schedule. The search proceeds by checking the presence of a Minimal Critical Set (MCS, see Section II-C) and testing all possible ways to resolve it (i.e. by adding precedence constraints). Search stops when the augmented graph contains no MCS, so that the capacity of every resource is not exceeded *in any on-line generated schedule*. Details on the branching process are given in Section IV-D. Our main procedure finds a partial schedule satisfying all problem constraints: optimization is performed by progressively tightening a threshold on the maximum allowed expected completion time, according to a binary search scheme.

The critical difficulties are: (1) checking the feasibility of temporal constraints; (2) computing the expected completion time; (3) checking and enforcing resource feasibility.

Maintaining Temporal Feasibility: For the temporal constraints to be satisfied as from Definition 1, the on-line scheduler must be able to assign valid start times based on the observed durations of predecessor tasks. This matches the definition of Dynamic Controllability from Section II-D and motivated the choice of Simple Temporal Networks with Uncertainty for the temporal model; this is presented in Section IV-A.

Computing the Expected Completion Time: Exact computation of the expected completion time has exponential complexity and requires probability distributions for task durations (see Section II-C). In this work, we use a very simple approximation as a proxy for the expected value, described in Section IV-B.

Checking and Enforcing Resource Feasibility: We check the feasibility of resource constraints by detecting the presence of MCSs; this is done in polynomial time by using Graph Flow theory results. Moreover, given the effectiveness of resource constraint filtering in Constraint Programming (see Section II-B), we enable some constraint propagation by coupling the temporal model with a pair of classical Constraint Programming scheduling models. Details are given in Section IV-C.

A. The time model

The adopted temporal model consists of a constraint based formulation of the STNU formalism[6]; we rely on constraint propagation to enforce consistency. The model provides the following building blocks:

[6]With the restriction that the time bounds must be all non-negative.

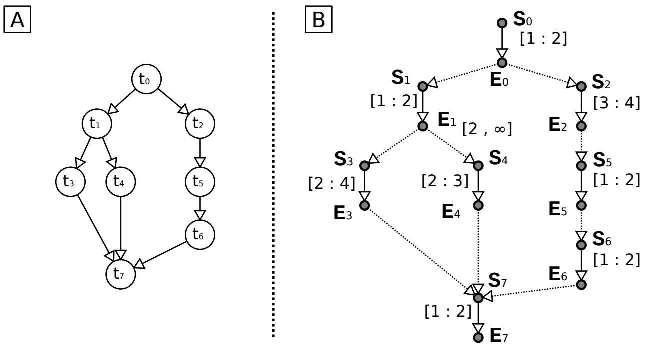


Fig. 3. Temporal model for the Problem of Figure 1; time lags are $[0, \infty]$ when not specified.

- *Event variables* ($\mathtt{T_i}$), associated to events τ_i (such as a task start or end); their domain represents the time span where τ_i can occur.
- *Free constraints* ($\mathtt{T_i} \xrightarrow{[a,b]} \mathtt{T_j}$), meaning that the domains of $\mathtt{T_i}$ and $\mathtt{T_j}$ must allow τ_j to occur θ time units after τ_j, for *at least* a value $\theta \in [a, b]$;
- *Contingent constraints* ($\mathtt{T_i} \xrightarrow{[a:b]} \mathtt{T_j}$), meaning that the domains of $\mathtt{T_i}$ and $\mathtt{T_j}$ must allow τ_j to occur θ time units after τ_j, for *every* value $\theta \in [a, b]$;

For all constraints $0 \leq a \leq b$ must hold; events may be forced to occur simultaneously by setting $a = b = 0$. Event variables, connected by directional binary constraints form a directed graph.

The above elements are sufficient to represent a variety of time constraints; in particular, a temporal model of the problem at hand can be built by:

1) introducing two event variables $\mathtt{S_i}, \mathtt{E_i}$ for the start and the end of each task t_i, respectively with domain $[est(t_i), \infty]$ and $[0, let(t_i)]$;
2) adding a free constraint $\mathtt{E_i} \xrightarrow{[\delta_{ij}, \Delta_{ij}]} \mathtt{S_j}$ for each arc (t_i, t_j) in the Task Graph;
3) adding a contingent constraint $\mathtt{S_i} \xrightarrow{[d_i:D_i]} \mathtt{E_i}$ for each task t_i in the Task Graph.

Where we recall d_i/D_i is the minimum/maximum duration for task t_i and δ_{ij}/Δ_{ij} is the minimum/maximum time lag for arc (t_i, t_j). Figure 3B shows the temporal model for the project graph in Figure 1 (reported to ease the reader). For each activity t_i in the graph, start/end event variables (resp. $\mathtt{S_i}, \mathtt{E_i}$) are linked by contingent constraints (solid arcs) with $a = d_i$ and $b = D_i$. Free constraints are used to represent inter task precedence relations (dotted arcs); in the example we have $a = 2, b = \infty$ for (t_1, t_4) and $a = 0$ and $b = \infty$ for all other arcs. Observe the presented temporal model can easily capture more complex temporal relations.

1) Dynamic Controllability via Constraint Propagation: Constraint propagation and filtering algorithm are in charge of maintaining the network dynamically controllable. In particular, dynamic controllability holds if and only if, for each free/contingent constraint between $\mathtt{T_i}$ and $\mathtt{T_j}$ and each value in the domain of $\mathtt{T_i}$, then the domain of $\mathtt{T_j}$ contains enough values for the constraint to be satisfied; any value in the domain of $\mathtt{T_j}$ not strictly required is useless and should be pruned.

In detail, the domain of each event variable $\mathtt{T_i}$ is specified by means of 4 values, namely $s_p(\mathtt{T_i})$, $s_o(\mathtt{T_i})$, $e_o(\mathtt{T_i})$, $e_p(\mathtt{T_i})$. Values $s_p(\mathtt{T_i})$ and $e_p(\mathtt{T_i})$ delimit the so-called *possible span* and specify the time interval where the event τ_i has some chance to take place; formally, let Σ be the set of possible execution states σ (i.e. combination of task durations and on-line scheduler decisions):

Definition 2 (Possible Span): The possible span of time variable $\mathtt{T_i}$ associated to time event τ_i is the interval $[s_p, e_p]$ such that $\forall\theta$ in the span:

$$\exists\sigma \in \Sigma \text{ such that } \tau_i \text{ occurs at time } \theta \tag{1}$$

Conversely, Values $s_o(\mathtt{T_i})$ and $e_o(\mathtt{T_i})$ bound the so-called *obligatory span*; if an event is forced to occur out of its *obligatory span* (i.e. if $s_p > e_o$ or $e_p < s_o$) dynamic controllability is compromised. Formally:

Definition 3 (Obligatory Span): The obligatory span of time variable $\mathtt{T_i}$ associated to time event τ_i is the smallest interval $[s_o, e_o]$ such that:

$$\begin{aligned}\forall\sigma \in \Sigma,\ \tau_i \text{ does not occur in } [s_o, e_o]\\ \Leftrightarrow\ \text{no dyn. controllability}\end{aligned} \tag{2}$$

Overall, the following theorem holds:

Theorem 1: Dynamic controllability holds if and only if $s_p \leq s_o \leq e_o \leq e_p$.

Proof: As for Definition 3, dynamic controllability implies the existence of an execution state σ such that τ_i occurs in $[s_o, e_o]$. Since the obligatory span is the *smallest* interval for which condition (2) holds, we have that $[s_o, e_o] \subseteq [s_p, e_p]$. Thus dynamic controllability is violated iff $[s_p, e_p] \cap [s_o, e_o] = \emptyset$, i.e. if $[s_p, e_p]$ or $[s_o, e_o]$ become empty ($s_p > e_p$ or $s_o > e_o$). ■

For an event variable not involved in any precedence constraint, we have $s_p = s_o$ and $e_o = e_p$; user defined release times and deadlines directly constrain s_p and e_p values, while s_o, e_o are only indirectly affected (i.e. to maintain $[s_o, e_o] \subseteq [s_p, e_p]$).

Precedence constraints affect variable domains, as depicted in Figure 4. Values s_p and e_p delimit the region where each τ_i *may* occur at run-time: for example τ_1 (corresponding to variable $\mathtt{T_1}$) can first occur at time 10, if τ_0 occurs at 0 and the constraint has duration 10; similarly τ_2 can first occur at 20 as at least 10 time units must pass between τ_1 and τ_2 due to the precedence constraint. As for the upper bounds, τ_2 cannot occur after time 60 or there would be a value $\theta \in [10, 20]$ with no support in the domain of τ_3; conversely, τ_1 can occur as late as time 50, since there is *at least* a value $\theta \in [10, 20]$ with a support in the domain of τ_2. Concerning bounds on the obligatory region, note that if τ_1 is *forced* to occur before time 20 the network is no longer dynamic controllable, as in that case there would not be sufficient time between τ_0 and τ_1. Similarly, τ_2 cannot be forced to occur later than time 60 or

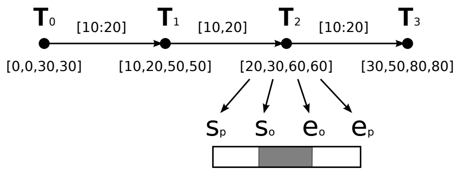


Fig. 4. The 4 value domain of time event variables

there would be a value $\theta \in [10, 20]$ such that the precedence constraint between τ_2 and τ_3 cannot be satisfied.

Dynamic Controllability can be enforced by iterative application of proper filtering rules for all free and contingent constraints, until a fix-point is reached. For a free constraint $\mathtt{T_i} \xrightarrow{[a,b]} \mathtt{T_j}$, the filtering rules are:

$$\begin{aligned}
s_p(\mathtt{T_j}) < s_p(\mathtt{T_i}) + a \quad&\Rightarrow\quad s_p(\mathtt{T_j}) = s_p(\mathtt{T_i}) + a && (3)\\
s_p(\mathtt{T_i}) < s_p(\mathtt{T_j}) - b \quad&\Rightarrow\quad s_p(\mathtt{T_i}) = s_p(\mathtt{T_j}) - b && (4)\\
s_o(\mathtt{T_j}) < s_o(\mathtt{T_i}) + a \quad&\Rightarrow\quad s_o(\mathtt{T_j}) = s_o(\mathtt{T_i}) + a && (5)\\
s_o(\mathtt{T_i}) < s_o(\mathtt{T_j}) - b \quad&\Rightarrow\quad s_o(\mathtt{T_i}) = s_o(\mathtt{T_j}) - b && (6)\\
e_o(\mathtt{T_j}) > e_o(\mathtt{T_i}) + a \quad&\Rightarrow\quad e_o(\mathtt{T_j}) = e_o(\mathtt{T_i}) + a && (7)\\
e_o(\mathtt{T_i}) > e_o(\mathtt{T_j}) - b \quad&\Rightarrow\quad e_o(\mathtt{T_i}) = e_o(\mathtt{T_j}) - b && (8)\\
e_p(\mathtt{T_j}) > e_p(\mathtt{T_i}) + a \quad&\Rightarrow\quad e_p(\mathtt{T_j}) = e_p(\mathtt{T_i}) + a && (9)\\
e_p(\mathtt{T_i}) > e_p(\mathtt{T_j}) - b \quad&\Rightarrow\quad e_p(\mathtt{T_i}) = e_p(\mathtt{T_j}) - b && (10)
\end{aligned}$$

In practice, $s_p(\mathtt{T_i})+a$ is a lower bound for $s_p(\mathtt{T_j})$, $s_p(\mathtt{T_j})-b$ is a lower bound for $s_p(\mathtt{T_i})$ and so on. The rules need to be triggered only when a domain change occurs. In the following, we discuss the soundness of rules (3) and (4); remaining rules can be motivated in a similar fashion.

Discussion of Rule (3)*:* Event τ_i cannot occur earlier than $s_p(\mathtt{T_i})$ (as by Definition 2). Since at least a time units must elapse between τ_i and τ_j, then event τ_j *has no chance* to occur earlier than $s_p(\mathtt{T_i}) + a$. ■

Discussion of Rule (4)*:* Event τ_j cannot occur earlier than $s_p(\mathtt{T_j})$. Since at most b time units must elapse between τ_i and τ_j, then event τ_i *is not allowed* to occur earlier than $s_p(\mathtt{T_j}) - b$. ■

Dynamic Controllability on a contingent constraint $\mathtt{T_i} \xrightarrow{[a:b]} \mathtt{T_j}$ is enforced by application of an analogous set of rules:

$$\begin{aligned}
s_p(\mathtt{T_j}) < s_p(\mathtt{T_i}) + a \quad&\Rightarrow\quad s_p(\mathtt{T_j}) = s_p(\mathtt{T_i}) + a && (11)\\
s_p(\mathtt{T_i}) < s_p(\mathtt{T_j}) - a \quad&\Rightarrow\quad s_p(\mathtt{T_i}) = s_p(\mathtt{T_j}) - a && (12)\\
s_o(\mathtt{T_j}) < s_o(\mathtt{T_i}) + b \quad&\Rightarrow\quad s_o(\mathtt{T_j}) = s_o(\mathtt{T_i}) + b && (13)\\
s_o(\mathtt{T_i}) < s_o(\mathtt{T_j}) - b \quad&\Rightarrow\quad s_o(\mathtt{T_i}) = s_o(\mathtt{T_j}) - b && (14)\\
e_o(\mathtt{T_j}) > e_o(\mathtt{T_i}) + b \quad&\Rightarrow\quad e_o(\mathtt{T_j}) = e_o(\mathtt{T_i}) + b && (15)\\
e_o(\mathtt{T_i}) > e_o(\mathtt{T_j}) - b \quad&\Rightarrow\quad e_o(\mathtt{T_i}) = e_o(\mathtt{T_j}) - b && (16)\\
e_p(\mathtt{T_j}) > e_p(\mathtt{T_i}) + b \quad&\Rightarrow\quad e_p(\mathtt{T_j}) = e_p(\mathtt{T_i}) + b && (17)\\
e_p(\mathtt{T_i}) > e_p(\mathtt{T_j}) - b \quad&\Rightarrow\quad e_p(\mathtt{T_i}) = e_p(\mathtt{T_j}) - b && (18)
\end{aligned}$$

Rule (11) is analogous to rule (3) for free constraints and so on; however, the time distance between the events is not under user control in this case, resulting in different rules for the possible and the obligatory span. Some discussion on soundness of rules (12) and (13) is reported in the following; similar reasoning applies to the remaining ones.

Rule (12)*:* Event τ_j cannot occur earlier than $s_p(\mathtt{T_j})$; since the contingent constraint coud assume any value θ in the interval $[a, b]$ (including a), then τ_i *is not allowed* to occur earlier than $s_p(\mathtt{T_j}) - a$. ■

Rule (13)*:* Let θ be the time instant when τ_i occurs; then, for dynamic controllability to hold, the domain of τ_j must have enough flexibility to accommodate every duration θ which Nature could assign to the constraint; in other words, τ_j cannot be forced to occur before $\theta + b$. The minimum θ for which dynamic controllability of $\mathtt{T_i}$ is not compromised is $s_o(\mathtt{T_i})$, hence $s_o(\mathtt{T_i}) + b$ provides a valid bound for $s_o(\mathtt{T_j})$. ■

The described constraint model allows testing *in constant time* whether a contingent or free constraint can be consistently added with the current network; this can be done by simply checking if the application of the corresponding filtering rules would violate the condition from Theorem 1.

B. Expected Completion Time Approximation

The cost function for our optimization approach is a proxy for the expected completion time of the application; namely, this is the completion time corresponding to the scenario where all tasks take their average case duration $\bar{d}_i$. This is achieved by building an accessory temporal model, containing:

1) two event variables $\mathtt{S}'_\mathtt{i}, \mathtt{E}'_\mathtt{i}$ for the start and the end of each task t_i, respectively with domain $[est(t_i), \infty]$ and $[0, let(t_i)]$;
2) a free constraint $\mathtt{E}'_\mathtt{i} \xrightarrow{[\delta_{ij}, \Delta_{ij}]} \mathtt{S}'_\mathtt{j}$ for each arc (t_i, t_j) in the Task Graph;
3) a free constraint $\mathtt{S}'_\mathtt{i} \xrightarrow{[\bar{d}_i, \bar{d}_i]} \mathtt{E}'_\mathtt{i}$ (i.e. with constant duration) for each task t_i in the Task Graph;

Observe the accessory model has free constraints only; at search time, whenever a constraint is added between variables $\mathtt{E_i}$ and $\mathtt{S_i}$, a twin constraint is added between $\mathtt{E}'_\mathtt{i}$ and $\mathtt{S}'_\mathtt{i}$. The expected completion time approximation is given by the maximum $e_p(\mathtt{E}'_\mathtt{i})$ value (see Figure 2B, where the approximation value is 9.5).

C. Handling Resource Constraints

1) Enabling `cumulative` *Constraint Propagation:* Resource restrictions are handled in Constraint based scheduling by means of resource constraints (such as `cumulative`) and related filtering algorithms. The approach is *very* effective, but usual filtering algorithms do not apply to uncertain durations (while variable, *user decided* durations are supported).

We managed to enable constraint propagation by chaining pairs of tasks with constant or user decided duration to specific spans. The key point is being able to map tasks in our temporal model (i.e. pairs of event variables connected by a contingent constraint) to integer domain variables used in constraint based scheduling (see Section II-B).

Let $\mathtt{S_i}$, $\mathtt{E_i}$ be the event variables representing the start and end of a task t_i; let $\mathtt{S_i} \xrightarrow{[d_i:D_i]} \mathtt{E_i}$ be the contingent constraint modeling the task duration. Observe that:

1) rules (3)-(10) for contingent constraints apply on the *possible* intervals $[s_p(\mathtt{S_i}), e_p(\mathtt{S_i})]$ and $[s_p(\mathtt{E_i}), e_p(\mathtt{E_i})]$ the same kind of filtering which would be applied by a *user decided* duration constraint, with duration ranging in $[d_i, D_i]$;
2) rules (11)-(18) for contingent constraints apply on the *obligatory* intervals $[s_o(\mathtt{S_i}), e_o(\mathtt{S_i})]$ and $[s_o(\mathtt{E_i}), e_o(\mathtt{E_i})]$ the same kind of filtering which would be applied by a fixed duration constraint, with duration equal to D_i.

Hence we can chain the event variables to pairs of classical integer variables, namely $\mathtt{S}^*_\mathtt{i}$, $\mathtt{E}^*_\mathtt{i}$ and $\mathtt{S}^{**}_\mathtt{i}$, $\mathtt{E}^{**}_\mathtt{i}$. Variable $\mathtt{S}^*_\mathtt{i}$ is synchronized to the interval $[s_p(\mathtt{S_i}), e_p(\mathtt{S_i})]$ by simple chaining constraints, while variable $\mathtt{S}^{**}_\mathtt{i}$ is chained to $[s_o(\mathtt{S_i}), e_o(\mathtt{S_i})]$ and so on. Then we post the following constraints:

$$\mathtt{E}^*_\mathtt{i} = \mathtt{S}^*_\mathtt{i} + \mathtt{D}^*_\mathtt{i} \qquad\qquad \mathtt{E}^{**}_\mathtt{i} = \mathtt{S}^{**}_\mathtt{i} + D_i$$

where $\mathtt{D}^*_\mathtt{i}$ is an extra integer variable, ranging in $[d_i..D_i]$. Variables $\mathtt{S}^*_\mathtt{i}, \mathtt{E}^*_\mathtt{i}$ define a classical task in Constraint Programming with variable, user decided, duration; variables $\mathtt{S}^{**}_\mathtt{i}, \mathtt{E}^{**}_\mathtt{i}$ identify a second CP task with fixed duration, equal to D_i. Usual `cumulative` constraints and related filtering algorithms can be taken off-the-shelf and applied to perform propagation. Specifically, we use timetable, edge-finder and balance filtering (see [18]).

2) Minimal Critical Set Detection: One of the key difficulties with complete search based on MCS branching is how to detect and choose conflict sets to branch on; this stems from the fact that the number of MCS is in general exponential in the size of the Task Graph, hence complete enumeration incurs the risk of combinatorial explosion.

We argue that just *detecting* a possible conflict does not require MCS enumeration; in detail, we provide a method to detect (non necessarily minimal) Critical Sets (CS) by solving a minimum flow problem. By definition, a temporal network is MCS-free if and only if it is CS-free. Moreover, once a CS is available, an MCS can then be obtained by simply removing tasks.

Note that an MCS represents a *possibly occurring* resource conflict: it may be the case that no over-usage arises for some duration instantiations. According to our method, the conflict must be nevertheless resolved to remove scheduling anomalies, resulting in a partially overconstrained schedule. Unfortunately, the only alternatives are either to use a dynamic scheduling heuristic (incurring the risk of anomalies) or to switch between many precomputed schedules depending on the observed durations (requires an exponentially large scheduling table).

Detecting the presence of a Critical Set with our method has *polynomial complexity* and *does not loose completeness*. As a drawback, the identified CS set is not in general well suited to open a choice point. We cope with this issue via a simple conflict minimization step, guided by a greedy heuristic. A sketch of the adopted MCS detection strategy is shown in Algorithm 1. In the following, each of the steps will be described in deeper detail; the adopted criterion to

Algorithm 1 MCS detection method

```
1: set best MCS so far ($best \leftarrow \emptyset$)
2: for $r_k \in R$ do
3:     find a conflict set $cs$ by solving a minimum flow problem
4:     if weight of $cs$ is higher than $c_k$ then
5:         refine $cs$ to a Minimal Critical Set $mcs$ via greedy minimization
6:         if $mcs$ is better than $best$ then
7:             $best = mcs$
8: return $best$
```

evaluate the quality of a Critical Set will be given as well. As a starting point, observe that:

Statement 1: Since the tasks in a CS must have the chance to overlap, they always form a *stable set* (or independent set) on the Task Graph, further annotated with all precedence constraint added during search or detected by time reasoning.

We augment such graph with fake source and sink nodes; this is referred to as *resource graph*. Formally, the resource graph is a pair $\langle T \cup \{source, sink\}, A_R\rangle$, where $(a_i, a_j) \in A_R$ iff the arc:

1) is in the original graph (i.e. $(a_i, a_j) \in A$);
2) is added by search (i.e. $(a_i, a_j) \in A^*$);
3) is detected by complete or incomplete time reasoning[7]; in particular, we check if $e_p(\mathtt{E_i}) \le s_p(\mathtt{S_i})$;
4) connects the source to any node (i.e. $a_i = source$);
5) connects any node to the sink (i.e. $a_j = sink$).

Figure 5.A shows the resource graph for the TG in Figure 1, at the root of the search tree (i.e. when no additional precedence constraint has yet been posted), assuming a global deadline of 12 time units has been assigned to every task and constraint propagation has updated the domains in the temporal model. Note the (dashed) arcs (t_2, t_3) and (t_2, t_4) are not in the original graph: one can check that they are detected by time reasoning, since $e_p(t_2) = 12 - D_7 - D_6 - D_5 = 4$ and $s_p(t_3) = s_p(t_4) = d_0 + d_1 = 4$, hence $e_p(t_2) \le s_p(t_3)$, $e_p(t_2) \le s_p(t_4)$.

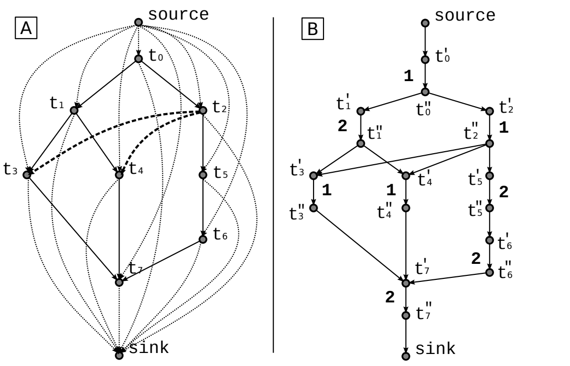


Fig. 5. A) Resource Graph for the TG in Figure 1; B) Resource Graph with split nodes, to allow the application of Edmonds-Karp algorithm (arcs from/to source and sink are omitted).

Given a target resource r_k, let us assign to each task t_i the requirement rq_{ik} as a weight; then a stable set S is a CS iff $\sum_{t_i \in S} rq_{ik} > c_k$. Therefore:

[7]Adding detected arcs is not essential for the method correctness, but prevents the algorithm from finding some MCS with no chance to occur in practice due to bounds on task durations.

Statement 2: We can check the existence of a MCS on a resource r_k by finding the maximum weight stable set on the resource graph (Figure 5A) and checking its total weight.

This can be done by solving a minimum flow problem; namely, we can read each *resource* requirement as a *flow* requirement, and then try to route the minimum possible amount of flow from *source* to *sink*, so that all flow requirements are satisfied (see [39]).

The problem can be solved by starting from an initial feasible flow and performing iterative reductions with the inverse Edmonds-Karp's algorithm [40], with complexity $O(|A_R| \cdot \mathcal{F})$ (where $\mathcal{F}$ is the value of the initial flow – note the algorithm is in fact pseudo-polynomial). Once the final flow is known, activities in the S/T cut[8] form the maximum weight independent set.

a) Solving the Minimum Flow Problem: Since Edmonds-Karp's algorithm only allows requirements on arcs, each task t_i must be split into two connected subnodes t'_i, t''_i, so that the minimum flow requirement rq_{ik} is effectively assigned to the arc (t'_i, t''_i). Every arc $(t_i, t_j) \in A_R$ is then converted into an arc (t''_i, t'_j) and assigned minimum flow requirement 0. Figure 5B shows the modified graph corresponding to the Resource Graph of Figure 5A; flow requirements label each (t'_i, t''_i); arcs connecting the source and the sink node are omitted (for sake of clarity) in case they can be deduced by the transitive property.

The initial flow at the root of the search tree is computed by routing rq_{ik} flow units for each task t_i:

1) along arc $(source, t'_i)$;
2) along arc (t'_i, t''_i), to satisfy the requirement;
3) along arc $(t''_i, sink)$.

Figure 6A shows an example of initial feasible flow, computed as described; flow values appear in dashed boxes and node names are omitted for sake of clarity. The quality of this initial solution is usually pretty poor; since the value of the initial flow has a direct impact on the Edmond-Karp algorithm runtime, one may want to devise a better algorithm. However, observe that during search arcs can

[8]I.e. all arcs (t'_i, t''_i) such that t'_i is not connected to the graph sink (and t''_i is not connected to the source) via non-saturated arcs.

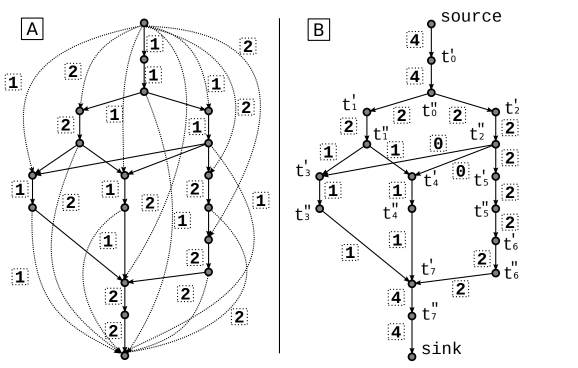


Fig. 6. A) Initial flow at the root node, for the TG in Figure 1; B) Minimum flow at the root node for the same graph

only be *added* to the resource graph; since adding arcs may only *reduce* the flow, we have:

Statement 3: The minimum flow at a node in the search tree is a feasible initial flow for all its child nodes

Furthermore, it is likely a very good starting flow; for example, Figure 6B shows the minimized flow computed from the initial value in Figure 6A; one can check the overall value is very low, compared to the initial one (4 versus 12). We therefore prime the process with the mentioned (poor quality) initial flow, then perform minimization at the root of the search tree and carry on the minimized flow as initial feasible flow for the child nodes; the process is repeated and the flow value is restored upon backtracking. This allows a very efficient, incremental solution of the minimum flow problem. If the flow value exceeds the capacity c_k, a Critical Set has been identified; otherwise, the graph contains no CS with respect to resource r_k.

b) Critical Set Evaluation and Minimization: Once a Critical Set has been identified, a number of issues still have to be coped with; namely (1) the detected CS is not necessarily minimal and (2) the detected CS does not necessarily yield a good choice point. Note that branching on non-minimal CS can result in exploring unnecessary search paths.

While issue (1) can be simply dealt with by removing tasks from the CS, issue (2) requires the definition of a criterion to rank different CS. Similarly to [31] we base our ranking on *preserved space*, which evaluates the amount of flexibility retained if a new precedence constraint is posted between tasks t_i and t_j. In detail, we have to distinguish between *preserved possible span* and *preserved obligatory span*; the former can be computed as:

$$\textit{preserved}_p(t_i, t_j) = \begin{cases} 1 \text{ if } s_p(\mathtt{S_j}) \geq e_p(\mathtt{E_i}) \\ 0 \text{ if } s_p(\mathtt{E_i}) \geq s_p(\mathtt{S_j}) \\ \dfrac{B - C_{min} - C_{max}}{2A} \text{ otherwise} \end{cases}$$

with:

$$\begin{aligned} A &= (e_p(\mathtt{S_j}) - s_p(\mathtt{S_j}) + 1) \times (e_p(\mathtt{E_i}) - s_p(\mathtt{E_i}) + 1) \\ B &= (e_p(\mathtt{S_j}) - s_p(\mathtt{E_i}) + 1) \times (e_p(\mathtt{S_j}) - s_p(\mathtt{E_i}) + 2) \\ C_{min} &= \max\{0, s_p(\mathtt{S_j}) - s_p(\mathtt{E_i})\} \times (s_p(\mathtt{S_j}) - s_p(\mathtt{E_i}) + 1) \\ C_{max} &= \max\{0, e_p(\mathtt{S_j}) - e_p(\mathtt{E_i})\} \times (e_p(\mathtt{S_j}) - e_p(\mathtt{E_i}) + 1) \end{aligned}$$

Note the quantity $\textit{preserved}_p(t_i, t_j)$ is always between 0 and 1. Formulas for the preserved obligatory span $\textit{preserved}_o(t_i, t_j)$ can be obtained by replacing s_p with s_o and e_p with e_o. We define the preserved possible/obligatory span of a Conflict Set as the cumulative preserved spans of the precedence constraints which can be posted to resolve it. Formally:

Definition 4: The preserved *possible* span of a Conflict Set CS is given by:

$$\textit{preserved}_p(CS) = \sum_{\substack{t_i, t_j \in CS \\ \mathtt{E_i} \rightarrow \mathtt{S_j} \textit{ is consistent}}} \textit{preserved}_p(t_i, t_j)$$

The definition of the preserved *obligatory* span of a CS is analogous. According to the First-Fail principle [41], we give preference to Conflict Sets having the *smallest* preserved obligatory span, while the *smallest* preserved possible span is considered in case of ties.

Our CS minimization procedure simply consists of iteratively removing the task yielding the CS with the highest quality; the process stops when no further task can be removed (i.e. the CS has become minimal). This specific method was chosen from a set of experimented minimization algorithms due to its efficiency and reasonably good effectiveness.

D. The branching process

Once a MCS is selected at each search step, then a choice point is open; in detail, let RS be the list of all ordered pairs of tasks (t_i, t_j) in the CS; ideally, each pair corresponds to a possible resolver. We discard from RS pairs such that the corresponding resolver cannot be consistently added given the current state of the temporal model (this can be checked in constant time as described in Section IV-A1). Moreover, we discard further pairs by applying the simplification method from [31]. Those techniques enable the algorithm to focus on meaningful resolvers and reduce the number of search nodes.

Then, the remaining pairs in RS are ranked by *decreasing* preserved span of the corresponding resolver (similarly to [18]); the preserved obligatory span is given priority, while the preserved possible span is used to break ties. Let t_{i_k}, t_{j_k} be the chosen pair, then binary choice point is opened; on the left branch, a new free constraint $\mathtt{E_{i_k}} \xrightarrow{[0,\infty]} \mathtt{S_{j_k}}$ is added to resolve the MCS. On the right branch, the same constraint is forbidden: this amounts to add the free constraint $\mathtt{S_{j_k}} \xrightarrow{[\epsilon,\infty]} \mathtt{E_{i_k}}$, where the minimum time lag ϵ (arbitrarily low) enforces a strict inequality. Note that, while precedence constraints added to resolve the MCS actually have to be taken into account by the on-line scheduler, forbidden precedence constraints represents *deductions*; as a consequence, they do not need to be explicitly enforced by the on-line scheduler.

V. Experimental Evaluation

We implemented our approach on top of the commercial CP solver IBM-ILOG Solver 6.7. The test problem consists in scheduling applications on a platform template consisting of clustered execution engines, connected by limited bandwidth ports. Our scheduling routine is invoked after a preliminary heuristic resource mapping step, performed via a state of the art Graph Partitioning algorithm.

A first set of experiments has the objective to assess the efficiency of the approach; next the quality of the provided solution is evaluated via Monte Carlo simulation and compared against a fixed-priority scheduling method and a simple on-line FIFO scheduler.

The adopted application and platform models are described in details in Section V-A; the mapping method is discussed in Section V-B; the details of the evaluation

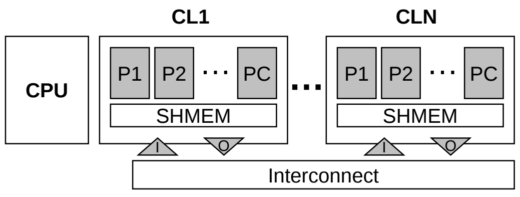


Fig. 7. (a) Target Platform; (b) DMA carried communication; the shaded and the white part of each task represent minimal and maximal durations

process are given in Section V-C. Finally efficiency tests and the comparison results are provided in Sections V-D and V-E.

A. Application and Platform Models

We consider a common template for embedded Multi Processor Systems on Chip (MPSoCs), featuring 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 often feature private memory subsystems [1], [2], [42]. Explicit communication is efficiently supported (e.g. with specialized instructions or DMA), but uniform shared memory abstraction is either unavailable or it implies significant overheads[9].

We model the heterogeneous MPSoC as shown in Figure 7a. 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 n execution engines is considered a resource with capacity $c_k = n$ and 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.

We model the target application prior to mapping as a set of tasks t_i, with dependencies due to data communication. A formal description according to the definition and the notation in Section III will be available *after the mapping stage*. 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 has non-zero latency and requires a known fraction $b(t_i, t_j)$ of the total bandwidth available at the output communication port (triangle marked with O) of 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. Input and output ports can be seen as resources with capacity equal to the provided bandwidth.

[9]This can be emulated in software by propagating via copy operations all private memory modifications onto external shared memory [1].

Tasks and inter-cluster communications have uncertain durations, bounded by known values; let those be respectively d_i^{min}, d_i^{max} and $d_{ij}^{min}, d_{ij}^{max}$; for our experimentation, we assume the average durations values to be respectively $(d_i^{min} + d_i^{max})/2$ and $(d_{ij}^{min} + d_{ij}^{max})/2$. We assume all duration bound to be integer.

B. Computing a Resource Allocation

Prior to the scheduling, tasks are heuristically mapped to the available resources; once a task-to-cluster allocation is given, Input and Output ports are implicitly assigned to inter-cluster communications. We recall that, since the focus of this paper is on the scheduling algorithm, our objective is just to compute a *reasonably good* mapping. Given the large impact of resource allocation on the final schedule quality, a different partitioning technique could replace the one describe in the following, with a chance to obtain better results.

Tasks are mapped to clusters by means of the graph partitioning algorithm METIS [43]. Namely, tasks are partitioned in a number of sets equal to the number of clusters, with the double objective to 1) balance the computational load and 2) minimize inter-cluster communication. We consider the estimated average execution time as a measure of the computation load of a task; therefore, if S is a set of tasks, the overall set load is the sum of the average execution times.
The METIS algorithm forces the difference between the maximum and minimum workload to be within a small specified bound (6% of the smallest weight) and, as a secondary objective, minimizes the sum of the bandwidth requirements of arcs connecting tasks in different sets.

The output of the partitioning algorithm is a collection of task sets, one per each cluster; in particular, let $pe(t_i)$ be the index of the computational engine where task t_i is mapped. Based on this information, a Task Graph as from Section III is built by introducing, for each task in the original application, a node with $d_i = d_i^{min}, D_i = d_i^{max}$ and $\bar{d}_i = (d_i^{min} + d_i^{max})/2$. Intra-cluster communication correspond to arcs (t_i, t_j). Then an extra task t'_{ij} is introduced for each inter-cluster communication (t_i, t_j); task t'_{ij} represents a data transmission activity and uses $b(t_i, t_j)$ units of the *Output Port* of cluster $pe(t_i)$ and of the *Input Port* of cluster $pe(t_j)$. The communication task t'_{ij} has uncertain duration corresponding to that of the inter-cluster communication, i.e. in the interval $[d_{ij}^{min}, d_{ij}^{max}]$ and average value $(d_{ij}^{min}, d_{ij}^{max})/2$. Finally, the task is connected to t_i and t_j by means of arcs (t_i, t'_{ij}) and (t'_{ij}, t_j). All minimal time lags are 0 and all maximal time lags are ∞. This Task Graph is the actual input for our scheduling method.

C. The evaluation process

We perform an experimental evaluation on a benchmark consisting of synthetically generated Task Graphs, to be mapped on synthetic platforms. The choice was motivated by: (1) the need for a large number of graphs to provide

a reliable efficiency assessment of our solver[10]; (2) the intention to investigate the impact of different duration distributions on the expected completion time, in order to assess the accuracy of out approximate cost function.
As a drawback, run-time overhead is not taken into account; being the schedule statically computed and based on simple precedence relation, we expected the computational burden on the run-time layer to be fairly small.

In details, two set of instances (referred to as Group 1 and Group 2 in the following) were obtained by means of a flexible generator, designed to produce realistic Task Graphs[11]. Group 1 consists of TGs containing 20 to 70 nodes prior the mapping process; the number of outgoing arcs for each node (branching factor) is in the range 3-5 (sink nodes excluded); TGs in group 2 have fixed number of nodes prior to mapping (namely, 40) and branching factor from 2-4 to 6-8. For both groups maximum duration values are up to 2 times the minimum ones (both for tasks and data communications).

We consider two (abstract) target platforms (Platform A and Platform B, in the following); platform 1 has 16 single-thread accelerators, while platform 2 has 4 of them (quad-thread). Each accelerator has a single input port and a single output port; all ports are assumed to have uniform bandwidth. Each task in the original graph requires one execution thread; data communications require from 10% to 90% of the port bandwidth, in case the producer and the consumer are mapped to different accelerators.

The TGs are mapped to the target platforms as described in Section V-B, the resulting transformed graphs are the actual input for the evaluation process; this consists of two distinct phases; in first place, we use our method to identify the tightest global hard real-time constraint the graph can meet; in a second step, we compute a schedule with minimum expected completion time, guaranteed to meet the identified deadline.

D. Finding the tightest achievable deadline

We assume the applications are subject to a global hard real-time constraint; the identification of a non trivial deadline value is itself a challenging task and can be used to provide a first assessment of the method efficiency.

The identification proceeds by performing binary search over the possible deadline values (in particular, we consider integer values only); the approach requires to repeatedly run an NP-hard feasibility test, which is provided by our tree search solver. In this phase, *the cost function is disregarded* and the solver stops as soon as it finds a partial schedule guaranteed to meet the deadline value currently being tested; note the problem is still NP-hard. The output of the process are a lower- and an upper-bound (lb, ub) on the tightest achievable deadline and a partial schedule guaranteed to meet the upper bound value. Obviously, if $lb = ub$

[10]The production and characterization of a sufficient number of real applications would take prohibitive time.

[11]The generator has been presented at the PST ICAPS2007 workshop and is available for download at http://ai.unibo.it/node/410.

	nodes before mapping	nodes before mapping	prec. rel.	time	time outs	gap
Platform A	20	41-49	17.70(3.98)	0.14(0.02)	0	0.00%
	30	56-66	25.44(5.46)	0.22(0.02)	1	0.32%
	40	75-82	31.20(5.64)	0.32(0.06)	0	0.00%
	50	93-103	50.30(10.69)	43.61(129.22)	0	0.00%
	60	110-119	64.20(8.92)	0.72(0.15)	0	0.00%
	70	118-128	61.30(6.03)	0.91(0.18)	0	0.00%
Platform B	20	29-36	11.40(4.05)	0.10(0.01)	0	0.00%
	30	41-52	14.90(5.28)	0.18(0.03)	0	0.00%
	40	54-60	17.30(6.63)	0.25(0.03)	0	0.00%
	50	65-78	31.20(12.81)	0.42(0.06)	0	0.00%
	60	78-86	46.00(10.41)	0.54(0.06)	1	1.53%
	70	86-96	42.70(10.08)	0.78(0.10)	0	0.00%

TABLE I
TIGHTEST DEADLINE IDENTIFICATION: RESULTS ON GROUP 1

the tightest deadline has been identified. Experiments are performed on an Intel Xeon X5570, 2.93GHz; being the problem NP-hard, a limit of 600 seconds was enforced over all runs.

Tables I and II show the results of this first evaluation, respectively for Group 1 and Group 2. Each row reports results for a group of 10 instances; we list the minimum and maximum number of nodes after the mapping process (within the group), the average number of added precedence relations in the final schedule, the time for the binary search process (in seconds), the number of timed out instances, and the bound gap (i.e. $^{ub-lb}/_{lb}$) in case the tightest deadline is not reached; standard deviations are between round brackets. The first column shows the initial number of nodes prior to mapping for Group 1 and the branching factor range for Group 2.

The increase in the number of nodes after the mapping process is due to the added inter-accelerator communication tasks: this is more relevant for Platform A, due to the smaller number of threads per accelerator. The number of added precedence relations is relatively large; in principle, this may result in a deterioration of the expected completion time and should be considered in the next evaluation phase. Despite the problem being NP-hard, most

	branching factor	nodes before mapping	prec. rel.	time	time outs	gap
Platform A	2-4	74-84	31.10(7.82)	2.21(5.68)	0	0.00%
	3-5	78-84	41.40(5.94)	0.37(0.05)	0	0.00%
	4-6	78-88	44.80(7.00)	0.44(0.14)	0	0.00%
	5-7	79-90	51.50(11.01)	0.46(0.20)	0	0.00%
	6-8	83-95	58.40(8.96)	0.51(0.12)	0	0.00%
Platform B	2-4	55-62	18.80(11.00)	0.26(0.04)	0	0.00%
	3-5	55-66	27.00(10.41)	0.30(0.05)	0	0.00%
	4-6	57-69	32.40(12.31)	0.33(0.07)	0	0.00%
	5-7	52-72	33.90(14.44)	0.36(0.17)	0	0.00%
	6-8	58-67	41.00(7.60)	0.35(0.06)	0	0.00%

TABLE II
TIGHTEST DEADLINE IDENTIFICATION: RESULTS ON GROUP 2

of the instances are solved to optimality in a fraction of second; the availability of resource constraint propagation (see Section IV-C1) plays a key role here, as disabling resource propagation raises the solution time by orders of magnitude. The efficiency of the MCS detection procedure improves the method scalability, since with growing number of nodes the number of Critical Sets in a graph can become prohibitive for complete enumeration.

A few instances in Group 1 (namely, two) could not be solved to optimality: even in this case, however, the bound gap is very low, meaning the produced schedule is still very good. Moreover, a feasible solution close from the final lower bound is usually available pretty early in the search process (1.6% of the bound after 0.07 seconds for the timed out instance in Group 1). Note that time-out instances are excluded from the computation of the mean solution time.

E. Expected completion time minimization

In the second stage of the evaluation, we tackle the minimization of the expected completion time. Here, only a subset of the benchmarks instances is considered (Group 2, branching factor 4-6); the tightest hard-deadline from the previous evaluation stage is enforced on each graph. Optimization is performed via binary search on the expected completion time proxy (see Section IV-B), while the deadline stays fixed throughout the process. Finally, the actual quality of the resulting schedule is assessed via Monte Carlo simulation.

PCP approaches are quite uncommon in the multiprocessor scheduling domain; hence, the main objective of this experimentation is to assess the viability of the proposed technique compared to traditional ones. For this purpose, our method is compared against a Fixed Priority Scheduling (FPS) approach with off-line optimization and a pure online FIFO scheduler.

FPS scheduler: The considered FPS method is a variant of the one presented in [44], based on Tabu-search and performing both mapping and scheduling. Since the focus of this paper is on pure scheduling, we force the mapping from Section V-B and use tabu-search only for priority assignment. Moreover, we have extended the approach to deal with non-unary resources[12]: in [44] there is no such issue since only single-thread processors are taken into account. Finally, we replaced per-resource priority lists in [44] with a single global one; this is necessary to avoid conflicting priority assignments, since tasks in our model can have multiple resource requirements[13]. Estimating the completion time with FPS requires more complex techniques compared to our approach. In [44] a clever approximation method is used to compute end time distributions for each task; the computation is fast enough to be embedded in the Tabu-search optimization, although (by far) slower than the simple proxy we adopt in this work; on the other hand, the accuracy of the estimate is much higher[14].

[12]E.g. the 4 thread accelerators in Platform B or the input/output ports.

[13]E.g. communication tasks require two distinct ports.

[14]Moreover, as a by product, the proposed approximation method in [44] allows the authors to tackle soft real-time constraints.

	Platform A					Platform B				
inst.	nodes	arcs	time	time outs	gap	nodes	arcs	time	time outs	gap
tg46-000	78	99	0.23	0	0.00%	57	78	0.2	0	0.00%
tg46-001	82	104	0.30	1	0.19%	60	82	0.26	0	0.00%
tg46-002	85	110	0.52	0	0.00%	65	90	0.3	0	0.00%
tg46-003	85	108	0.27	1	0.53%	63	86	0.29	0	0.00%
tg46-004	81	104	0.88	0	0.00%	58	81	0.28	0	0.00%
tg46-005	81	101	0.38	0	0.00%	63	83	0.33	0	0.00%
tg46-006	83	106	0.34	0	0.00%	65	88	0.23	0	0.00%
tg46-007	86	112	2.82	0	0.00%	69	95	0.6	0	0.00%
tg46-008	83	107	0.34	0	0.00%	67	91	0.34	0	0.00%
tg46-009	88	115	0.41	0	0.00%	66	93	0.32	0	0.00%

TABLE III
PERFORMANCE FOR EXPECTED COMPLETION TIME MINIMIZATION

An advantage of FPS scheduling over our method is the ability to always start the highest priority task as soon as it is possible. In PCP scheduling, the added precedence constraints may force a task to wait for an "artificial" predecessor, even if all the required data and resources are available. On the other side, FPS may cause scheduling anomalies, while the PCP schedules are by construction anomaly free.

FIFO scheduler: This is a pure on-line approach, requiring no off-line tuning; tasks are tentatively started as soon as they are ready, and enqueued in case the required resources are not available. Whenever a task ends, a waiting task with all required resources available is chosen for execution by following a FIFO order.

This scheduling strategy has the same advantages as FPS over PCP; moreover, priority is accorded to the enqueued tasks in a *dynamic* fashion, depending on the order the tasks got enqueued; this may lead to myopic decisions or to clever ones depending on the run-time task durations. As a consequence, FIFO scheduling is not theoretically dominated by either PCP nor FPS: the "best" scheduling strategy for a target benchmark can only be identified via experimental evaluation.

The tests are performed on an Intel Xeon X5570, 2.93GHz; a limit of 600 seconds was enforced for the PCP approach. Tabu-search optimization was configured[15] as described in [44] and allowed to run up to 4 hours per instance. The quality of the schedules is assessed by sampling task durations in each graph 10,000 times, then performing Monte Carlo simulation. On this purpose, the duration of each task t_i is assumed to follow a Normal distribution with mean ${}^{d_i+D_i}/_2$ and standard deviation ${}^{d_i+D_i}/_6$; the values are chosen so that 98% of the possible duration values are in the interval $[d_i, D_i]$.

Table III reports the performance for the expected completion time minimization with our approach; all instances have 40 tasks prior to mapping; the number of nodes and arcs after the mapping phase is reported under columns

[15]Tabu tenure: 4; max number of iterations: $4 \times \#nodes$; number of non-improving iterations before a diversification: $0.33 \times \#nodes$; frequency threshold for forcing a mode to become tabu: 0.6. The basic time interval for the completion time approximation is 2.

nodes and *arcs*. Columns *time*, *time outs* and *gap* are the same as in Table I and II. Once again, despite the problem is NP-hard, the optimum is identified very quickly; this is mainly due to the effectiveness of constraint propagation, the efficiency of the MCS detection procedure *and the simplicity of the expected completion time proxy*. Conversely, the 4 hours time limit was exceed by tabu-search in all but a few cases, due to the computational complexity of the completion time approximation. Two time-outs are reported on Platform A, but the final gap value is very low and a high quality solution is produced in just 0.14 seconds.

Table IV shows the outcome of the Monte Carlo simulation; thanks to the Central Limit Theorem, the completion time can be reasonably assumed to have Normal distribution[16]. The expected completion time can therefore be characterized in terms of mean (main number in each column) and standard deviation (between round brackets). Somehow surprisingly, the PCP approach obtains consistently better results in terms of mean completion time; the solution stability (i.e. standard deviation) is often better, but in this case there is no sharp dominance. Basically, for the considered benchmark *the unpredictable delay due to scheduling anomalies seems to be a far greater issue than the controlled waiting time due to artificial precedence relations*.

The gap is sensibly smaller for the less constrained Platform B: this matches the proposed explanation, since anomalies are due to limited resources and inter-task dependencies. To probe this conjecture, we have run experiments with uniformly distributed durations (hence, higher variance) and larger branching factor. As expected, higher variability tends to increase the performance gap, while higher branching factors make it smaller; however, the final outcome does not move too much and the tightness of the resource constraints seems to be main discriminating factor.

Figure 8 shows the sampled completion time distributions for graph *tg46-007* on Platform B; this is one of the cases where the performance difference between PCP and FPS scheduling is less prominent (yet sensible); as one can see, the probability distributions are (roughly) Normal and the mean value of PCP and FPS is quite close. More insight in the behavior is provided by Figure 9, showing for the same graph a scatter plot with the PCP and FPS completion times for the same realization of task durations. Observe that PCP and and FPS tend to perform well/badly on the same instances so that, despite the quite large overlapping between the probability distribution, PCP actually beats FPS in most cases. This behavior does not show up when the expected completion times are too close (i.e. *tg46-000 and tg46-005* on Platform B).

[16]As long as the graph is sufficiently large and scheduling anomalies are not too critical: in that case, a composition of Normal distributions would be more proper.

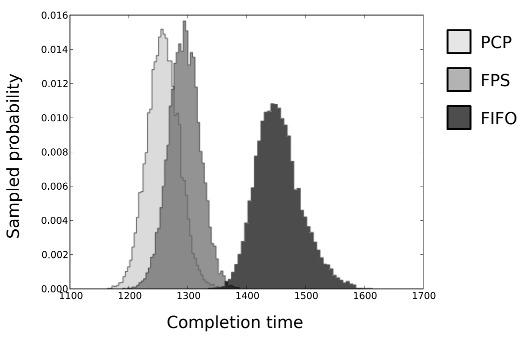


Fig. 8. Sampled completion time distribution for the instance tg46-005

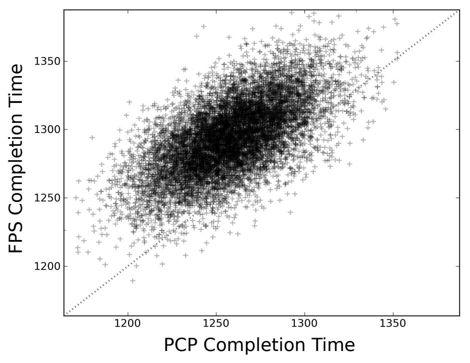


Fig. 9. Sampled completion time distribution for the instance tg46-005

		prec.	prior.	FIFO
Platform A	tg46-000	1,305.29 (29.89)	1,351.64 (42.35)	1,473.62 (49.71)
	tg46-001	1,566.63 (34.06)	1,787.99 (40.00)	1,870.38 (39.93)
	tg46-002	1,221.23 (26.54)	1,466.65 (31.32)	1,607.98 (41.38)
	tg46-003	1,515.85 (29.90)	1,661.15 (34.97)	1,797.56 (34.36)
	tg46-004	1,260.55 (18.21)	1,416.59 (19.53)	1,329.15 (17.82)
	tg46-005	1,806.69 (33.66)	1,955.63 (48.10)	2,010.18 (42.31)
	tg46-006	868.73 (24.92)	1,142.17 (36.07)	1,224.06 (47.37)
	tg46-007	1,377.57 (28.17)	1,456.27 (37.15)	1,454.50 (38.66)
	tg46-008	869.69 (27.30)	1,073.77 (45.76)	1,130.26 (28.44)
	tg46-009	1,098.90 (21.26)	1,231.61 (39.98)	1,304.77 (23.22)
Platform B	tg46-000	1,256.43 (29.99)	1,256.71 (30.05)	1,256.87 (29.48)
	tg46-001	1,496.09 (34.19)	1,577.15 (34.11)	1,543.03 (34.70)
	tg46-002	1,186.40 (26.00)	1,212.21 (28.67)	1,213.42 (27.77)
	tg46-003	1,356.53 (29.71)	1,450.60 (30.92)	1,401.59 (29.90)
	tg46-004	1,213.13 (20.53)	1,250.48 (17.54)	1,218.51 (20.35)
	tg46-005	1,696.53 (33.53)	1,705.39 (33.88)	1,698.91 (33.53)
	tg46-006	840.06 (25.25)	863.45 (25.93)	913.68 (38.65)
	tg46-007	1,258.31 (26.84)	1,294.27 (26.51)	1,323.61 (33.14)
	tg46-008	841.80 (27.47)	846.39 (24.29)	843.83 (25.24)
	tg46-009	1,014.62 (20.19)	1,043.63 (21.49)	1,015.09 (20.56)

TABLE IV
COMPLETION TIME VIA MONTE CARLO SIMULATION

Finally, since our approach relies on approximate execution times information to produced optimized schedules, we performed some tests with increased duration variability. Namely, we modified graphs *tg46-002* and *tg46-003* so that maximum duration values are up to 10 times the minimum ones. We evaluated PCP, FPS and FIFO schedules for Platform A. Interestingly, the performance gaps are preserved, with the PCP schedules being 30% shorter compared to FIFO and around 20% compared to FPS.

VI. CONCLUSION

We have developed a PCP based off-line schedule dataflow described applications with uncertain task durations. The method does not require probability distributions to be known and relies instead on simpler and cheaper-to-obtain information. The use of an expected makespan proxy as objective (rather than a more accurate approximation) and advanced Constraint Programming techniques makes the solution time very affordable for applications of realistic size; nevertheless, the approach proved to be able to produce anomaly-free schedules with low expected completion time and guaranteed to meet hard deadline constraints[17].

The Precedence Constraint Posting technique naturally produces flexible schedules, allowed to stretch and accommodate actual task durations, reducing idle time; additionally, an off-line computed PCP schedule incurs a very little run-time overhead and can serve as a guideline for a more complex on-line scheduler (e.g. to react to hardware faults). All in all, the PCP technique has a strong, largely underestimated, potential: we hope our work provides motivation for the research community and the industry to seriously consider the use of PCP as a scheduling method for embedded processing.

REFERENCES

[1] D. C. Pham et. al., "Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor," *Solid-State Circuits, IEEE Journal of*, vol. 41, no. 1, pp. 179–196, 2006.

[2] M. Paganini, "Nomadik: A Mobile Multimedia Application Processor Platform," in *Proc. of ASP-DAC*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 749–750.

[3] J. Reineke et. al., "A definition and classification of timing anomalies," in *Proc. of WCET*, 2006.

[4] J. W. S. Liu, *Real-time systems*. Prentice Hall, 2000.

[5] L. Thiele and R. Wilhelm, "Design for timing predictability," *Real-Time Systems*, vol. 28, no. 2, pp. 157–177, 2004.

[6] M. Lombardi, M. Milano, and L. Benini, "Robust non-preemptive hard real-time scheduling for clustered multicore platforms," in *Proc. of DATE*. IEEE, 2009, pp. 803–808.

[7] M. Lombardi and M. Milano, "A Precedence Constraint Posting Approach for the RCPSP with Time Lags and Variable Durations," in *Proc. of CP*. Springer, 2009, pp. 569–583.

[8] N. Policella et. al., "From precedence constraint posting to partial order schedules: A CSP approach to Robust Scheduling," *AI Communications*, vol. 20, no. 3, pp. 163–180, 2007.

[9] N. Fisher and S. K. Baruah, "The partitioned multiprocessor scheduling of non-preemptive sporadic task systems," *Proc. of RTNS*, 2006.

[10] S. K. Baruah, "The non-preemptive scheduling of periodic tasks upon multiprocessors," *Real-Time Systems*, vol. 32, no. 1, pp. 9–20, 2006.

[11] M. Ruggiero et. al., "A fast and accurate technique for mapping parallel applications on stream-oriented MPSoC platforms with communication awareness," *International Journal of Parallel Programming*, vol. 36, no. 1, pp. 3–36, 2008.

[12] E. Zitzler et. al., "Performance assessment of multiobjective optimizers: an analysis and review," *Evolutionary Computation, IEEE Transactions on*, vol. 7, no. 2, pp. 117–132, Apr. 2003.

[13] R. L. Graham, "Bounds for certain multiprocessing anomalies," *Bell System Technical Journal*, vol. 45, no. 9, pp. 1563–1581, 1966.

[14] A. Brekling, M. R. Hansen, and J. Madsen, "Models and formal verification of multiprocessor system-on-chips," *Journal of Logic and Algebraic Programming*, vol. 77, no. 1-2, pp. 1–19, 2008.

[15] P. Brucker et. al., "Resource-constrained project scheduling: Notation, classification, models, and methods," *European Journal of Operational Research*, vol. 112, no. 1, pp. 3–41, 1999.

[16] S. Hartmann and D. Briskorn, "A survey of variants and extensions of the resource-constrained project scheduling problem," *European Journal of Operational Research*, vol. 207, no. 1, pp. 1—-14, 2010.

[17] Obviously, provided the model is suitable for the target application and platform.

[17] P. Baptiste, C. Le Pape, and W. Nuijten, *Constraint-based scheduling*. Kluwer Academic Publishers, 2001.

[18] P. Laborie, "Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results," *Artificial Intelligence*, vol. 143, no. 2, pp. 151–188, Feb. 2003.

[19] J. C. Beck and A. J. Davenport, "A survey of techniques for scheduling with uncertainty," 2002. [Online]. Available: http://www.eil.utoronto.ca/profiles/chris/gz/uncertainty-survey.ps

[20] W. Herroelen and R. Leus, "Project scheduling under uncertainty: Survey and research potentials," *European journal of operational research*, vol. 165, no. 2, pp. 289–306, 2005.

[21] S. de Vonder, E. L. Demeulemeester, and W. Herroelen, "A classification of predictive-reactive project scheduling procedures," *Journal of Scheduling*, vol. 10, no. 3, pp. 195–207, 2007.

[22] A. J. Davenport, C. Gefflot, and J. C. Beck, "Slack-based techniques for robust schedules," *Proc. of ECP*, pp. 7–18, 2001.

[23] G. Igelmund and F. J. Radermacher, "Preselective strategies for the optimization of stochastic project networks under resource constraints," *Networks*, vol. 13, no. 1, pp. 1–28, Jan. 1983.

[24] F. Stork, "Stochastic resource-constrained project scheduling," Ph.D. dissertation, Technische Universitat Berlin, 2001.

[25] F. Wang and Y. Xie, "Embedded Multi-Processor System-on-chip (MPSoC) design considering process variations," in *Proc. of IEEE IPDPS*. Ieee, Apr. 2008, pp. 1–5.

[26] S. Manolache, P. Eles, and Z. Peng, "Schedulability analysis of applications with stochastic task execution times," *ACM Transactions on Embedded Computing Systems*, vol. 3, no. 4, pp. 706–735, Nov. 2004.

[27] R. Hall et al., "A comparison of dag-scheduling strategies for Internet-based computing," in *Proc. of IEEE IPDPS*. Citeseer, 2007, pp. 1–9.

[28] G. Cordasco and A. Rosenberg, "Area-maximizing schedules for series-parallel DAGs," *Proc of Euro-Par*, pp. 380–392, 2010.

[29] V. Shestak and J. Smith, "Iterative algorithms for stochastically robust static resource allocation in periodic sensor driven clusters," *Proceedings of PDCS*, pp. 166–174, 2006.

[30] A. Cesta, A. Oddi, and S. Smith, "Scheduling Multi-Capacitated Resources under Complex Temporal Constraints," *Proc. of CP*, pp. 465–465, 1998.

[31] P. Laborie, "Complete MCS-Based Search: Application to Resource Constrained Project Scheduling," in *Proc. of IJCAI*. Professional Book Center, 2005, pp. 181–186.

[32] A. Cesta, A. Oddi, and S. F. Smith, "Iterative Flattening: A Scalable Method for Solving Multi-Capacity Scheduling Problems," in *Proc. of AAAI/IAAI*, 2000, pp. 742–747.

[33] R. Dechter, I. Meiri, and J. Pearl, "Temporal Constraint Networks," *Artificial Intelligence*, vol. 49, no. 1-3, pp. 61–95, 1991.

[34] T. Vidal and H. Fargier, "Handling contingency in temporal constraint networks: from consistency to controllabilities," *J. Exp. Theor. Artif. Intell.*, vol. 11, no. 1, pp. 23–45, 1999.

[35] P. H. Morris and N. Muscettola, "Temporal dynamic controllability revisited," in *Proc. of AAAI*, 2005, pp. 1193–1198.

[36] N. Muscettola, "Computing the Envelope for Stepwise-Constant Resource Allocations," in *Proc. of CP*, 2002, pp. 139–154.

[37] D. Culler et al., "LogP: A practical model of parallel computation," *Communications of the ACM*, vol. 39, no. 11, pp. 78–85, 1996.

[38] L. Benini et. al., "Allocation and scheduling for mpsocs via decomposition and no-good generation," *Proc. of IJCAI*, vol. 19, p. 1517, 2005.

[39] M. Golumbic, *Algorithmic Graph Theory And Perfect Graphs*, 2nd ed. Elsevier, 2004.

[40] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *Journal of the ACM*, vol. 19, no. 2, pp. 248–264, 1972.

[41] R. M. Haralick and G. L. Elliott, "Increasing tree search efficiency for constraint satisfaction problems," *Artificial intelligence*, vol. 14, no. 3, pp. 263–313, 1980.

[42] S. Bell et. al., "TILE64 processor: A 64-core SoC with mesh interconnect," *Proc. of ISSCC*, pp. 88–598, 2008.

[43] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, p. 359, 1999.

[44] S. Manolache, P. Eles, and Z. Peng, "Task mapping and priority assignment for soft real-time applications under deadline miss ratio constraints," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 2, pp. 1–35, Feb. 2008.