Stochastic Allocation and Scheduling for Conditional Task Graphs in MPSoCs Techical report

Michele Lombardi and Michela Milano

DEIS, University of Bologna
V.le Risorgimento 2, 40136, Bologna, Italy

Abstract. This paper describes a complete and efficient solution to the stochastic allocation and scheduling for Multi-Processor System-on-Chip (MPSoC). Given a conditional task graph characterizing a target application and a target architecture with alternative memory and computation resources, we compute an allocation and schedule minimizing the expected value of communication cost, being the communication resources one of the major bottlenecks in modern MPSoCs. Our approach is based on the Logic Based Benders decomposition where the stochastic allocation is solved through an Integer Programming solver, while the scheduling problem with conditional activities is faced with Constraint Programming. The two solvers interact through no-goods. The original contributions of the approach appear both in the allocation and in the scheduling part. For the first, we propose an exact analytic formulation of the stochastic objective function based on the task graph analysis, while for the scheduling part we extend the timetable constraint for conditional activities. Experimental results show the effectiveness of the approach.

1 Introduction

The increasing levels of system integration in Multi-Processor Systems on Chips (MPSoCs) emphasize the need for new design flows for efficient mapping of multi-task applications onto hardware platforms. The problem of allocating and scheduling conditional, precedence-constrained tasks on processors in a distributed realtime system is NP-hard. As such, it has been traditionally tackled by means of heuristics, which provide only approximate or near-optimal solutions, see for example [1], [2], [3].

In a typical embedded system design scenario, the platform always runs the same application. Thus, extensive analysis and optimization can be performed at design time. This paper proposes a complete decomposition approach to the allocation and scheduling of a conditional multi-task application on a multi-processor system-on-chip (MPSoCs) [4]. The target application is pre-characterized and abstracted as a Conditional Task Graph (CTG). The task graph is annotated with computation time, amount of communication, storage requirements. However, not all tasks will run on the target platform: in fact, the application contains conditional branches (like if-then-else control structures). Therefore, after an accurate application profiling step, we have a probability distribution on each conditional branch that intuitively gives the probability of choosing that branch during execution.

This paper proposes a non trivial extension to [5] that used Logic Based Benders decomposition [6] for resource assignment and scheduling in MPSoCs. In that paper, however, task graphs did not contain conditional activities. Allocation and scheduling were therefore deterministic. The introduction of stochastic elements complicates the problem.

We propose two main contributions: the first concerns the allocation component. The objective function we consider depends on the allocation variables. Clearly, having conditional tasks, the exact value of the communication cost cannot be computed. Therefore our objective function is the expected value of the communication cost. We propose here to identify an analytic approximation of this value. The approximation is based on the Conditional Task Graph analysis for identifying two data structures: the activation set of a node and the coexistence set of two nodes. The approximation turns out to be exact and polynomial. The second contribution concerns scheduling. We propose an extension of the time-table constraint for cumulative resources, taking into account conditional activities. The propagation is polynomial if the task graph satisfies a condition called *Control Flow Uniqueness* which is quite common in many conditional task graphs for system design. Its deterministic version [7] is available in ILOG Scheduler. The use of the so called *optional activities* (what we call conditional tasks) has been taken into account in [8] for filtering purposes into the precedence graph, originally introduced by Laborie in [9]. To the best of our knowledge, only disjunctive constraints have been defined in presence of conditional activities in [10].

In the system design community, this problem is extremely important and many researchers have worked extensively on it, mainly with incomplete approaches: for instance in [1] a genetic algorithm is devised on the basis of a conditional scheduling table whose (exponential number of) columns represent the combination of conditions in the CTG and whose rows are the starting times of activities that appear in the scenario. The number of columns is indeed reasonable in real applications. The same structure is used in [10], which is the only approach that uses Constraint Programming for modelling the allocation and scheduling problem. Indeed the solving algorithm used is complete only for small task graphs (up to 10 activities).

Besides related literature for similar problems, the Operations Research community has extensively studied stochastic optimization in general. The main approaches are: sampling [11] consisting in approximating the expected value with its average value over a given sample; the *l-shaped* method [12] which faces two phase problems and is based on Benders Decomposition [13]. The master problem is a deterministic problem for computing the first phase decision variables. The subproblem is a stochastic problem that assigns the second phase decision variables minimizing the average value of the objective function. A different method is based on the branch and bound extended for dealing with stochastic variables, [14].

The CP community has recently faced stochastic problems: in [15] stochastic constraint programming is formally introduced and the concept of solution is replaced with the one of *policy*. In the same paper, two algorithms have been proposed based on backtrack search. This work has been extended in [16] where an algorithm based on the concept of scenarios is proposed. In particular, the paper shows how to reduce the number of scenarios, maintaining a good expressiveness.

This paper is organized as follows: in section 2 we present the architecture and the target application considered. In section 3 we present the allocation and scheduling models used. Experimental results are shown in section 4.

2 Problem description

2.1 The architecture

Multi Processor Systems on Chips (MPSoCs) are multi core architectures developed on a single chip. They are finding widespread application in embedded systems (such as cellular phones, automotive control engines, etc.). Once deployed in field, these devices always run the same application, in a well-characterized context. It is therefore possible to spend a large amount of time for finding an optimal allocation and scheduling off-line and then deploy it on the field, instead of using on-line, dynamic (sub-optimal) schedulers [17, 18].

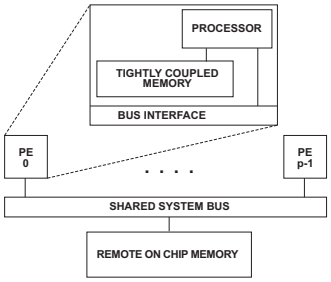

Fig. 1. Single chip multi-processor architecture.

The multi-processor system we consider consists of a pre-defined number of distributed Processing Elements (PE) as depicted in Figure 1. All nodes are assumed to be homogeneous and composed by a processing core and by a low-access-cost local scratchpad memory. Data storage onto the scratchpad memory is directly managed by the application, and not automatically in hardware as it is the case for processor caches.

The scratchpad memory is of limited size, therefore data in excess must be stored externally in a remote on-chip memory, accessible via the bus. The bus for state-of-the-art MPSoCs is a shared communication resource, and serialization of bus access requests of the processors (the bus masters) is carried out by a centralized arbitration mechanism. The bus is re-arbitrated on a transaction basis (e.g., after single read/write transfers, or bursts of accesses of pre-defined length), based on several policies (fixed priority, round-robin, latency-driven, etc.). Modeling bus allocation at such a fine granularity would make the problem overly complex, therefore a more abstract additive bus model was devised, explained and validated in [5] where each task can simultaneously access the bus requiring a portion of the overall bandwidth. The communication resource in most cases ends up to be the most congested resource. Communication cost

is therefore critical for determining overall system performance, and will be minimized in our task allocation framework.

2.2 The target application

The target application to be executed on top of the hardware platform is the input to our algorithm. It is represented as a Conditional Task Graph (CTG). A CTG is a triple $\langle T, A, C \rangle$, where T is the set of nodes modeling generic tasks (e.g. elementary operations, subprograms, ...), A the set of arcs modeling precedence constraints (e.g. due to data communication), and C is a set of conditions, each one associated to an arc, modeling what should be true in order to choose that branch during execution (e.g. the condition of a if-then-else construct). A node with more than one outgoing arc is said to be a *branch* if all arcs are conditional, a *fork* if all arcs are not conditional; mixed nodes are not allowed. A node with more than one ingoing arc is an *or-node* if all arcs are mutually exclusive, it is instead an *and-node* if all arcs are not mutually exclusive; again, mixed nodes are not allowed.

Since the truth or the falsity of conditions is not known in advance, the model is stochastic. In particular, we can associate to each branch a stochastic variable $\mathcal{B}$ with probability space $\langle C, \mathcal{A}, p \rangle$, where C is the set of possible branch exit conditions c, $\mathcal{A}$ the set of events (one for each condition) and p the branch probability distribution (in particular $p(c)$ is the probability that condition c is true).

We can associate to each node and arc an activation function, expressed as a composition of conditions by means of the logical operators $\wedge$ and $\vee$. We call it $f_i(X(\omega))$, where X is the stochastic variable associated to the composite experiment $\mathcal{B}_0 \times \mathcal{B}_1 \times ... \times \mathcal{B}_b$ ($b =$ number of branches) and $\omega \in \mathcal{D}(\mathcal{B}_0) \times \mathcal{D}(\mathcal{B}_1) \times ... \times \mathcal{D}(\mathcal{B}_b)$ (i.e. ω is a scenario).

Computation, storage and communication requirements are annotated onto the graph. In detail, the worst case execution time (WCET) is specified for each node/task and plays a critical role whenever application real time constraints (expressed here in terms of deadlines) are to be met.

Each node/task also has three kinds of associated memory requirements: **Program Data**: storage locations are required for computation data and for processor instructions; **Internal State**; **Communication queues**: the task needs queues to transmit and receive messages to/from other tasks, eventually mapped on different processors. Each of these memory requirement can be allocated either locally in the scratchpad memory or remotely in the on-chip memory.

Finally, the communication to be minimized counts two contributions: one related to single tasks, once computation data and internal state are physically allocated to the scratchpad or remote memory, and obviously depending on the size of such data; the second related to pairs of communicating tasks in the task graph, depending on the amount of data the two tasks should exchange.

3 Model definition

As already presented and motivated in [5], the problem we are facing can be split into the resource allocation master problem and the scheduling sub-problem.

3.1 Allocation problem model

With regards to the platform described in section 2.1, the allocation problem can be stated as the one of assigning processing elements to tasks and storage devices to their memory requirements. First, we state the stochastic allocation model, then we show how this model can be transformed into a deterministic model through the use of existence and co-existence probabilities of tasks. To compute these probabilities, we propose two polynomial time algorithms exploiting the CTG structure.

Stochastic integer linear model Suppose n is the number of tasks, p the number of processors, and n_a the number of arcs. We introduce for each task and each PE a variable T_{ij} such that $T_{ij} = 1$ iff task i is assigned to processor j. We also define variables M_{ij} such that $M_{ij} = 1$ iff task i allocates its program data locally, $M_{ij} = 0$ otherwise. Similarly we introduce variables S_{ij} for task i internal state requirements and C_{rj} for arc r communication queue. X is the stochastic variable associated to the scenario ω. The allocation model, where the objective function is the minimization of bus traffic expected value, is defined as follows:

$$\min z = E(\textit{busTraffic}(M, S, C, X(\omega)))$$

$$s.t. \quad \sum_{j=0}^{p-1} T_{ij} = 1 \qquad \forall i = 0, .., n-1 \tag{1}$$

$$S_{ij} \leq T_{ij} \qquad \forall i = 0, .., n-1, j = 0, .., p-1 \tag{2}$$

$$M_{ij} \leq T_{ij} \qquad \forall i = 0, .., n-1, j = 0, .., p-1 \tag{3}$$

$$C_{rj} \leq T_{ij} \qquad \forall arc_r = (t_i, t_k), r = 0, .., n_a - 1, j = 0, .., p-1 \tag{4}$$

$$C_{rj} \leq T_{kj} \qquad \forall arc_r = (t_i, t_k), r = 0, .., n_a - 1, j = 0, .., p-1 \tag{5}$$

$$\sum_{i=0}^{n-1} [s_i S_{ij} + m_i M_{ij}] + \sum_{r=0}^{n_a-1} c_r C_{rj} \leq Cap_j \qquad \forall j = 0, .., p-1 \tag{6}$$

Constraints (1) force each task to be assigned to a single processor. Constraints (2) and (3) state that program data and internal state can be locally allocated on the PE j

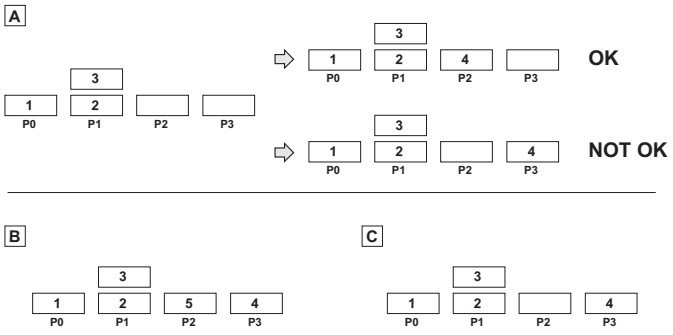

Fig. 2. *Breaking some symmetries*

only if task i runs on it. Constraints (4) and (5) enforce that the communication queue of arc r can be locally allocated only if both the source and the destination tasks run on processor j. Finally, constraints (6) ensure that the sum of locally allocated internal state (s_i), program data (m_i) and communication (c_r) memory cannot exceed the scratchpad device capacity (Cap_j). All tasks have to be considered here, regardless they execute or not at runtime, since a scratchpad memory is, by definition, statically allocated.

In addition, some symmetries breaking constraints have been added to the model. Suppose for example to allocate tasks one by one: since the PEs are symmetric resources, when choosing where to execute task i, if PE j is free there's no point in choosing PE $j+1$ instead (fig. 2A). To partly enforce this we can state that in any allocation if i is the minimum index value on PE j, then no task with index *less* than i can be allocated on PEs $j+1, j+2$, etc.:

$$\forall i = 1,..,n-1,\ \forall j = 0,..,p-2 \qquad \sum_{h=0}^{i-1}\sum_{k=j+1}^{p-1} T_{hk} \leq i \sum_{h=0}^{i-1} T_{hj} \tag{7}$$

The left member of 7 is the number of task with index higher than i executing on PEs with index higher that j. The right member is 0 iff task i has the lowest order on PE j (was allocated when the PE was empty), otherwise it does not enforce any limit.

The constraints 7 cut some symmetries, but still allow a PE to be completely skipped; for example they forbid the allocation of figure 2B, but not the one of figure 2C. This can be prevented by enforcing:

$$\forall j = 0,..,p-2 \qquad \sum_{k=j+1}^{p-1}\sum_{i=0}^{n-1} T_{ik} \leq n \sum_{i=0}^{n-1} T_{ij} \tag{8}$$

Constraints 8 forbid allocations where an empty PE is followed by not empty PEs.

The bus traffic expression is composed by two contributions: one depending on single tasks and one due to the communication between pairs of tasks.

$$\textit{busTraffic} = \sum_{i=0}^{n-1} \textit{taskBusTraffic}_i + \sum_{arc_r=(t_i,t_k)} \textit{commBusTraffic}_r$$

where

$$\textit{taskBusTraffic}_i = f_i(X(\omega)) \left[m_i(1 - \sum_{j=0}^{p-1} M_{ij}) + s_i(1 - \sum_{j=0}^{p-1} S_{ij}) \right]$$

$$\textit{commBusTraffic}_r = f_i(X(\omega)) f_k(X(\omega)) \left[c_r(1 - \sum_{j=0}^{p-1} C_{rj}) \right]$$

In the *taskBusTraffic* expression, if task i executes (thus $f_i(X(\omega)) = 1$), then $(1 - \sum_{j=0}^{p-1} M_{ij})$ is 1 iff the task i program data is remotely allocated. The same holds for the internal state. In the *commBusTraffic* expression we have a contribution if both the source and the destination task execute ($f_i(X(\omega)) = f_k(X(\omega)) = 1$) and the queue is remotely allocated ($1 - \sum_{j=0}^{p-1} C_{rj} = 1$).

Transformation in a deterministic model In most cases, the minimization of a stochastic functional, such as the expected value, is a very complex operation (even more than exponential), and it's often achived by repeatedly solving a deterministic subproblem [12]. The cost of such a procedure is not affordable for hardware design purposes since the deterministic subproblem is by itself NP-hard. **One of the main contributions of this paper is the way to reduce the bus traffic expected value to a deterministic expression.** Since all tasks have to be assigned before running the application, the allocation is a stochastic *one phase* problem: thus, for a given task-PE assignment, the expected value depends only on the stochastic variables. Intuitively, if we properly weight the bus traffic contributions according to task probabilities we should be able to get an analytic expression for the expected value.

Now, since both the expected value operator and the bus traffic expression are linear, the objective function can be decomposed into task related and arc related blocks:

$$E(\textit{busTraffic}) = \sum_{i=0}^{n-1} E(\textit{taskBusTraffic}_i) + \sum_{arc_r=(t_i,t_k)} E(\textit{commBusTraffic}_r)$$

Since for a given allocation the objective function depends only on the stochastic variables, the contributions of decision variables are constants: we call them $KT_i = \left[m_i(1 - \sum_{j=0}^{p-1} M_{ij}) + s_i(1 - \sum_{j=0}^{p-1} S_{ij})\right]$, and $KC_r = \left[c_r(1 - \sum_{j=0}^{p-1} C_{rj})\right]$. Let us call $p(\omega)$ the probability of scenario ω.

The expected value of each contribution to the objective function is a weighted sum on all scenarios. Weights are scenario probabilities.

$$E(\textit{taskBusTraffic}_i) = \sum_{\omega \in \Omega} p(\omega) f_i(X(\omega)) KT_i = KT_i \sum_{\omega \in \Omega_i} p(\omega)$$

$$E(\textit{commBusTraffic}_r) = \sum_{\omega \in \Omega} p(\omega) f_i(X(\omega)) f_k(X(\omega)) KC_r = KC_r \sum_{\omega \in \Omega_i \cap \Omega_k} p(\omega)$$

where r is the index of arc (t_i, t_j) and $\Omega_i = \{\omega \mid \text{task } i \text{ executes}\}$ is the set of all scenarios where task i executes. Now every stochastic dependence is removed and the expected value is reduced to a deterministic expression. Note that $\sum_{\omega \in \Omega_i} p(\omega)$ is simply the existence probability of node/task i while $\sum_{\omega \in \Omega_i \cap \Omega_k} p(\omega)$ is the coexistence probability of nodes i and k. To apply the transformation we need both those probabilities; moreover, to achieve an effective overall complexity reduction, they have to be computed in a reasonable time. We developed two polynomial cost algorithms to compute these probabilities.

Existence and co-existence probabilities All developed algorithms are based on three types of data structures derived from the CTG. In Figure 3A we show an example of a CTG on the left and the related data structures:

– the *activation set* of a node n ($AS(n)$): it is computed by traversing all paths from the starting node to n and collecting the conditions on the paths.

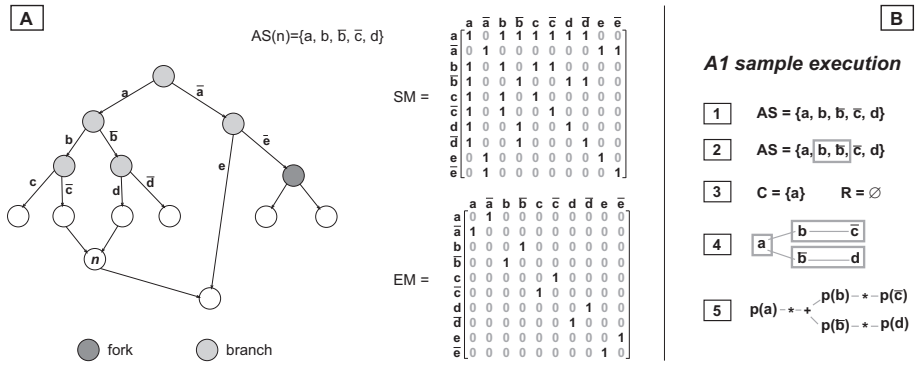

Fig. 3. A: An example of the three data structures; **B:** a sample execution of A1

– a binary $c \times c$ *exclusion matrix (EM)*, where c is the number of conditions and $EM_{ij} = 1$ iff c_i and c_j are mutually exclusive (i.e. they originate at the same branch)
– a binary $c \times c$ *sequence matrix (SM)*: $SM_{ij} = 1$ iff c_i and c_j are both needed to activate some node in the CTG ($SM_{ij} = 1 \Leftrightarrow \forall n \mid c_i, c_j \in AS(n),\ n$ executes iff $c_i = true \wedge c_j = true$).

There's a connection between those data structure and the activation functions; in particular:

- $c_i \in AS(n) \Leftrightarrow c_i$ appears in $f_n(X(\omega))$
- $c_i, c_j \in AS(n) \wedge SM_{ij} = 1 \Rightarrow c_i$ and c_j are connected by means of the operator $\wedge$ in $f_n(X(\omega))$
  $c_i, c_j \in AS(n) \wedge EM_{ij} = 1 \Rightarrow c_i$ and c_j are connected by means of the operator $\vee$ in $f_n(X(\omega))$

The computation of EM is trivial, while SM and all the activation sets can be extracted from the graph in polynomial time by means of the following algorithm:

algorithm: AS and SM extraction algorithm (A0)

1. As long as there are unexplored nodes:
  1.1. find a node n whose predecessors have all been explored
  1.2. for each *conditional* ingoing arc e with source n_s:
    1.2.1. let c_i the condition e is labelled with; set $SM_{ij} = 1$ for each $c_j \in AS(n_s)$
  1.3. if n has no ingoing arc, set $AS(n) = \emptyset$
  1.4. if n has a single ingoing arc e with source n_s: set $AS(n) = AS(n_s) \cup c(e)$
  1.5. if n has more than one ingoing arc and n is an *or*-node: for each ingoing arc e with source n_s:
    1.5.1. set $AS(n) = AS(n) \cup AS(n_s) \cup c(e)$
  1.6. if n has more than one ingoing arc and n is an *and*-node:
    1.6.1. for each ingoing arc e with source n_s:
      1.6.1.1. set $AS(n) = CS(AS(n), AS(n_s) \cup c(e)$
    1.6.2. for each ingoing arc e with source n_s:
      1.6.2.1. Set $SM_{ij} = 1$ for each $c_i \in AS(n) \cap (AS(n_s) \cup c(e))$ and for each $c_j \in AS(n) \setminus (AS(n_s) \cup c(e))$

end

The AS of each node n is computed according to three rules:

1. if n has no ingoing arc, then $AS(n) = \emptyset$
2. if n is an or-node, then $AS(n)$ is the union of the AS of all the predecenssors, evetually augmented with the conditions on the ingoing arcs
3. if n is and and-node, then $AS(n)$ is the coexistence set (CS) of all the AS fo the predecessors, evetually augmented with the conditions on the ingoing arcs. The notion of coexistence set will be detailed later; as an intuition, $CS(AS(n_i), AS(n_j))$, is the set of conditions involved in the activation of both n_i and n_j

As for the sequence matrix (SM):

1. if c_i is on an arc e with source n_s, then $SM_{ij} = 1 \; \forall c_j \in AS(n_s)$
2. if n is an and-node, every condition $c_i \in AS(n)$ coming from a predecessor n_s is "in sequence" with all conditions $c_j \in AS(n)$ coming from other predecessors

Once all the data structure are available, we can determine the existence probability of a node or an arc using algorithm A1, which has $O(c^3)$ complexity representing sets as bit vectors; in the algorithm the notation SM_i stands for the set of conditions "sequenced" with a given one ($SM_i = \{c_j \mid SM_{ij} = 1\}$); the same holds for EM_i.

algorithm: Activation set probability (A1) – probability of a node or an arc

1. let S be the input set for this iteration; initially $S = AS(n)$
2. find a condition $c_h \in S$ such that $(EM_h \setminus \{c_h\}) \cap S \neq \emptyset$
3. if such a condition doesn't exist return $p = \prod_{c \in S} p(c)$
4. otherwise, set $B = EM_h \cap S$
5. compute set $C = S \cap \bigcap_{c_i \in B} SM_i$
6. compute set $R = \bigcap_{c_i \in B} (S \setminus SM_i)$
7. set $p = 0$
8. for each condition $c_i \in B$:
8.1. set $p = p + A1((S \cap SM_i) \setminus (C \cup R))$
9. set $p = p * A1(C) * A1(R)$
10. return p

2: *find a group of exclusive conditions in S and choose one of them*
4: *get all conditions in S originating from the chosen branch ("Branch" set)*
5: *get conditions "in sequence" with all branch outcomes ("Common" set)*
6: *get conditions not "in sequence" with any of the branch outcomes ("Rest" set)*
8.1: *for each branch outcome, get the conditions "in sequence" and compute probability*
9: *multiply computed value for the probability of "Common" and "Rest" set*

end

Algorithm A1 works recursively partitioning the activation set of the target node: let us follow the algorithm on the example in figure 3B. We have to compute the probability of node n, while activation set is `AS(n) = {a,b, not b, not c, d}`. The algorithm looks for a group of mutually exclusive conditions (the B set), see `b` and `not b` in `AS(n)`. If there is no such condition the probability of the activation set S is the product of the probabilities of its elements (step 3). If there is are at least two exclusive conditions, the algorithm then builds a "common" (C) and a "rest" (R) set: the first contains conditions c_j such that $SM_{ij} = 1 \quad \forall c_i \in B$, the second conditions c_h such that $SM_{ih} = 0 \quad \forall c_i \in B$. In the example `C = {a}` and `R = ∅`. Finally A1 builds

for each found branch condition a set containing the sequenced conditions ($S \cap SM_i$ at step 8.1), and chains `b` and `not c` and `not b` and `d` in figure 3. A1 is then recursively called on all these sets. The probabilities of sets corresponding to mutually exclusive conditions are summed (step 8.1), the ones of C and R are multiplied (step 9).

Theorem 1. *If fed with the activation set of a node or arc or with the coexistence set of more nodes or arcs, A1 correctly computes the existence probability.*

Proof. Given the activation function of a node, expressed as a disjunction of conjunction of conditions (Disjunctive Normal Form – DNF), the existence probability can be computed by substituting the conditions with their probabilities, the operator "$\wedge$" with "$\times$" and the operator "$\vee$" with "$+$":

$$f_n(X(\omega)) = (c_0 \wedge c_1) \vee (c_2 \wedge c_3) \vee \ldots$$
$$p(n) = p(c_0)p(c_1) + p(c_2)p(c_3) + \ldots$$

Suppose c_i, c_j are the mutually exclusive conditions selected at steps 2-4; since $c_i, c_j \in AS(n)$ and $EM_{ij} = 1$, they are connected by a $\vee$ operator. The activation function can thus be decomposed as:

$$\begin{aligned} f_n(X(\omega)) &= (c_i \wedge g_i(X(\omega)) \vee c_j \wedge g_j(X(\omega))) \\ &= (c_i \wedge g_i'(X(\omega))c_j \wedge g_j'(X(\omega))) \wedge g_{ij}(X(\omega)) \end{aligned}$$

Where g_i, g_j and g_{ij} are again boolean expressions in disjunctive normal form. From the definition of sequence matrix comes that $SM_{ik} = 1$ for each c_k appearing in g_i', $SM_{jk} = 1$ for each c_k in g_j', $SM_{ik} = 1 \wedge SM_{jk} = 1$ for each c_k in g_{ij}. Therefore, the conditions in g_{ij} are exactly those in the C set computed at step 5, while conditions in g_i', g_j' are those in the set $(S \cap SM_i) \backslash C$ (see step 8.1). The procedure can be reiterated.

Finally, to get probability we have to compute $p(f_n(X(\omega))) = (p(c_i)p(g_i) + p(c_j)p(g_j))p(g_{ij})$, which is the same computation performed by A1.

Note that if the input of A1 is a "normal" activation set, then $R = \emptyset$. R is not empty only if the input set is a coexistence set (explained in more detail later on); in that case conditions in R are needed to activate other nodes/arcs, and the probabilities must therefore be multiplied. **QED**

---

algorithm: Coexistence set determination (A2)

1. if $AS_i = \emptyset$ then $CS = AS_j$; the same if $AS_j = \emptyset$ — 1: *If one of the input set is $\emptyset$, then the coexistence set is simply the other activation set*

2. otherwise, if there are still not processed conditions in AS_i, let c_h be the first of them:

- **2.1.** compute set $S = AS_i \cap SM_h$ — 2.1: *Select conditions "in sequence" with the chosen one; they individuate a group of backward paths*
- **2.2.** compute the exclusion set $EX(S)$
- **2.3.** compute set: $C = AS_j \cap \bigcup_{c_k \in AS_j \cap EX(S)} SM_k$ — 2.3: *Find in AS_j conditions excluded by the selected forward paths ($AS_j \cap EX(S)$); all conditions "in sequence" with those are canditates to be deleted (i.e. excluded from the feasible forward paths)*
- **2.4.** compute set: $R = AS_j \cap \bigcup_{c_k \in AS_j \setminus C} SM_k$ — 2.4: *Consider not candidate conditions in AS_j: all conditions "in sequence" with them must not be deleted*
- **2.5.** set $D = C \setminus R$ (conditions to delete)
- **2.6.** if AS_j is not a subset of D: — 2.6: *If AS_j has not been completely deleted, add to CS all conditions in backward and forward paths*
  - **2.6.1.** set $CS(AS_i, AS_j) = CS(AS_i, AS_j) \cup S \cup (AS_j \setminus D)$

end

Given a set of nodes, we can determine a kind of common activation set (*coexistence set (CS)*) using algorithm A2, whose inputs are two AS (AS_i, AS_j) and whose complexity is again $O(c^3)$. The notation $EX(S)$ stands for the exclusion set, i.e. the set of conditions surely excluded by those in S ($EX(S) = \{c_i \notin S \mid \exists c_j \in S$ such that $EM_{ij} = 1\}$); it can be computed easily in $O(c^2)$.

Suppose we have the activation sets of two nodes n_i and n_j: then A2 works trying to find all paths from n_i to a source (backward paths) and from the source to n_j (forward paths). The algorithm starts building a group of backward paths; it does it by choosing a condition (for instance condition `a` in 1 figure 4) and finding all other conditions sequenced with it (set S in 2 figure 4).

Then the algorithm finds the exclusion set ($EX(S)$) of set S and intersects it with $AS(n_j)$. In 3 figure 4 the only condition in the intersection is `not a` (crossed arc): conditions in the intersection and those sequenced with them are called "candidates conditions" (set C in 3 figure 4). These conditions will be removed from $AS(n_j)$, unless they are sequenced with one or more non-candidate conditions, i.e., they belong to the set R (for instance condition `f` is in sequence with `not b` and is not removed from $AS(n_j)$ in 4, figure 4). The conditions not removed from $AS(n_j)$ identify a set of forward paths we are interested in. The algorithm goes on until all conditions in $AS(n_i)$ are processed. If there is no path from n_i to n_j (i.e. the coexistence set is empty) the two nodes are mutually exclusive.

Theorem 2. *Given two nodes or arcs with activation set AS_i and AS_j, their joint activation depends only on the conditions in $CS(AS_i, AS_j)$*

Proof. First, note that from the definition of activations set comes $CS(AS_i, AS_j) \subseteq AS_i \cup AS_j$. Since some conditions are mutually exclusive, from this set we have to discard any condition $c_h \in AS_i$ which prevents the node/arc j to execute, and vice versa; more in detail we have to discard any $c_h \in AS_i$ such that:

$$\forall c_k \in AS_j : \quad \exists c_r \in AS_j \mid EM_{hr} = 1 \wedge SM_{lk} = 1 \tag{9}$$

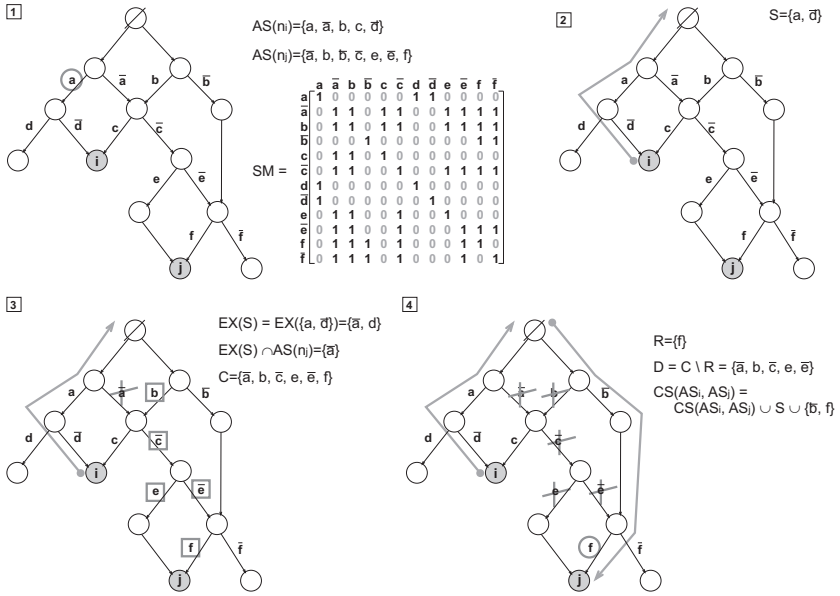

Fig. 4. Coexistence set computation

Furthermore, we have to discard any condition dependent on such a c_h, i.e. any condition originating at a branch that can be reached only through c_h.

Let's suppose such a condition $c_h \in AS_i$ to exist: at a certain step it has to be part of the S set. Note that $C = \{c_k \in AS_j \mid \exists c_r \in EX(S) \cap AS_j \wedge SM_{lk} = 1\}$ and $c_r \in EX(S) \Rightarrow EM_{hl} = 1$ and so, since property 9 holds, we have $C = AS_j$. For the same reason also $R = \emptyset$. In this case the entire S set (and thus also c_h) is not added to the coexistence set. Moreover, any condition dependent on c_h will never be added (since c_h will always appear in their S set).

Now let's suppose such c_h to be part of AS_j: since property 9 holds $\forall c_k \in AS_i$ there will always be a condition c_l in the S set of c_k such that $EM_{hl} = 1$; this implies $c_h \in EX(S)$, and c_h will thus never included in the coexistence set. Note also that by construction all conditions dependent on c_h will be part of the set $C \setminus R$, which is never added to the coexistence set, too. **QED**

The probability of a coexistence set can be computed once again by means of A1: thus, with A1 and A2 we are able to compute the existence probability of a single node and the coexistence probability of a group of nodes or arcs. Since the algorithms complexities are polynomial, the reduction of the bus traffic to a deterministic expression can be done in polynomial time.

3.2 Scheduling Model

The scheduling subproblem has been solved by means of Constraint Programming. Since the objective function depends only on the allocation of tasks and memory re-

quirements, scheduling is just a feasibility problem. Therefore we decided to provide a unique worst case schedule, forcing each task to execute after all its predecessors in any scenario. Tasks using the same resources can overlap if they are on alternative paths (under two mutually exclusive conditions).

Fig. 5. Each task has a five phase behavior

Tasks have a five phases behavior (fig. 5): they read all communication queues (INPUT), eventually read their internal state (RS), execute (EXEC), write their states (WS) and finally write all the communications queues (OUTPUT). Each task is modeled as a group of not breakable activities; the adopted schema and precedence relations vary with the type of the corresponding node (or/and, branch/fork) and are summarized in figure 6.

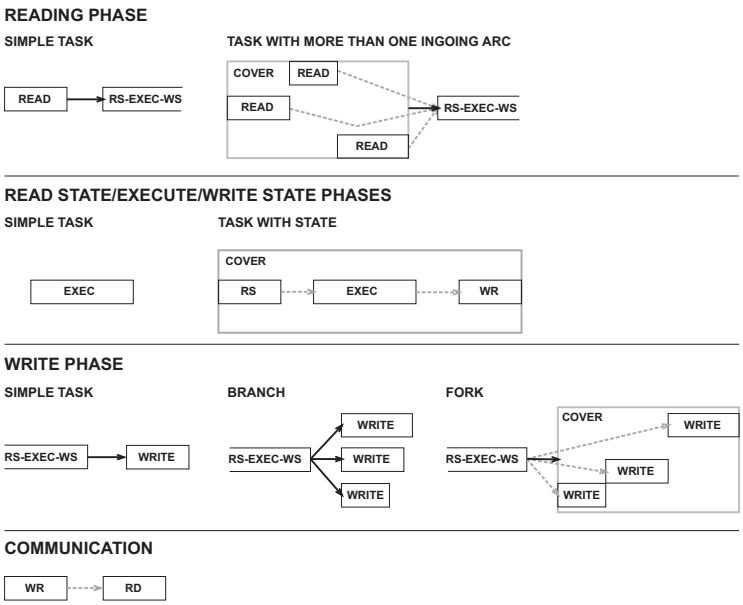

Fig. 6. Each task has a five phase behavior

In the picture the black arrows represent immediate precedence relations ($end(A) = start(B)$), while the gray hyphened arrows are simple precedence relations ($end(A) \leq start(B)$). Sometimes we used ILOG cover activities (which start with the first activity of a set and end with the last one) to define activity groups and model particular precedence relations or resource usage constraints; in detail:

- a cover activity is used to state that the execution must start immediately after reading the last queue
- in a task with state, the PE is used by a cover activity and not by the single R/W state and execute activities
- for an and task, a cover activity is used to state that all the queues must be written in any order after the execution phase

Each activity duration is an input parameter and can vary depending on the allocation of internal state and program data. The processing elements are unary resources: we modeled them defining a simple disjunctive constraint proposed in [10].

The bus, as in [5], is modeled as a cumulative resource, according with the so called "additive model", which allows an error less than 10% until bandwidth usage is under 60% of the real capacity. Computing the bus usage in presence of alternative activities is not trivial, since the bus usage varies in a not linear way and every activity can have its own bus view (see fig 7).

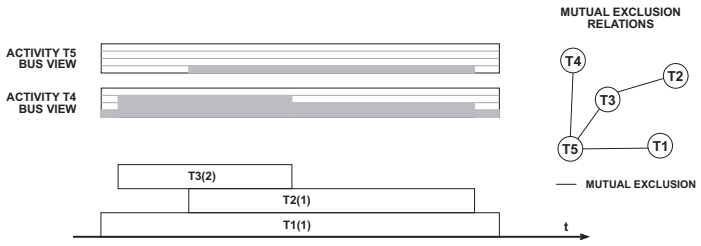

Fig. 7. Activity bus view

Suppose for instance we have the five tasks of figure 7; activities **T1**, **T2**, **T3** have already been scheduled: the bus usage for each of them is reported between round brackets, while all the mutual exclusion relations are showed on the right. Let's consider activity **T4**, which is not mutually exclusive with any of the scheduled tasks. As long as only **T1** is present, the bus usage is 1. It becomes 1 + 2 = 3 when also activity **T3** starts, but when both **T1**, **T2** and **T3** execute the bus usage remains 3, since **T2** and **T3** are alternative. Thus the bus usage at a given time is always the maximum among all the combinations of not alternative running tasks. Furthermore, let's consider activity **T5**: since it is mutually exclusive with all tasks but **T2**, it only sees the bus usage due to that task. Therefore the bus view at a given time depends on the activity we are considering.

We modeled the bus creating a new global timetable constraint for cumulative resources and conditional tasks in the not preemptive case. The global constraint keeps a list of all known entry and exit points of activities: given an activity A, if $lst(A) \leq eet(A)$ then the entry point of A is $lst(A)$ and $eet(A)$ is its exit point (where lst stands for latest start time and so on).

Let A be the target activity: A3 scans the interval $[est(A), finish)$ checking the bus usage at all entry points (as long as $good = true$). If it finds an entry point with not enough bandwidth left it starts to scan all exit points ($good = false$) in order to

determine a new possible starting time for activity A. If such an instant is found its value is stored ($lastGoodTime$) and the finish line is updated (step 4.2.2.2), then A3 restarts to scan other entry points, and so on. When the finish line is reached the algorithm updates $est(A)$ or fails. A3 has $O(a(c+b))$ complexity, where a is the number of activities, b the one of branches, c the number of conditions. The algorithm can be easily extended to update also $let(A)$: we tried to do it, but the added filtering is not enough to justify the increased propagation time.

algorithm: Propagation of the cumulative resource constraint with alternative activities (A3)

1. $time = est(a), finish = eet(a)$
2. $latestGoodTime = time$
3. $good = true$
4. While $\neg\,[(good = false \wedge time > lst(a)) \vee (good = true \wedge time >= finish)]$:
  4.1. if $busreq(a) + usedBandwith > busBandwidth$:
    4.1.1. $time =$ next exit point
    4.1.2. $good = false$
  4.2. else:
    4.2.1. $time =$ next entry point
    4.2.2. if $good = false$:
      4.2.2.1. $lastGoodTime = time$
      4.2.2.2. $finish = max(finish, time + mindur(a))$
      4.2.2.3. $good = true$
5. if $good = true$: $est(a) = lastGoodTime$
6. else: $fail$

end

A3 is able to compute the bandwidth usage seen from each activity in $O(a)$ by taking advantage of a particular data structure we introduced, named Branch Fork Graph (BFG).

A BFG is a directed, acyclic, bichromatic graph which captures the control structure of a conditional task graph. It is composed of alternate branch ("B") nodes (representing a choice point) and fork ("F") nodes, representing conjunctions. Arcs coming out of a "B" node are labeled with conditions. A BFG is always rooted at a for node.

Given a conditional task graph (augmented with a fake root node r) a BFG can be get by first building the "F" type root node ($\overline{r}$), and running the following algorithm with $n = r, \overline{n} = \overline{r}$:

algorithm: BF Graph Construction ($BFGC(n, \overline{n})$)

1. for each branch task n_b in the original graph connected to n by means of non conditional arcs:
  1.1. if the corresponding "B" node $\overline{n}_b$ has already been built:
    1.1.1. connect $\overline{n}_b$ to $\overline{n}$
  1.2. else:
    1.2.1. add a "B" node $\overline{n}_b$ to the BFG
    1.2.2. connect $\overline{n}_b$ to $\overline{n}_f$
    1.2.3. for each outgoing arc e with source n_b and destination n_d:
      1.2.3.1. add an "F" node $\overline{n}_f$ to the BFG
      1.2.3.2. connect $\overline{n}_f$ a $\overline{n}_b$ with condition $c(e)$
      1.2.3.3. run $BFGC(n_d, \overline{n}_f)$

end

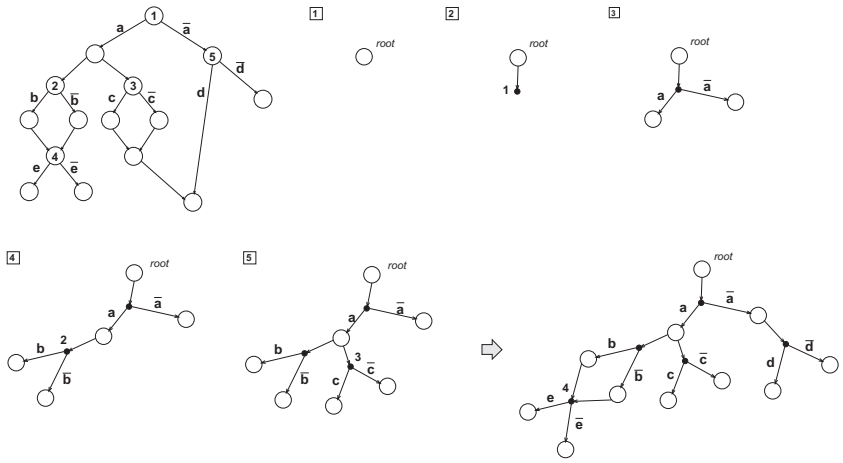

Fig. 8. *Construction of a BF Graph*

An example of BFG construction is showed in figure 8: first (step 2) the "B" node corresponding to the branch node *1* is added, than two "F" nodes (one for outgoing arc) are built and connected by means of conditional arcs. In the same way also the nodes corresponding to branches *2* and *3* are built and connected, and so on. Note that the "B" node corresponding to branch *4* has two ingoing arcs, thus showing that the BFG is actually a graph and not a tree.

In order to work correctly BFGC requires the graph to satisfy a particular condition (called "Control Flow Uniqueness") which states that each "and" node must have a main ingoing arc, whose activation implies the activation of the other ingoing arcs. Figure 9A shows a graph which satisfies the CFU and figure 9B one which doesn't.

Note that the CFU is not a very restrictive condition since it is satisfied by every graph resulting from the natural parsing of programs written in a language such C++ or Java.

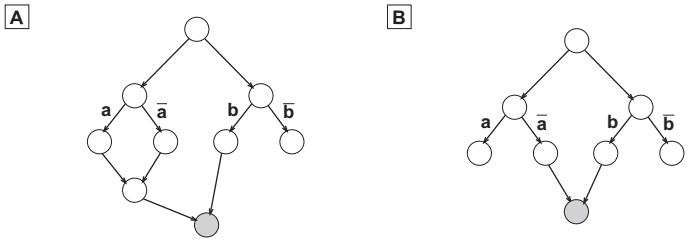

Fig. 9. *A graph that satisfies the control flow uniqueness (with the corresponding BFG) and one which doesn't (with the* supposed *corresponding BFG)*

Every node or arc in the original CTG can be "loaded" on one or more "F" nodes in the BF graph by means of the following algorithm, which takes as input arguments the AS of the given node or arc and a node of the BFG (initially $\overline{n} = root$).

algorithm: load(AS, $\overline{n}$)

1. if $\overline{n}$ is a fork node:
  1.1. for each child branch node $\overline{n}_b$:
    1.1.1. if $\neg query(\overline{n}_b, AS)$ then $load(\overline{n}_b, AS)$
  1.2. if AS was not loaded on any child node, load it on current node
2. if $\overline{n}$ is a branch node:
  2.1. for each child fork node $\overline{n}_f$ (let $c_{\overline{n}_f}$ be the condition on the corresponding arc):
    2.1.1. if $c_{\overline{n}_f} \in AS$ then $load(\overline{n}_f, AS)$
  2.2. if AS was loaded on at least one node return some indication of that

end

algorithm: query($\overline{n}$)

1. if $\overline{n}$ is a fork node:
  1.1. if $\overline{n}$ has no successor branch node return $true$
  1.2. for each successor branch node $\overline{n}_b$
    1.2.1. $query(\overline{n}_b, AS)$
  1.3. if at least one $query$ operation was successfull: return $true$
  1.4. else: return $false$
2. if $\overline{n}$ is a branch node:
  2.1. for each successor fork node $\overline{n}_f$ (let $c_{\overline{n}_f}$ be the condition on the arc):
    2.1.1. if $c_{\overline{n}_f} \notin AS$: return $false$
    2.1.2. else: $query(\overline{n}_f, AS)$
  2.2. if at least one $query$ operation was not successful: return $false$
  2.3. else: return $true$

end

Figures 10 A and B show examples of the result of a node load operation: in principle each node or arc should be loaded on all nodes that can be reached by a path including only conditions in the AS.

In practice, however, to avoid useless replications, if an AS would be loaded on all the conditions connected to a given fork node, than the AS can be safely mapped on the node itself. For example in figure 10A the node 4 can be safely loaded on the root of the BFG.

The `load` algorithm takes this into account by means of the `query` subalgorithm. When processing an "F" node, `load` queries all children nodes: when invoked on a "B" node the `query` algorithm returns *true* iff the given AS would be loaded on all the outgoing arcs; if all the children of the current "F" node answer "yes" to the query then the AS can be loaded on the current node. Each time a "B" child answers "no", then `load` is recursively executed.

Since every activity (reading an input queue, executing, etc.) in the scheduling model corresponds to a node or an arc, *a BFG can be used also to load activities*, depending on their activation set.

Given a BFG loaded with scheduled activities, the bandwidth usage seen by a non scheduled activity can be computed by executing the following algorithm on the root "F" node.

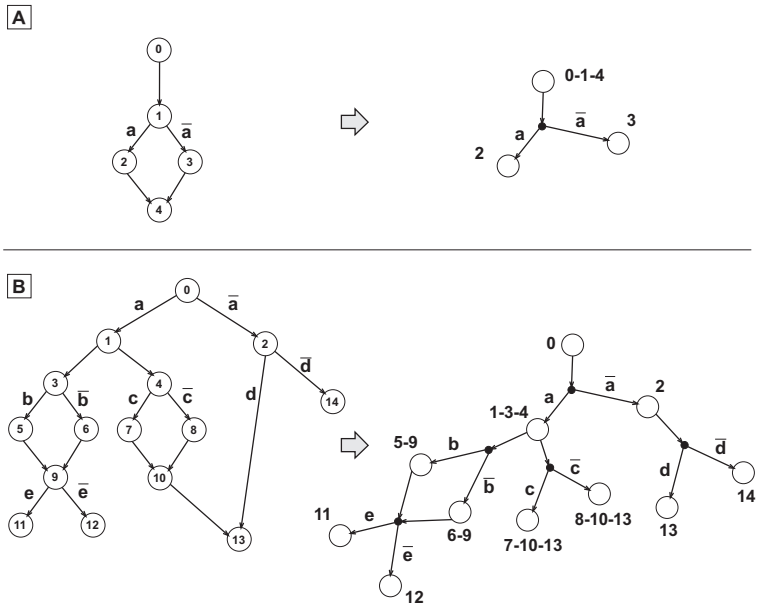

Fig. 10. Node loading on a BFG

algorithm: maxusage($\overline{n}$, $EX(AS)$)

1. if $\overline{n}$ is a fork node
  1.1. $U = \sum_{a \in M} u(a)$, with $M = \{a \mid a$ is loaded on $\overline{n}\}$
  1.2. for each child node $\overline{n}_i$:
    1.2.1. $U = U + maxusage(\overline{n}_i, EX(AS))$
  1.3. return U
2. if n is a branch node
  2.1. $U = 0$
  2.2. for each child node $\overline{n}_i$ with condition c_i:
    2.2.1. if $c_i \notin ES(AS)$: $U = max(U, maxusage(\overline{n}_i, ES(AS))$
  2.3. return U

end

In the algorithm description $u(a)$ is the bandwidth usage of activity a, while $EX(AS)$ is the exclusion set of the reference activity. Basically, `maxusage` visits the BFG, choosing at each "B" node the outgoing branch with maximum usage and skipping non compatible arcs according to $EX(AS)$.

The control flow uniqueness condition ensures that the best choice at every "B" node depends only on the descendant nodes, and thus every branch node can be independently processed once all its descendants. Figure 11 shows a weighed graph which does not satisfy CFU: if both branches (gray) nodes are processed independently after their descendants the chosen outcomes would be $\{\overline{a}, b\}$, while the best ones are $\{a, \overline{b}\}$.

To compute the bus usage at a given instant t it's sufficient to take into account only activities that are surely running at time t (i.e. for which $lst(a) \leq t \leq eet(a)$).

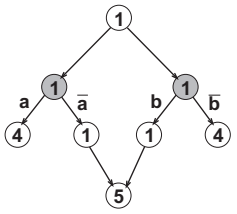

Fig. 11. Weighted graph which does not satisfy CFU condition

To compute maximum usage we have to process each activity, thus `maxusage` complexity is at least $O(a)$. Since every activity can be replicated during the loading phase, the worst case complexity is upper bounded by $O(a(b+c))$; note however that in practice the number of replication is likely to be much less than $b+c$.

3.3 Benders cuts and subproblem relaxation

Each time the master problem solution is not feasible for the scheduling subproblem a cut is generated which forbids that solution. Moreover, all solutions obtained by permutation of PEs are forbidden, too.

Unfortunately, this kind of cut, although sufficient, is weak; this is why we decided to introduce another cut type, generated as follows: (1) solve to feasibility a single machine scheduling model with only one PE and tasks running on it; (2) if there is no solution the tasks considered cannot be allocated to any other PE.

The cut is very effective, but we need to solve an NP-hard problem to generate it: however, the solution time appears to be strongly dependent on the number of iterations; in this condition it is likely to be worthy to compute an expensive cut, provided it is strong enough. Moreover, in practice, the cut generation problem can be quickly solved.

Again with the objective to limit iteration number, we also inserted in the master problem a relaxation of the subproblem. This consists of constraints of two kinds:

1. $\forall$ track t $\quad \sum_{n \in t} dur_n(M_n, S_n, C) \leq deadline$

   Memory devices cannot be allocated in such a way that the total length of a track is greater than the deadline (see fig. 12B). The expression of $dur_n(M_n, S_n, C_n)$ is an upper bound on the duration of all the activities (input, output, execution, state, ..) related to node/task n.

2. $\forall$ set of independent tasks $I, \forall j = 0, .., p-1$ $\quad \sum_{n \in I} dur_{nj}(M_{nj}, S_{nj}, C_j) \leq deadline$

   The allocation cannot be such that a set of non mutually exclusive tasks whose duration is greater than the deadline is allocated on a single PE (see fig. 12C): non mutually exclusive tasks cannot overlap and must execute in sequence if they are on the same PE. Again $dur_n(M_n, S_n, C_n)$ is an upper bound on the duration of the activities related to node n

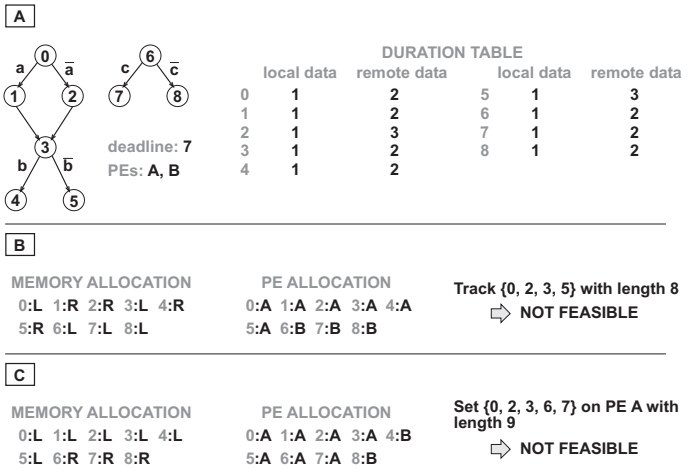

Fig. 12. Two allocation forbidden by scheduling problem relaxation

Note that both the number of tracks and the number of sets of non mutually exclusive tasks are exponential: if all the relaxation constraints were added the allocation problem would be overconstrained. To mitigate the problem, only set and tracks able to exceed deadline are considered; moreover, the relaxation cuts are dynamically added during the solution process ($\sqrt{n}$ every iteration, with n number of nodes).

4 Experimental results

We implemented all exposed algorithms in C++, using the state of the art solvers ILOG Cplex 9.0 (for ILP) and ILOG Solver 6.0 and Scheduler 6.0 (for CP). We tested all instances on a Pentium IV pc with 512MB RAM. The time limit for the solution process was 30 minutes.

We tested the method on two set of instances: the first ones are characterized by means of a synthetic benchmark; peculiar input data of this problem (such as the branch probabilities) were estimated via a profiling step. Instances of this first group are only slightly structured, i.e. they have very short tracks and quite often contain singleton nodes: therefore we decided to generate a second group of instances, completely structured (one head, one tail, long tracks)[1].

The results of the tests on the first group are summarized in table 1. Instances are grouped according to the number of activities (acts); beside this, the table reports also the number of processing elements (PEs), the number of instances in the group (inst.), the instances which were proven to be infeasible (inf.), the mean overall time (in seconds), the mean time to analyze the graph (init), to solve the master and the subproblem, to generate the no-good cuts and the mean number of iterations (it). The solution times are of the same order of the deterministic case (scheduling of Task Graphs), which is a very good result, since we are working on conditional task graphs and thus dealing with a stochastic problem.

For a limited number of instances the overall solving time was exceptionally high: the last column in the table shows the number of instances for which this happened, mainly due to the master problem (A), the scheduling problem (S) or the number of iterations (I). The solution time of these instances was not counted in the mean; in general it was greater than than thirty minutes.

Although this extremely high solution time occurs with increasing frequency as the number of activities grows, it seems it is not completely determinated by that parameter: sometimes even a very small change of the deadline or of some branch probability makes the computation time explode.

We guess that in some cases, when the scheduler is the cause of inefficiency, this happens because of search heuristic: for some input graph topologies and parameter configurations the heuristic does not make the right choices and the solution time dramatically grows. Perhaps this could be avoided by randomizing the solution method and by using restart strategies [19].

The results of the second group of instances (completely structured) are reported in table 2. In this case the higher number of arcs (and thus of precedence constraints) reduces the time windows and makes the scheduling problem much more stable: no instance solution time exploded due to the scheduling problem. On the other hand the increased number of arcs makes the allocation more complex and the scheduling problem approximation less strict, thus increasing the number of iterations and their duration. In two cases we go beyond the time limit.

We also ran a set of tests to verify the effectiveness of the cuts we proposed in section 3.3 with respect to the basic cuts removing only the solution just found: table 3

[1] All instances are available at http://www-lia.deis.unibo.it/Staff/MichelaMilano/tests.zip

acts	PEs	inst.	inf.	time	init	master	sub	nogood	it	A/S/I
10-12	2	6	0	0.0337	0.0208	0.0075	0.0027	0.0027	1.1667	0/0/0
13-15	2	8	1	0.5251	0.1600	0.0076	0.0040	0.0020	1.1250	0/0/0
16-18	2-3	12	0	0.1091	0.0922	0.0089	0.0067	0.0013	1.0833	0/0/0
19-21	2-3	14	1	0.1216	0.0791	0.0279	0.0079	0.0046	1.2143	0/0/0
22-24	2-3	23	4	0.2336	0.1520	0.0259	0.0061	0.0081	1.1739	0/0/0
25-27	2-3	16	3	1.7849	0.0319	1.7285	0.0108	0.0088	1.3125	0/0/0
28-30	2-3	13	2	0.3331	0.0284	0.0770	0.1900	0.0338	1.6667	0/1/0
31-33	3-4	4	2	0.3008	0.2303	0.0510	0.0040	0.0000	1.0000	0/0/0
34-36	3-4	13	4	0.6840	0.0204	0.4245	0.0132	0.0108	1.2308	0/0/0
37-39	3-4	7	0	1.5670	0.0399	1.2010	0.1384	0.1877	4.4286	0/0/0
40-42	3-4	6	3	2.9162	0.0182	0.5857	2.2267	0.0390	1.6667	0/0/0
43-45	3-4	6	1	5.3670	0.2757	4.8200	0.0630	0.2005	4.1667	0/0/0
46-48	4-5	11	0	3.2719	0.0508	0.6913	2.4616	0.0683	2.0000	1/2/0
49-51	4-5	11	1	1.9950	0.1840	1.7900	0.0071	0.0087	1.1111	1/1/0
52-54	5-6	6	0	8.0000	1.3398	1.5743	4.8788	0.2073	2.7500	1/1/0
55-67	6	8	0	2.2810	0.8333	1.4377	0.0100	0.0000	1.0000	1/4/0

Table 1. Results of the tests on the first group of instances (slightly structured)

acts	PEs	inst.	inf.	time	init	master	sub	nogood	it	A/S/I
20-29	2	7	2	0.5227	0.0200	0.0134	0.0090	0.0021	8.8571	0/0/0
30-39	2-3	6	0	1.7625	0.0283	1.2655	0.2057	0.2630	5.8333	0/0/0
40-49	3	3	0	0.4380	0.0313	0.3493	0.0573	0.0000	1.0000	0/0/0
50-59	3-4	7	0	1.1403	0.0310	0.6070	0.2708	0.2315	3.6667	0/0/1
60-69	4-5	4	0	10.1598	0.0385	6.8718	1.2798	1.9698	18.0000	0/0/0
70-79	4-5	4	0	88.9650	0.0428	88.6645	0.2578	0.0000	1.0000	0/0/0
80-90	4-6	7	0	202.4655	0.0755	184.0177	6.5008	11.8715	28.6667	0/0/1

Table 2. Result of the tests on the second group of instances (completely structured)

mean time to gen. a cut					
basic case:					0.0074
with relaxation based cuts (RBC):					0.0499
	number of iterations		**excution time**		
deadline	**basic case**	**with RBC**	**basic case**	**with RBC**	**result**
8557573	2	3	1.18	0.609	opt. found
625918	1	1	0.771	0.765	opt. found
590846	1	1	0.562	0.592	opt. found
473108	19	6	6.169	1.186	opt. found
464512	190	14	201.124	9.032	opt. found
454268	195	24	331.449	10.189	opt. found
444444	78	15	60.747	6.144	opt. found
433330	9	4	4.396	1.657	opt. found
430835	5	3	3.347	1.046	opt. found
430490	5	3	3.896	1.703	opt. found
427251	3	2	2.153	0.188	inf.

Table 3. Number of iterations without and with scheduling relaxation based cuts

reports results for a 34 activities instance repeatedly solved with a decreasing deadline values, until the problem becomes infeasible. The iteration number greatly reduces. Also, despite the mean time to generate a cut grows by a factor of ten, the overall solving time per instance is definitely advantageous with the tighter cuts.

Finally, to estimate the quality of the chosen objective function (bus traffic expected value), we tested it against an easier, heuristic technique of deterministic reduction. The chosen heuristic simply optimizes bus traffic for the scenario when each branch is assigned the most likely outcome; despite its simplicity, this is a particularly relevant technique, since it is widely used in modern compilers ([20]).

We ran tests on three instances: we solved them with our method and the heuristic one (obtaining two different allocations) and we computed the bus traffic for each scenario with both the allocations. The results are shown in table 4, where for each instance are reported the mean, minimum and maximum quality improvement against the heuristic method. Note that on the average our method always improves the heuristic solution; moreover, our solution seems to be never much worse then the other, while it is often considerably better.

			quality improvement		
instance	**activities**	**scenarios**	**mean**	**min**	**max**
1	53	10	4.72%	-0.88%	13.08%
2	57	10	2.59%	-0.11%	8.82%
3	54	24	12.65%	-0.72%	39.22%

Table 4. Comparison with heuristic deterministic reduction

5 Conclusion and future works

We have proposed a stochastic method for planning and scheduling in the stochastic case. The method proposed has two main contributions: the first is a polynomial transformation of a stochastic problem into a deterministic one based on the conditional task graph analysis. Second, the implementation of two constraints for unary and cumulative resources in presence of conditional activities. We believe the results obtained are extremely encouraging. In fact, computation times are comparable with the deterministic version of the same instances. We still have much work to do: first we have to solve the extremely hard instances possibly through randomization; second we have to take into account other aspects where stochasticity could come into play, like task duration which could not be known in advance. Third, we have to validate these results on a real simulation platform to have some feedback on the model.

Acknowledgement This work has been partially supported by MIUR under the COFIN 2005 project *Mapping di applicazioni multi-task basate su Programmazione a vincoli e intera*.

References

1. Wu, D., Al-Hashimi, B., Eles, P.: Scheduling and mapping of conditional task graph for the synthesis of low power embedded systems. In: Computers and Digital Techniques, IEE Proceedings. Volume 150 (5). (2003) 262–273
2. Eles, P.P.P., Peng, Z.: Bus access optimization for distributed embedded systems based on schedulability analysis. In: International Conference on Design and Automation in Europe, DATE2000, IEEE Computer Sociery (2000)
3. Shin, D., Kim, J.: Power-aware scheduling of conditional task graphs in real-time multiprocessor systems. In: International Symposium on Low Power Electronics and Design (ISLPED), ACM (2003)
4. Wolf, W.: The future of multiprocessor systems-on-chips. In: In Procs. of the 41st Design and Automation Conference - DAC 2004, San Diego, CA, USA, ACM (2004) 681–685
5. Benini, L., Bertozzi, D., Guerri, A., Milano, M.: Allocation and scheduling for MPSOCs via decomposition and no-good generation. In: Proc. of the Int.l Conference in Principles and Practice of Constraint Programming. (2005)
6. Hooker, J.N., Ottosson, G.: Logic-based benders decomposition. Mathematical Programming **96** (2003) 33–60
7. Baptiste, P., Pape, C.L., Nuijten, W.: Constraint-Based Scheduling. Kluwer Academic Publisher (2003)
8. Vilim, P., Bartak, R., Cepek, O.: Extension of o(n.log n) filtering algorithms for the unary resource constraint to optional activities. Constraints **10** (2005) 403–425
9. Laborie, P.: Algorithms for propagating resource constraints in ai planning and scheduling: Existing approaches and new results. Journal of Artificial Intelligence **143** (2003) 151–188
10. Kuchcinski, K.: Constraints-driven scheduling and resource assignment. ACM Transactions on Design Automation of Electronic Systems **8** (2003)
11. Ahmed, S., Shapiro, A.: The sample average approximation method for stochastic programs with integer recourse. In: Optimization on line. (2002)
12. Laporte, G., Louveaux, F.V.: The integer l-shaped method for stochastic integer programs with complete recourse. Operations Research Letters **13** (1993)
13. Benders, J.F.: Partitioning procedures for solving mixed-variables programming problems. Numerische Mathematik **4** (1962) 238–252
14. Norkin, V.I., Pflug, G., Ruszczynski, A.: A branch and bound method for stochastic global optimization. Mathematical Programming **83** (1998)
15. Walsh, T.: Stochastic constraint programming. In: Proc. of the European Conference on Artificial Intelligence, ECAI. (2002)
16. Tarim, A., Manandhar, S., Walsh, T.: Stochastic constraint programming: A scenario-based approach. Constraints **11** (2006) 53–80
17. Culler, D.A., Singh, J.P.: Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann (1999)
18. Compton, K., Hauck, S.: Reconfigurable computing: A survey of systems and software. ACM Computing Surveys **34** (1999) 171–210
19. Gomes, C.P., Selman, B., McAloon, K., Tretkoff, C.: Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. In: AIPS. (1998) 208–213
20. Faraboschi, P.; Fisher, J.A.; Young, C.: Instruction scheduling for instruction level parallel processors. In: Proceedings of the IEEE , vol.89, no.11 pp.1638-1659, Nov 2001
21. M. Lombardi, M. Milano Stochastic Allocation and Scheduling for Conditional Task Graphs in MPSoCs. Technical Report 77 LIA-003-06.