Alma Mater Studiorum - Università di Bologna

Dottorato di Ricerca in
Ingegneria Elettronica, Informatica e delle Telecomunicazioni

Ciclo XXII

Settore Scientifico Disciplinare: ING-INF/05

Hybrid Methods for Resource Allocation and Scheduling Problems in Deterministic and Stochastic Environments

Michele Lombardi

Il Coordinatore di Dottorato:
Paola Mello

I Relatori:
Paola Mello
Michela Milano

Esame Finale Anno 2008/2009

Cando penso que te fuches,
negra sombra que me asombras,
ó pé dos meus cabezales
tornas facéndome mofa.

Cando maxino que es ida,
no mesmo sol te me amostras,
i eres a estrela que brila,
i eres o vento que zoa.

Si cantan, es ti que cantas,
si choran, es ti que choras,
i es o marmurio do río
i es a noite i es a aurora.

En todo estás e ti es todo,
pra min i en min mesma moras,
nin me abandonarás nunca,
sombra que sempre me asombras.

(Rosalía De Castro)

Contents

Abstract

This work presents exact, hybrid algorithms for mixed resource Allocation and Scheduling problems; in general terms, those consist into assigning over time finite capacity resources to a set of precedence connected activities.

The proposed methods have broad applicability, but are mainly motivated by applications in the field of Embedded System Design. In particular, high-performance embedded computing recently witnessed the shift from single CPU platforms with application-specific accelerators to programmable Multi Processor Systems-on-Chip (MPSoCs). Those allow higher flexibility, real time performance and low energy consumption, but the programmer must be able to effectively exploit the platform parallelism. This raises interest in the development of algorithmic techniques to be embedded in CAD tools; in particular, given a specific application and platform, the objective if to perform optimal allocation of hardware resources and to compute an execution schedule.

On this regard, since embedded systems tend to run the same set of applications for their entire lifetime, off-line, exact optimization approaches are particularly appealing. Quite surprisingly, the use of exact algorithms has not been well investigated so far; this is in part motivated by the complexity of integrated allocation and scheduling, setting tough challenges for "pure" combinatorial methods. The use of hybrid CP/OR approaches presents the opportunity to exploit mutual advantages of different methods, while compensating for their weaknesses.

In this work, we consider in first instance an Allocation and Scheduling problem over the Cell BE processor by Sony, IBM and Toshiba; we propose three different solution methods, leveraging decomposition, cut generation and heuristic guided search. Next, we face Allocation and Scheduling of so-called Conditional Task Graphs, explicitly accounting for branches with outcome not known at design time; we extend the CP scheduling framework to effectively deal with the introduced stochastic elements. Finally, we address Allocation and Scheduling with uncertain, bounded execution times, via conflict based tree search; we introduce a simple and flexible time model to take into account duration variability and provide an efficient conflict detection method.

The proposed approaches achieve good results on practical size problem, thus demonstrating the use of exact approaches for system design is feasible. Furthermore, the developed techniques bring significant contributions to combinatorial optimization methods.

Chapter 1

Introduction

This work presents exact, hybrid algorithms for mixed resource Allocation and Scheduling problems; in general terms, those consist into assigning over time finite capacity resources to a set of precedence connected activities. Despite the proposed methods have broad applicability, the main motivation for their development comes from the field of embedded system design, where Allocation and Scheduling (A&S) covers a key role.

1.1 Context

The transition in high-performance embedded computing from single CPU platforms with custom application-specific accelerators to programmable Multi Processor Systems-on-Chip (MPSoCs) is now a widely acknowledged fact; this revolution is motivated by the evidence that traditional approaches to maximize single processor performance have reached their limits. However, the multicore promise of real time performance with limited power consumption comes at a condition; namely, the programmer must be able to effectively exploit the exposed platform parallelism. With hundred core devices being planned by major producers for the near future, squeezing out the full performance of modern MPSoCs is a far from straightforward task.

Virtually all key markets in data-intensive embedded computing are in desperate need of expressive programming abstractions and tools enabling programmers to take advantage of MPSoC architectures, while at the same time boosting productivity. This raises interest in the development of automatic optimization techniques to be embedded in CAD tools. A specific optimization problem covering a key role in the design flow of embedded systems is that of optimizing a given software application for a target platform; this usually requires to optimally allocate hardware resources (such as processors, storage devices and communication channels) and to provide an optimal execution schedule. The actual cost metric can vary considerably: achieving minimum execution time or minimum power consumption are among the most common design objectives.

Due to the large production numbers often involved, small design improvements can result in huge savings; moreover, a typical embedded system runs the same set of applications for its whole life time, thus allowing time consuming analysis and improvements to be performed prior to the deployment. Those peculiarities make off-line, exact optimization approaches particularly appealing.

Quite surprisingly, the use of exact algorithms has not been well investigated by the Embedded System community, whereas heuristic approaches are widely employed. This is in part motivated by the complexity of integrated allocation and scheduling problems, setting tough challenges for existing combinatorial methods.

Part of such a high complexity is motivated by the tight integration between the assignment and the scheduling problem component. A&S problems have been faced by Constraint Programming (CP) as scheduling problems with alternative resources. Despite effective filtering and search techniques have been developed for pure scheduling, adding resource mapping decisions hinders constraint propagation and widens the search space, often making the problem overly difficult. On the other hand, Operations Research (OR) methods are successfully employed to solve large pure assignment problems; however, adding the time dimension sets serious issues to the computation of efficient and tight bounds, and counters the effectiveness of most OR approaches. In this context, the use of hybrid CP/OR algorithms presents an opportunity to exploit the advantages of different approaches, while compensating for their mutual weaknesses. It is interesting to note that, in the Embedded System community, the opportunities offered by hybrid approaches are nearly unexploited.

A second major obstacle to the use of exact methods for Embedded System design is the issue of predictability. Many embedded applications run under real-time constraints, i.e. deadlines that have to be met for any possible execution; this is the case of many safe-critical applications, such as in the automotive and the aircraft industries. At the same time, every system naturally exhibits some degree of randomness. On one side, failing to cope with such uncertainty may make the use of a strong optimization algorithm simply not worth the effort, due to the incapacity to predict the actual objective function value. On the other hand, including probabilistic elements in the solution approach generally increases the complexity, taking real size problems out of the reach of complete methods.

1.2 Content

We tackle a class of mixed Allocation & Scheduling problems consisting into (1) assigning a pool of finite capacity resource to a set of precedence connected activities (2) computing a time and resource feasible schedule. This problem type covers a critical step in the Hardware/Software co-design of modern embedded systems, namely, optimizing an input application for a target hardware platform. The main content in this work is articulated in three parts.

A&S over Cell BE (Chapter 3): in first instance we consider a software allocation and scheduling problem over a real platform (namely, the Cell BE processor by Sony, IBM and Toshiba), with a makespan minimization objective. We propose an exact approach leveraging decomposition and heterogeneous techniques, and we observe how a classic two stage decomposition approach does not perform adequately. Hence we recursively split the problem and solve the resulting subparts via two ILP and a CP solver, cooperating to improve feasible solutions until optimality is proved. The three solvers are arranged in a nested scheme and

communicate via linear cuts, obtained through the iterative solution of relaxed NP-hard problems.

As an alternative, we propose a CP method building upon a classical OR heuristic algorithm and making use of no-good learning techniques. The two realized solvers obtain nice results on complementary classes of instances; hence a third hybrid approach is introduced, combining the advantages of the former ones and achieving even higher scalability.

A&S of Conditional Task Graphs (Chapter 4): Next, we face allocation and scheduling of so-called Conditional Task Graphs (CTG); those are kind of project graphs explicitly taking into account the presence of conditional branches, and originating alternative control flows. Since the exact outcome of the branches is not known prior to the execution, a stochastic model must be used for off-line optimization.

We define a framework to enable polynomial time probabilistic reasoning over CTGs, provided a condition referred to as Control Flow Uniqueness holds (CFU); we show how CFU is usually satisfied in many realistic settings. The probabilistic framework is then used to devise methods for the deterministic reduction of the stochastic problem objective; a key observation with this regard is that the considered embedded system design problems are one-stage, i.e. all decisions must be taken off-line and the constraints must be satisfied for all execution scenarios. The satisfiability requirement for all possible branch outcomes is guaranteed by introducing a novel class of *conditional constraints*. We show how the proposed techniques enable up to one order of magnitude speed ups compared to a state of the art method (scenario based optimization).

A&S with Uncertain Durations (Chapter 5): Finally, we address allocation and scheduling with uncertain durations. In particular, we assume the execution time of the tasks to be scheduled is unknown, but a priori bounded. The assumption goes along with an emerging trend in embedded system platform design, allowing for the removal of some performance enhancement techniques (e.g. caches) to improve predictability. This provides support for off-line analysis tools, enabling the computation of actual worst case and best case execution times.

We consider mapping and scheduling problems over clustered platforms with point to point communication channels; a hybrid algorithm is devised, encompassing a heuristic allocation step and a complete scheduling stage. The allocation is performed by casting the mapping problem to balanced graph partitioning, and by using a state-of-the art partitioning heuristics. The complete scheduling approach leverages the Precedence Constraint Posting method and performs search by systematically identifying and resolving possible resource conflicts.

The key step in this process (conflict detection) in a first stage is modeled and solved via Constraint Programming; the method effectively guides search, but requires the solution of an NP-hard problem at each node. Hence we devise a second approach, coupling a polynomial time (complete) detection algorithm with a local search (heuristic) conflict minimization step. The proposed solver achieves significant results on a set of synthetic benchmarks, obtained with a generator designed to produce realistic problems.

1.3 Contribution

This work situates itself at the boundary between combinatorial optimization and embedded system design. Despite algorithmic techniques (in particular CP and OR) are with no doubt the main topic, their consistent application to mapping and scheduling of software applications to MPSoCs brought significant contributions to this research field as well.

Concerning Embedded System Design: All the proposed approaches target a problem of strong practical interest; in fact, not only optimal A&S allows a better exploitation of hardware resources, but it may enable the use of less powerful and more energy efficient platforms (design space exploration).

In particular, the practical advantages of a mapping and scheduling approach rely on the possibility to include the method within a CAD tool. An effective ad hoc approach may fail to deliver its full potential, if not sufficiently flexible to deal with the complex constraints set by the target architecture, user design choices and the need to provide accurate models. The use of Constraint Programming and powerful integration and decomposition techniques provides the developed approaches with the necessary flexibility. In particular, nasty architecture dependent side constraints can be handled easily in the allocation stage, while the use of CP allows to capture the complex temporal relations required to have accurate models. The advantage is neat if the hybrid approach is compared with classical OR scheduling methods, often reaching efficiency by targeting specific cases of an extremely fine grained problem classification. In our case, the exposed flexibility is so high that targeting user customizable architectures via automatic ILP/CP model generation can be considered doable on the near future.

We show that many practical size allocation and scheduling problems are within the reach of complete solvers, thus contradicting a well established belief in Embedded System Design. Spending considerable amount of time in performing extensive optimization is well worth the effort at system deployment time. In case of tighter design time constraints (e.g. during application development), all the proposed approaches can be stopped before optimality is proved and still provide pretty good solutions; additionally, partial optimality proof (e.g. gaps) can be returned as well.

We demonstrate that in several practical cases elements of uncertainty (conditional branches and variable durations) can be taken into account without drastically increasing problem complexity. Indeed, the very presence of hard real time requirements and the relative simplicity of run time policies (due to efficiency issues) often allow the use of analytical techniques to obtain deterministic reductions. Usually this can be done in polynomial time, thus keeping the design problem NP-hard (instead of P-SPACE as stochastic problems often happen to be).

Concerning Combinatorial Optimization: Heterogeneous algorithmic techniques (ILP, no-good learning, filtering, local search) can be combined to achieve significant speed-ups. In the case at hand, hybridization allows to tackle complex mixed allocation and scheduling problems, where classical CP and OR approaches find tough difficulties. Moreover, despite the presented approaches

mainly target mapping and scheduling problems over embedded systems, many of the developed techniques have broader applicability.

We investigate the use of the Logic Based Benders' Decomposition (LBD) scheme in the context of scheduling problems with general precedence relations. It is a common opinion that LBD approaches tend to pay off only when decoupling the problem allows to further split the subproblem into smaller independent components; we show that even if this is not the case, LBD can still be worthwhile if the decomposition simplifies the master and subproblem models and enables their efficient solution. Most notably, this consideration led here to a multi-stage LBD approach, where an initially very complex master problem is further decomposed achieving orders of magnitude speed-ups. The main reason for this effect is that the efficiency of a combinatorial algorithm for NP-hard problems decreases non-linearly as the problem size grows.

Moreover, we show how the use of a lower bound of the overall objective as cost function for the master problem is not a requirement for the LBD scheme; in fact, if the subproblem relaxation subsumes any bound which could be used in the Benders' cuts, then replacing the classical cuts with strong no-goods and using a more easy to optimize cost function in the master problem becomes a better option.

Finally, we show how the cut generation procedure does not need to have polynomial complexity in general; indeed, we exploit a no-good refinement method requiring the repeated solution of NP-hard problems with very good results. The key point is that the only actual requirement is to save more solving time by adding cuts than needed for their generation; as a consequence, building cuts by solving an NP-hard problem is perfectly feasible, as long as it is sufficiently easy. We provide empirical guidelines for tuning such a process.

We devise a framework for efficient probabilistic reasoning over conditional graphs; in the considered case study this allows the reduction of a stochastic cost function to a deterministic functional, thus preventing the problem complexity from becoming P-SPACE. The proposed reduction method applies well to many one-stage stochastic problems over conditional graphs.

Moreover, we introduce a class of so-called *conditional constraints*, enforcing local consistency on a set of decision variables for every possible instantiation of a set of random variables. In the context of scheduling problems, we devise a conditional timetable constraint, taking advantage of the probabilistic reasoning framework to provide efficient filtering.

Finally, similar ideas are used to devise an *expected makespan* constraint (where "expected" is used in its probability theory sense); despite the expected makespan does not admit a polynomial declarative formulation, we demonstrate how our probabilistic framework can be used to devise a polynomial procedure to compute valid bounds.

Perhaps counterintuitively, we show that taking into account uncertain, bounded, durations with hard deadline constraints can be done in an efficient manner; in particular, a simple polynomial time propagation method can be used to enforce consistency on a temporal network with both deterministic and stochastic constraints.

Variable durations raise however non-trivial issues at search time; namely, providing a schedule by fixing start times is no longer an option. Conflict

resolution via Precedence Constrain Posting can be applied, but dealing with the (exponential) number of task sets causing resource overusage becomes an issue. In this context, we show how a polynomial time conflict check can be done by solving a minimum flow problem. The outcome of the checking method is a maximal set, which can be used as input for a refinement process or to perform fixed parameter tractable enumeration.

Chapter 2

Related Work

This chapter provides an overview of existing research related to allocation and scheduling problems, with a focus on exact approaches.

As embedded system provides the main application fields for the method discussed in this work, Section 2.1 is devoted to a discussion of the typical optimization problems arising in their design flow. A brief discussion of the adopted formalisms and of the existing approaches is given as well. Constraint programming and constraint based scheduling are the algorithmic techniques most widely used throughout this work; those are presented in Sections 2.2 and 2.3. Several of the main OR approaches to allocation and scheduling problems are reported in Section 2.4; finally, some hybrid methods for combinatorial optimization (most notably, Logic based Benders' Decomposition) are presented in Section 2.5. Note that more application specific references are given in Chapters 3, 4 and 5.

2.1 A&S for Embedded System Design

With the term "embedded system" we refer to any information processing device embedded into another product. In the last years their diffusion has been growing at a surprising rate: automotive control engines, mobile phones, multimedia electronic devices are only few examples of the ubiquity of such devices. Being widely employed in portable devices for real time applications, these systems need both energy efficient and high performance hardware. Moreover, the large variety of different devices calls for flexible and easily customizable platforms; this is true despite embedded devices are not general purpose, but tend to run a set of predefined applications during the entire system lifetime.

Mixed Hardware / Software design and Multiprocessor Systems on Chips come as the most promising solution to the mentioned issues. In particular, *mixed HW/SW design* promotes a view of dedicated hardware and software as equally good tools to implement application specific functions. Avoiding dedicated hardware as much as possible allows a single general purpose platform to be used in completely different embedded systems; additionally this approach boosts the design phase, cuts the time-to-market and reduces maintenance costs.

The drawback for the flexibility provided by a software implementation of real time functions is the need for general purpose platforms having *both high performance and low power consumption.* As a consequence, the last ten years witnessed a shift in the design of integrated architectures from single processor

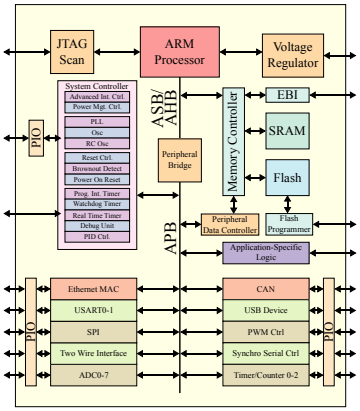


Figure 2.1: An ARM processor based System on Chip

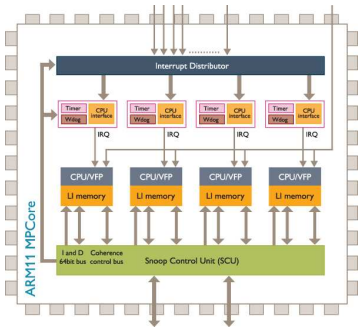


Figure 2.2: Block diagram for the ARM11 MPCore processor

systems to multiprocessors. This revolution is motivated by the evidence that traditional approaches to maximize single processor performance have reached their limits. Power consumption, which has stopped the race in boosting clock frequencies for monolithic processor cores [Hor07] [HSN+02] [Mud01] [HMH01] and design complexity are the most critical factors that limit performance scaling [Bor99].

However, Moore's law continues to enable doubling in the number of transistors on a single die every 18-24 months. Consequently, designers are turning to multicore architectures (so called *Multi-Processor Systems on Chip* – MPSoCs) to satisfy the ever-growing computational needs of applications within a reasonable power envelope [Bor07]. MPSoCs are multi core, general purpose, systems realized on a single chip; each core has low energy consumption and limited computational power: real time performance is achieved by extensive parallelization. MPSoCs integrate on a single silicon die both the multiple core, memory, the communication infrastructure, DMAs and input/output interfaces.

Following Moore's law, the semiconductor industry roadmap foresees a doubling in the number of core units per die with every technology generation [Gra07]. This trend is noticeable both in mainstream [Cor02, AMD], as well as in embedded computing [Sys, Sem, Tec, ST, NEC]. MPSoCs with previously unseen complexity, hosting a huge number of processors to provide scalable computational power, are planned for the near future.

On the other hand consumer applications are characterized by tight time-to-market constraints and extreme cost sensitivity. Hence, there is a strong need for software optimization tools that can optimally exploit the available resources [Mar06]. The current design methodology for multicore systems on chip is hampered by a lack of appropriate Computer Aided Design tools, leading to low efficiency and productivity (the so called *Design Productivity Gap*). Software optimization is a key requirement for building cost- and power-efficient electronic systems, while meeting tight real-time constraints and ensuring predictability and reliability, and is one of the most critical challenges in today's high-end computing.

2.1.1 Optimization and Embedded System Design

Optimization problems arise in many step of the design process of an embedded system; specifically, in this work we focus on optimally exploiting hardware parallelism. The key to successful application deployment lies in effectively mapping concurrency in the application to the architectural resources provided by the platform and providing an efficient run-time execution schedule; this task often goes under the name of *mapping and scheduling problem*. Once an optimal mapping and scheduling procedure is available, investigating different platform options becomes feasible; this is a second optimization problem, often referred to in the literature as *design space exploration*. Figure 2.3 shows how a simple design space exploration algorithm could be realized by repeatedly solving a mapping and scheduling problem.

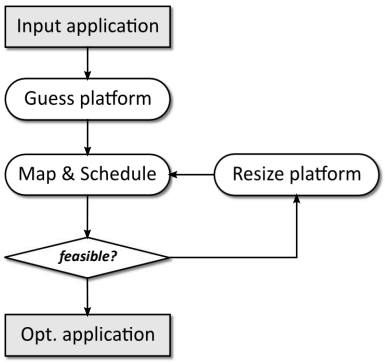


Figure 2.3: Basic design space exploration algorithm

Existing mapping and scheduling methods usually assume the *input application to be described with a proper abstract formalism*, explicitly exposing the application concurrency. In particular, embedded system software may exhibit significant concurrency at different levels of granularity; task level concurrency (at the level of functions and procedures) is often evident in data-intensive embedded applications, which motivates the focus on task-based application abstractions throughout this work. Similarly, an *abstract description of the target platform* is assumed to be available. In detail, the class of design problems we consider consists in (1) optimally allocating the resources take in the hardware description to the tasks in the application description, and (2) providing an optimal task execution order.

Off-line vs on-line approaches The design process described above (and adopted throughout this work) implicitly assumes an optimal mapping and schedule are computed before the actual execution, i.e. *off-line*; moreover the method requires many details about the platform and the target application to be known a priori. With no doubt this is a drawback; nevertheless, off-line approaches still are very appealing for embedded system design; the reason is that such devices usually run a set of predefined applications during their entire lifetime. Therefore it is reasonable to spend a significant amount of time in optimizing software compilation once for all, improving the performance of the overall system.

Alternatively, one may focus on devising an effective policy to perform the same operations *at run-time*; this leads to so called *on-line approaches*. In this case, allocation and scheduling decisions must be taken upon arrival of tasks in the system, and the employed 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 [GJ+79], obviously, on-line approaches cannot guarantee optimality, but they focus on safe acceptance tests, i.e. a schedulable task set may be rejected but a non-schedulable task set will never be accepted.

Section outline A large variety of mapping and scheduling methods exists in the literature; the approaches differ in the adopted application abstraction, the application feature, the target platform and its model, the applied optimization techniques and the problem objective. While attempting a classification is out of the scope of the chapter, in the followings we try to mention at least the most relevant features in the context of this work.

In particular, due to the large number and variety the existing target platforms, we chose to provide the description of a few practical hardware models directly in Chapters 3, 4 and 5. As an intuition, however, hardware is usually abstracted as a set of renewable resources, subject to additional architecture dependent constraints. Conversely applications models can actually be divided into a small number of main classes; a selected number of the most relevant ones is reported in the following Section 2.1.2.

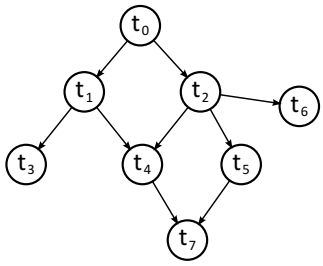


Figure 2.4: A Task Graph

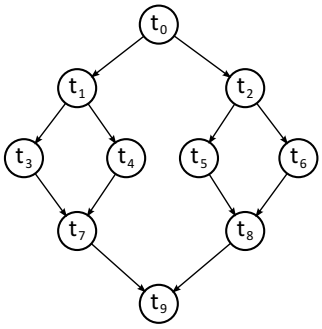


Figure 2.5: A structured Task Graph

2.1.2 The Task Graph: definition and semantic

The target application to be executed on top of the hardware platform is often represented throughout this work as a Task Graph (TG, see Figure 2.4 for an example). This latter [ERL90] consists of a directed acyclic graph (DAG) pointing out the parallel structure of the program; specifically, the application workload is partitioned into computation sub-units denoted as *tasks*, which are the nodes of the graph. Graph edges connecting any two nodes indicate task dependencies due, for example, to communication and/or synchronization. Formally, a TG is a pair $\langle T, A \rangle$, where T is the set of graph nodes t_i (referred to as *tasks*) and A is the set of graph arcs, represented as pairs (t_i, t_j). Tasks and

arcs are usually annotated by attributes; typically at least a duration value is specified for each task.

Task Graphs can be assumed to have a single starting node (a dummy node may be inserted in case more than one source node exists), while several sink nodes may exist. As an exception, a particular subclass of TGs (so-called *structured* Task Graphs) are constrained to have, for each task t_i with more than one successor (fork node), a single node t_j where all the paths originating from t_i merge (join node). Figure 2.5 shows an example of structured Task Graph. Such TGs often result from the automatic parsing of computer code.

Task Graph Semantics Task graphs are usually associated a self-timed execution *semantics*, i.e. a task t_i starts as soon as all input data are available. Actual inter-task communication often occurs by writing/reading data to a communication buffer in a shared memory device; in data intensive applications, accessing the communication buffer can take non-negligible time and hence require to be be taken into account for the model to be meaningful.

In vast part of the literature tasks are assumed to have deterministic execution time; in practice, however, the duration of each function/procedure usually depends on the input data. In some cases failing to capture this behavior may make the TG abstraction not sufficiently accurate and compromise the effectiveness of an optimization approach. An established off-line method to cope with run time data dependencies is to replace fixed execution time with random variables having a known probability distribution. The same type of issues motivated the proposal of so-called Conditional Task Graphs (CTG – see [XW01]), explicitly modeling the presence of conditional branches by means of probability theory concepts. TGs with variable durations and CTGs are taken into account in Chapter 4.

2.1.2.1 Other Application Abstractions

Other application abstractions are commonly used in the literature; those will be mentioned and briefly described, but not discussed in detail, as this is out of the scope of the work. Indeed, Task Graphs happen to be the least expressive (and more manageble) in a ranking of proposed abstractions.

Kahn Process Networks At the top of such ranking are *Kahn Process Networks (KPN)*, first introduced in [Kah74] (see Figure 2.6 for an example); a KPN consists of a directed graph (non necessarily acyclic). Nodes represent processes, communicating through FIFO queues (represented by the arcs) and synchronizing via a blocking-read operation. A KPN has deterministic structure, enabling the use of graph information to compute optimized partitionings and schedules. For some examples of design approaches relying on KPNs see [TBHH07, SZT+04].

Synchronous Data Flow Graphs A popular restriction of the KPN model are *Synchronous Data Flow Graphs (SDFG)*, proposed in [LM87] (see Figure 2.7 for an example). SDFGs are directed, often cyclic graphs representing communicating autonomous processes (the graph nodes, referred to as *actors*). Inter process communication is explicitly represented by the graph arcs; their endpoints are labeled with integer *rates*. The communication is modeled through

the exchange of homogeneous *tokens* (the basic data unit); rates on the source of the arcs represent the amount of token produced by the source node when it executes (*fires*), while rates on the destination of the arcs are the amount of token required by the destination node to fire. A node can fire whenever there are enough tokens on all its input arcs, and output tokens are produced at the end of the execution. SDFG actors are often bound to execute as soon as it is possible (*self-timed execution*); under this hypothesis, after an initial transition time, an SDFG enters a periodic phase.

Task Graphs are a specialization both of Kahn Process Networks and SDFGs; in particular, TGs are not well suited to model periodic behaviors, but are more amenable for optimization and time based schedule computation.

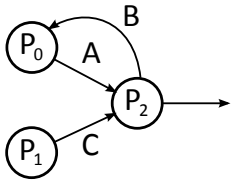


Figure 2.6: A Kahn Process Network; both processes and FIFO queues are named.

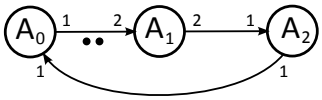


Figure 2.7: A synchronous data flow graph; rates are reported on the arc endpoints, the solid black circles represent the tokens

2.1.3 Existing Mapping and Scheduling Approaches

The synthesis of system architectures has been extensively studied in the past. Mapping and scheduling problems of Task Graphs on multi-processor systems have been traditionally tackled by means of Integer Linear Programming (ILP). In general, even though ILP is used as a convenient modeling formalism, there is consensus on the fact that pure ILP formulations are suitable only for small problem instances, because of their high computational cost. An early example is represented by the SOS system, which used mixed integer linear programming technique (MILP) [PP92]. A MILP model that allows to determine a mapping optimizing a trade-off function between execution time, processor and communication cost is reported in [Ben96].

Heuristics The complexity of pure ILP formulations for general Task Graphs has led to the development of heuristic approaches. Those provide no guarantees about the quality of the final solution, and many times the need to bound search times limits their applicability to moderately small task sets. A comparative study of well-known heuristic search techniques (Genetic Algorithms, Simulated Annealing and Tabu Search) is reported in [Axe97]. A scalability analysis of these algorithms for large real-time systems is introduced in [KWGS]. Many heuristic scheduling algorithms are variants and extensions of list scheduling [EKP+98].

Several structures of Task Graph have been considered in the literature; the pipelined workload, typical of several real applications, has been studied, for example in [FR97] (and in the complete approach in [BBGM05b]); in [CV02] a

retiminig heuristic is used to implement pipelined scheduling, while simulated annealing is used in [PBC04] (where both periodic and aperiodic task are considered).

Different platforms and Task Graphs have been considered: [Tho01] considers a multi-processor platform where the processors are dissimilar. Energy-aware platforms have been studied in several works; the first DVS approach for single processor systems which can dynamically change the supply voltage over a continuous range is presented in [YDS95]. More recent works on the argument can be found in [XMM05, JG05, ASE+04, ABB+06, BBGM06], to cite a few. A good survey can be found in [BBDM02].

Complete off-line approaches (other than ILP) Constraint Programming (CP) was applied to optimally solving mapping and scheduling problem in [Kuc03]. The work in [SK01] is based on Constraint (Logic) Programming to represent system synthesis problem, and leverages a set of finite domain variables and constraints imposed on these variables. Both ILP and CP techniques can claim individual successes but practical experience indicates that neither approach dominates the other in terms of computational performance. The development of a hybrid CP-IP solver that captures the best features of both would appear to offer scope for improved overall performance. However, the issue of communication between different modeling paradigms arises. One such method is Logic Benders Decomposition (LBD), described in Section 2.5.1; [RGB+06] presents a LBD approach in the context of MPSoC systems. The authors tackle the mapping sub-problem with ILP and the scheduling one with CP. The work considers only pipelined streaming applications and does not handle Task Graphs. In order to solve the problem of allocating and scheduling a general Task Graph onto a MPSoC, the introduction of more complex problem models, cost functions and Benders' cuts is needed, and is tackled for example in [BBGM06].

On-line approaches An excellent and comprehensive description of the online approach is given in [Liu00], and can be summarized as follows. When an application (i.e. a Task Graph), enters the system, it is first partitioned on processor clusters using a fast heuristic assignment (e.g. greedy first-fit bin packing). Then schedulability is assessed on a processor-by-processor basis. First, local task deadlines are assigned, based on a deadline assignment heuristic (e.g. ultimate deadline), then priorities are assigned to the tasks, and finally a polynomial-time schedulability test is performed to verify that deadlines can be met. It is important to stress that a given Task Graph can fail the schedulability test even if its execution would actually meet the deadline. This is not only because the test is conservative, but also because several heuristic, potentially sub-optimal decisions have been taken before the test which could lead to infeasibility even if a feasible schedule does exist. One way to improve the accuracy of the schedulability test is to modify synchronization between dependent tasks by inserting idle times (using techniques such as the release-guard protocol [SL96]) that facilitate worst-case delay analysis. Recent work has focused on techniques for improving allocation and reducing the likelihood of failing schedulability tests even without inserting idle times [FB06, Bar06].

2.2 Constraint Programming

Constraint Programming is an AI (Artificial Intelligence) method designed to solve Constraint Satisfaction Problems (CSP). Generally speaking, a CSP is a triple $\langle X, D, C \rangle$, where X is a set of variables, D a set of domains and C a set of constraints. Each domain D_i is a set of values v which can be assumed by the corresponding variables X_i; each constraint C_j in the set C specifies values which can be assumed by a subset of the variables, referred to as the *scope* of the constraint; the scope is denoted as $S(C_j)$ and the constraint is said to be *posted* on $S(C_j)$. Constraints can be given either in extensional form (i.e. list of tuples) or in intensional form (e.g. $X_i \leq X_k$). A *solution* to the CSP is an assignment of the problem variables, compatible with every constraint.

Filtering and propagation Similarly, a Constraint Programming *model* consists variables, domains and constraints; a *filtering algorithm* is associated to each problem constraint C_j and is able to remove provably infeasible values from the domain of variables in its scope. Note that, from a more abstract perspective, filtering may be seen as the process of deducing additional constraint from existing constraints. When a domain D_i gets modified by a filtering algorithm, such modification has a chance to allow another constraint involving X_i to filter out more values on the variables in its scope; such mechanism is know as constraint propagation. Many algorithms have been proposed to efficiently perform this propagation step (see [BC94]); the process stops when a fixpoint is reached.

Search and Local Consistency In Constraint Programming, a CSP is usually solved via tree-search; during the process, each branching choice triggers filtering and constraint propagation, effectively reducing the search space and often resulting in tremendous speed-ups over complete enumeration. The use of some kind of search technique is a requirement, for a twofold reason: in first place, in case the problem has multiple solutions, propagation and filtering cannot (by definition) arbitrarily discard any of them. In second place, eliminating all infeasible values in a CSP is generally as complex as solving the problem itself; hence, in practice one has to resort to enforcing some kind *local consistency* (see [DB05]) and use search to get to an actual solution.

Many different types of local consistency have been defined on this purpose; the most widely employed ones are *Arc (and Generalized Arc) Consistency (AC)*, *Bound Consistency (BC)* and *K-Consistency*. Namely:

Definition 1 (Arc Consistency)**.** *A binary constraint C_j on X_i, X_k is arc consistent if and only if, for each value v in D_i, there exist a value w in D_j such that the couple (v, w) is allowed (we say that v has a support in D_k). Formally:*

$$\forall X_i \in S(C_j), \forall v \in D_i \ : \ \exists w \in D_j \ : \ (v, w) \in C_j$$

Arc Consistency can be generalized to non binary constraint, leading to Generalized Arc Consistency (GAC):

Definition 2 (Generalized Arc Consistency)**.** *A constraint C_j with scope $S(C_j)$ (and $S(C_j) = X_0, \ldots X_{n-1}$) is generalized arc consistent if and only if, for every variable $X_i \in S(C_j)$, every value $v \in D_i$ has a support in $D_0 \times D_{i-1} \times D_{i+1} \times \ldots D_{n-1}$.*

In the following, we use the notation $\Pi\left[S(C_j) \setminus D_i\right]$ to denote the cartesian product $D_0 \times D_{i-1} \times D_{i+1} \times \ldots D_{n-1}$ (i.e. the set of all combinations of values from the domains in the constraint scope, X_i excluded). In Bound Consistency we assume the domain of every variable X_i is an interval defined by a minimum and maximum value (respectively l_i and u_i); then:

Definition 3 (Bound Consistency)**.** *A constraint C_j is bound consistent if and only if, for each variable $X_i \in S(C_j)$, both the extreme values l_i and u_i have a support in $\Pi[S(C_j) \setminus D_i]$. Formally:*

$$\forall X_i \in S(C_j) : \begin{cases} \exists \bar{w}' \in \Pi\left[S(C_j) \setminus D_i\right] \text{ such that } l \parallel \bar{w}' \in C_j \\ \exists \bar{w}'' \in \Pi\left[S(C_j) \setminus D_i\right] \text{ such that } u \parallel \bar{w}'' \in C_j \end{cases}$$

Where the expression $v \parallel \bar{w}$ denotes the concatenation of value v with a tuple $\bar{w}$. Note that the supports w' and w'' are not required to be include only extreme values for each $(D_k \in S(C_j) \setminus D_i)$. In general Bound Consistency is weaker than (Generalized) Arc Consistency, but the equivalence holds in some practical context (e.g. scheduling with usual precedence constraint). K-consistency is the most complete form of consistency and requires that:

Definition 4 (K-Consistency)**.** *A constraint C_j is k-consistent if and only if, for each variable X_i in the scope $S(C_j)$, every tuple $\bar{v} \in \Pi S(C_j) \setminus D_i$ has a support w in D_i. Formally:*

$$\forall X_i \in S(C_j), \forall \bar{v} \in \Pi\left[S(C_j) \setminus D_i\right] \; : \; \exists w \in D_i \text{ such that } w \parallel \bar{v} \in C_j$$

Briefly, in order to solve a Constraint Satisfaction Problems by means of Constraint Programming, one needs (1) to devise a model and (2) to specify how search is performed. Hence the equation:

$$\texttt{CP} = \texttt{model} + \texttt{search}$$

which suggests that model and search are *independent* in CP. More precisely, a single CP model may be solved with several search strategies, and vice-versa a single search strategy may be applied to several CP models.

2.2.1 Modeling in CP

Constraint Programming provides a rich and expressive language, effectively enabling the formulation of very compact models. Variables and constraints are basically subject to no restriction; as a consequence, in the literature one may find, besides the usual integer and real variables, more exotic items such as set-variables [Ger94] or interval-variables [LRS$^+$08]. The most common situation however, is to have variables with finite, integer domains (this combination is often referred to as CP-FD).

Each type of variable comes with a specific pool of constraints, ranging from simple linear constraints to much more complex relations; as an example, in the literature one may find:

1. linear mathematical constraints, e.g. $X_i \leq X_j + X_k + 1$;

2. non-linear mathematical constraints, e.g. $X_i = |X_j^3 - X_k|$, $X_i \neq X_j$, $X_i = \max\{X_j, X_k\}$;
3. basic constraints related to specific domains, e.g. $X_i \subseteq X_j$ for set variables, `non_overlapping`(X_i, X_j) for interval variables;
4. meta-constrained or "reified" constraints, denoting a 0/1 value and used within a mathematical or logical expression, e.g. $X_i \leq 10 \times (X_j = 2)$, or $(X_i = 1) \leq (X_j = 1)$ (which models a logical implication).

Global Constraints A particularly interesting case is that of so called *global constraints*. A global constraint factorizes a set of homogeneus constraints; for example, the well known `alldiff` constraint is posted on a set of variables $X_0, X_1 \ldots$ and factorizes a full network of inequality constraints:

$$\forall X_i, X_j \text{ with } i \neq j \ : \ X_i \neq X_j$$

this has a self-evident advantage on the expressiveness side, allowing a compact formulation of large sets of constraints. However, the main point for using global constraints if that *enforcing local consistency for the whole set is superior to enforcing consistency for each single constraint individually.* As an example, consider the following simple CSP on finite domains:

$$X_0 \neq X_1, X_0 \neq X_2, X_1 \neq X_2$$
$$X_0, X_1 \in \{1, 2\}$$
$$X_2 \in \{1, 2, 3\}$$

By enforcing Arc Consistency on each constraint individually, no domain can be reduced, as for each variable X_i, all values v in the domain have a support in X_j, in the context of every single constraint $X_i \neq X_j$. However, by considering all constraints at the same time, one may note that values 1 and 2 must be assigned to X_0, X_1 in order to have a feasible solution, and D_2 can be reduced to $D_2 = \{3\}$. The improved filtering provided by global constraints may result in a large performance difference, in particular when solving large size problems.

Additionally, global constraints often provide efficient filtering algorithm taking into account all variables in the scope at the same time. Heterogeneous algorithmic techniques may used to perform the filtering: for examples [Rég94] gives the first polynomial time GAC algorithm for the `alldiff` constraint, based on matching theory. The CP framework supports seamless integration of such a diversity of methods. For a first state of the art on global constraint see [Sim96], while [Rég03, HK06] provide more recent surveys and [BCR05] an exhaustive catalog.

2.2.2 Searching in CP

Constraint Programming allows one to adopt a wide range of search schemes, effectively casting into a CP framework algorithms devised in a completely different context. Many of the search strategies used in practice, however, belong to of three main categories: Backtrack Search, Local Search and Dynamic Search; here, the two former approaches will be briefly described, while Section 2.2.2.1 is entirely dedicated to backtrack search. For a comprehensive overview of CP search approaches, see [Bee06].

Local Search Local search approaches include Large Neighborhood Search (see [Sha98] and Section 2.3.3) and the methods used in the Comet Search Language (see [VHM05]). In the former, given an incumbent solution, CP is used to solve a neighborhood, defined by relaxing some decisions (e.g. by making some variables free). The Comet search language is mainly based on the use of *Soft Constraints*, i.e constraints admitting some degree of violation and providing a measure of such degree.; Comet search proceeds by trying to minimize the total problem violation degree, until a feasible solution is found.

Dynamic Programming Dynamic Programming search (DP) for CP mainly comprises Bucket Elimination [Dec99] and Non-Serial Dynamic Programming [BB72]. Both methods basically propose the same idea in different flavors: as a constraint C_j may be seen as the list of tuples it allows for the variables in its scope $S(C_j)$, a backtrack free search method can proceed by combining constraints into macro-constraints, until all problem solutions have been enumerated. Unfortunately, the approach has exponential time *and* space complexity; by properly ranking problem variables, however, the method is exponential only in a parameter of the constraint network known as *width* (which is usually much smaller than the number of variables).

2.2.2.1 Backtrack Search

As a matter of fact, most CP approaches are based on some variant of Backtrack Search (BS) and Depth First Search in Particular (DFS). The main advantage of DFS over DP is to retain polynomial space complexity. A BS approach proceeds by posting (usually) unary constraints on variables, triggering propagation at each step and backtracking upon failure. Basic design choices for a BS method are the type of unary constraint posted at each search node (a.k.a. *branching strategy*) and the criterion to choose the actual constraint (i.e. the *search heuristic*). A number of different techniques such as Backjumping and No-good Recording, Restarts and Randomization may be adopted as well.

Branching Strategy We can refer as *enumeration*, to the branching strategy consisting in choosing at each step a variable X_i and posting unary constraints in the form $X_i = v$, $\forall v \in D_i$. By adopting choice points in the form $X_i = v$, $X_i \neq v$ one gets so-called *binary choice points*; finally, *domain splitting* is a branching strategy dividing at each step the domain of a variable X_i by posting constraints $X_i \leq \theta$, $X_i > \theta$. See Figure 2.8 for a pictorial description of the three strategies.

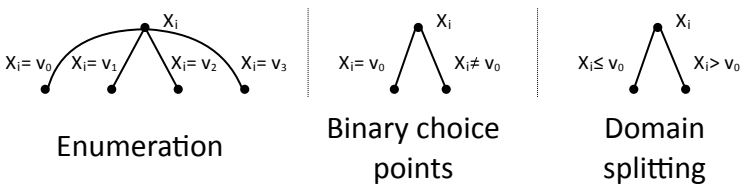


Figure 2.8: Three of the most commonly used branchinng strategies

Search Heuristics In all mentioned branching strategies, a criterion must be specified to evaluate both variables and values; variable selection heuristics have probably received so far the highest attention. A quite commonly accepted idea is to select as soon as possible the variable where it is more likely to fail; this goes under the name of "First-Fail Pricinple" (first appearing in [HE80]). A quite common embodiment of such idea is the *dom* heuristic (see [GB65]), choosing at each step the variable with the smallest domain. The heurstics was refined in [Bré79], where ties in the selection are resolved by giving precedence to the variable with the highest degree, i.e. involved in the highest number of non-bound constraints; this is usually referred to as $dom + deg$, or Brelaz heuristics. More recently, [BR96] proposed to use the degree to weight the domain size rather then to break ties (dom/deg).

Many modern heuristics try to gather some knowledge about the problem prior to its solution (or during the search process itself); as an example, so-called Impact Based Search (see [Ref04]) performs a diving step before problem solution to evaluate the impact of each variable/value pair (X_i, v); such impact is the relative search space reduction resulting from the assignment of v to X_i. Alternatively, the weighted domain-degree heuristic (proposed in [BHLS04] and referred to as $dom/wdeg$) builds upon dom/deg, but tries to learn constraint weights during search.

Nogood-Learning, Backjumping Extracting information from the problem is also at the base of no-good learning and backjumping techniques. The main underlying idea is, once a failure occurs, to analyze and deduce the cause of failure; this is called an *explanation* and can be given in a pure no-good format ($\{X_i = v \wedge X_j = w \ldots\}$, see [Bac00]) or as a generalized one [KB05]. The process can be performed by treating propagation as a black-box (as done in QuickXplain [Jun04], but is usually more efficient to take exploit knowledge about the problem constraint structure. Once a possible cause of the failure is known, backjumping [Gas78] consists in backtracking directly to the closest assignment invalidating the explanation.

Randomization and Restarts Perhaps counterintuitively, randomization and restarts are powerful techniques which can be used both to boost and to stabilize the behavior of a BS method. Randomization consists in introducing some degree of randomness in the choice of the variable/value to branch on (e.g. by randomly breaking ties); work [Gom04] provides a nice overview of the topic. A randomized search method is usually restarted from time to time, to prevent a bad choice taken in the early steps from sinking the overall performance. On this purpose, effective universal restart strategies both with and without strong theoretical support have been proposed (see [LSZ93, Wal99]).

Alternatives to Depth First Search Finally, BS alternatives to pure Depth First Search have been proposed as well; as an example in Limited Discrepancy Search [HG95], given a reference heuristics, the search space is explored by allowing an increasing number of decisions to be taken against the heuristic. Decomposition Based Search (see [HM04]) generalizes the previous approach by using the reference heuristic to classify values of each variable into "good"

and "bad"; initially, all variables are required to assume a good value, while an increasing number of bad values is allowed as the search proceeds.

2.3 Constraint Based Scheduling

According to Baker (see [Bak74]), a scheduling problem consists in allocating scarce resources to activities over time. We refer as Constraint-Based Scheduling [BLPN01] to to the use of CP techniques to solve scheduling problems. The equation `CP = model + search` is pretty much valid in this context as well; therefore, CP contributions to scheduling can be classified into *modeling techniques* and *search strategies*.

2.3.1 A CP Model for Cumulative Scheduling

Here, we mostly deal with Cumulative Non-preemptive Scheduling problems, involving *activities* with general precedence relations; throughout this section, the term "activity" and "task" will be used with the same meaning, while this is not the case for other sections in the work. The term "cumulative" tells that finite capacity resources are considered; resources are additive, i.e. the resource requirement of a sets of tasks simultaneously running is the sum of their resource requirements. Unit capacity (unary) resources are a particular case of cumulative resources. The term "non-preemptive" means that activities are non-interruptible once they have started.

Scheduling problems over a set of activities/tasks $T = \{t_0, t_1, \ldots\}$ and resources $R = \{r_0, r_1, \ldots\}$ are classically modeled in CP by introducing for every activity t_i three integer variables (see [BLPN01]):

- $\mathtt{S}_i$ representing the activity start time (i.e. the first time instant where the activity is executing)
- $\mathtt{E}_i$ representing the activity end time (i.e. the first time instant where the activity is not executing)
- $\mathtt{D}_i$ representing the activity duration (i.e. the number of time instants where the activity is executes)

"Start", "end" and "duration" variables must satisfy the constraint $\mathtt{E}_i = \mathtt{S}_i + \mathtt{D}_i$. Quite commonly activity durations are fixed values (say d_i) and hence $\mathtt{D}_i = d_i$. Bounds for the "start" and "end" variables are referred to by means of conventional names; the detailed list follows:

$$\begin{aligned}
&\min(\mathtt{S}_i) = EST(t_i) \text{ (Earliest Start Time)}\\
&\max(\mathtt{S}_i) = LST(t_i) \text{ (Latest Start Time)}\\
&\min(\mathtt{E}_i) = EET(t_i) \text{ (Earliest End Time)}\\
&\max(\mathtt{E}_i) = LET(t_i) \text{ (Latest End Time)}
\end{aligned}$$

The EST and LET value may be forced by the user, thus effectively setting a *release time* and a *deadline* on the activity execution. Precedence constraint can be modeled by means of simple linear constraints $\mathtt{E}_i \leq \mathtt{S}_j$. Limitations due to finite capacity resources are usually take into account by introducing, for each

$$
\begin{array}{rll}
\text{min:} & F(T) & \\
\text{subject to:} & \mathtt{E}_i = \mathtt{S}_i + \mathtt{D}_i & \forall t_i \in T \\
& \mathtt{E}_i \leq \mathtt{S}_i & \forall t_i \prec t_j \\
& \mathtt{cumulative}([\mathtt{S}_i], [\mathtt{D}_i], [rq_{ik}], C_k) & \forall r_k \in R \\
\text{with:} & \mathtt{S}_i, \mathtt{E}_i \in \{0, \ldots eoh\} & \\
& \mathtt{D}_i = d_i &
\end{array}
$$

Figure 2.9: A CP model for cumulative, non-preemptive scheduling with fixed durations

resource $r_k \in R$ a global `cumulative` constraint:

$$\mathtt{cumulative}([\mathtt{S}_i], [\mathtt{D}_i], [rq_{ik}], C_k)$$

where $[\mathtt{S}_i]$ and $[\mathtt{D}_i]$ are the arrays of activity start and duration variables, $[rq_{ik}]$ tells the resource requirements of each activity t_i and C_k is the capacity of the considered resource. The `cumulative` constraint guarantees the resource capacity is not exceeded at any point of time by the considered activities. For sake of simplicity, let us assume all $\mathtt{S}_i$ variables range between 0 and *eoh* (End Of Horizon); then the constraint enforces:

$$\forall \tau = 0, \ldots eoh \ : \sum_{\mathtt{S}_i \leq \tau < \mathtt{S}_i + \mathtt{D}_i} rq_{ik} \leq C_k \tag{2.1}$$

In other words, for each time instant τ, the sum of the requirements rq_i of activities executing at time τ ($\mathtt{S}_i \leq \tau < \mathtt{S}_i + \mathtt{D}_i$) must not exceed the resource capacity C_k. We define a *solution* to the scheduling problem (a.k.a. schedule) as a feasible assignment of all $\mathtt{S}_i$ and $\mathtt{E}_i$ variables. Figure 2.3.1 show a complete example of a quite usual CP based model for cumulative, non-preemptive scheduling problem with fixed durations. Note that The function F to be minimized has been intentionally left non-specified, as several type of objectives could be considered; moreover, thanks to the flexibility of the CP approach, a wide number of different scheduling problem variants can be taken into account by simply adding side-constraints.

2.3.1.1 Objective Function Types

Here we give a brief description of the objective functions most commonly used in cumulative scheduling problems. Probably, the most frequently occurring one is the *makespan* (or overall *completion time*); this is defined as the worst completion time among activities to be scheduled; hence:

$$F(T) = \max_{t_i \in T}(\mathtt{E}_i)$$

Note that, if a single activity t_{i^*} with no successor exists, its completion time indeed is the makespan value: $\mathtt{E}_{i^*} = makespan$. In the general case where the are more of such activities, one may want to minimize the *total weighted completion time*, defined as the weighted sum of the completion time of activities

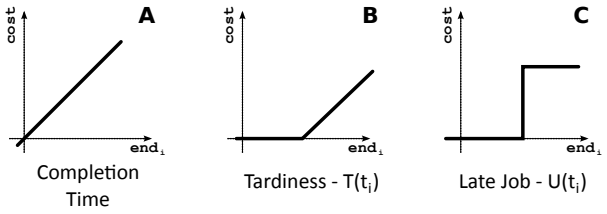


Figure 2.10: Basic blocks for scheduling objective functions

with no successors; formally, in this case:

$$F(T) = \sum_{t_i \text{ with no successor}} w_i \cdot \mathtt{E}_i$$

where w_i is the weight associated to the completion time of t_i. Alternatively, activities may be subject to deadlines dl_i, and incur some penalty in case such deadlines are exceeded. Formally, we define a activity *tardiness* as $Tr(t_i) = \max(0, dl_i - \mathtt{E}_i)$; then, one may want to minimize the *maximum tardiness* (see [Jac55]):

$$F(T) = \max_{t_i \in T} (Tr(t_i))$$

or to minimize the *total weighted tardiness*:

$$F(T) = \sum_{t_i \in T} Tr(t_i)$$

Finally, one may be interested in minimizing the number of activities exceeding the deadline (*number of late jobs*, see [Moo68]); formally, this can be modeled by introducing a function $U(t_i)$, with value 1 if $\mathtt{E}_i > dl_i$ and 0 otherwise. Then we have:

$$F(T) = \sum_{t_i \in T} U(t_i)$$

The presented objective functions are all *regular*, i.e. non decreasing in the the end time of activities. As a general rule, CP tend to perform best when the objective function is defined as the maximum over a set of expression (such in the makespan of the maximum tardiness case); in this case, any constraint on to the objective value (e.g. a global deadline) is effectively back-propagated, narrowing the time window of all activities. Conversely, CP is much less effective on the remaining objective function we have described, as they involve sum constraints and those exhibit poor propagation.

2.3.2 Filtering for the cumulative constraint

Filtering algorithms for the `cumulative` constraint (or more in general for resource constraints) represent a substantial part of the research body about CP based scheduling. In the following, the main techniques proposed in the literature and used in practice are presented (for more advanced methods one may refer to [BCP08, BP07, Lab03]); the deduction rules characterizing each filtering methods are shown as logical implications and described with the help of simple examples.

In the following, we assume activity durations to be fixed ($\mathtt{D}_i = d_i$); if this is not that case, all the presented filtering rules still apply by assuming $d_i = \min(\mathtt{D}_i)$. Note also that, when the target resource C_k is unary, all presented techniques have specialized algorithms performing the same filtering with higher efficiency.

2.3.2.1 Time-Table Filtering (TTB)

Time-table filtering for the cumulative constraint in the non-preemptive case mainly consists in maintaining bound consistency on $\mathtt{S}_i$ variable, according to Formula (2.1).

This is done by means of a global data structure, referred to as *time-table*, storing an updated best case usage profile for the current resource. More in detail, whenever at run time for an activity t_i we have $LST(t_i) \leq EET(t_i)$, then we know the activity has to execute (and use the resource) in $[LST(t_i)..EET(t_i))$ (this is called an *obligatory region*); the notation $[a..b]$ refers to the integer interval between a (included) and b (excluded). Correspondingly, the time-table is updated and the best case resource usage in such interval is increased by rq_{ik}. In the following, we denote as $RQ(\tau)$ the best case usage at time τ. As the resource consumption only changes on LST and EET, the time-table allows access to $RQ(\tau)$ in $O(\log(|T|))$ (by using binary search). Iterating over the entire data structure takes $O(|T|)$.

If during search at some time point $EST(t_i)$ there is not enough capacity left to let t_i start, we can push right the activity t_i; this is done by setting $EST(t_i)$ at the next value τ' after which the resource has enough residual capacity for at least d_i time units. Formally, let us first define $RQ(\tau, t_i)$ as the best case resource usage at time τ in the hypothesis that activity t_i is running at time τ:

$$RQ(\tau, t_i) = \begin{cases} RQ(\tau) & \text{if } LST(t_i) \leq \tau < EET(t_i) \\ RQ(\tau) + rq_{ik} & \text{otherwise} \end{cases}$$

Then, for each activity t_i:

$$\begin{aligned} \exists \tau \in [EST(t_i)..EST(t_i) + d_i) \ : \ RQ(\tau, t_i) > C_k \Rightarrow \\ \Rightarrow \mathtt{S}_i \geq \min\{\tau \mid \underbrace{RQ(\tau', t_i) \leq C_k, \forall \tau' \in [\tau..\tau + d_i)}_{(A)}\} \end{aligned} \tag{2.2}$$

Condition (A) simply requires the resource to have enough capacity left to allow t_i execute between τ and $\tau + d_i$ (excluded). For an introduction on time-table filtering see [BLPN01, Lab03]. Figure 2.11 shows an example of Time Table filtering; for each activity the boundaries of the light gray region (e.g. for t_2) represent the EST and the EET, the boundaries of the dark gray region are the LST and the LET. The duration is given by the length of the light gray region (which is the same as the dark gray one). An even darker part (e.g. tasks t_0, t_1) highlights an overlapping of the the two gray regions (i.e. an obligatory region). Since at time 4 we have $RQ(4, t_2) = 3 > 2$, activity t_2 is pushed forward.

2.3.2.2 Disjunctive Filtering (DSJ)

Disjunctive filtering is based on the simple idea that *pairs* or activities which would cause a resource over-usage cannot overlap in time. Formally, for every

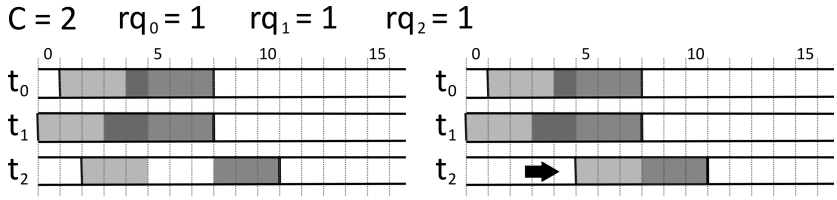


Figure 2.11: An example of Time Table filtering

pair of activities such that neither $t_i \prec t_j$ (t_i preceeds t_j) nor $t_j \prec t_i$:

$$rq_{ik} + rq_{jk} > C_k \Rightarrow \mathtt{E}_i \leq \mathtt{S}_j \vee \mathtt{E}_j \leq \mathtt{S}_i \tag{2.3}$$

in other words, if t_i and t_j would exceed the resource usage, then either $t_i \prec t_j$ or $t_j \prec t_i$. The idea can be generalized (although non-trivially) to set of activities rather than just pairs; however, this is very uncommon, as the number of resulting constraint is exponential in $|T|$.

The approach can detect some inconsistencies which are not spotted by time-table filtering; consider for example Figure 2.12: since the activities do not have any obligatory region, no deduction can be performed via timetabling. However, one can check that t_0 cannot start before t_1, while the overall requirements of the two activities exceeds C_k; hence, t_1 is forced to come before $t_0 \ldots$

Hence DSJ filtering is *not dominated* by TTB filtering. At the same time, there are situations (e.g. Figure 2.11, where no two activity overuse the resource), where all $\mathtt{S}$ variables are instantiated and no inconsistency is detected: hence TTB is not dominated by DSJ as well. For a detailed comparison among filtering algorithms for cumulative, see [BLPN01], Chapter 4.

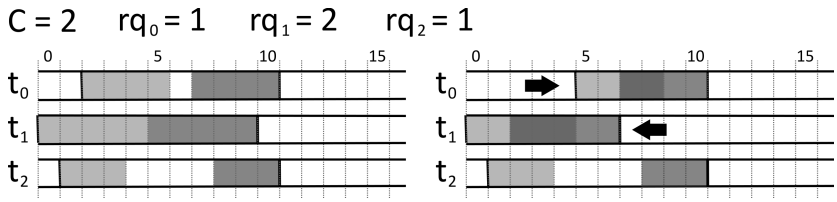


Figure 2.12: An example of Disjunctive filtering

2.3.2.3 Edge Finder (EFN)

Edge finder filtering performs some "energy based reasoning" to detect mutual relations between activities. In practice, a resource may be thought as an energy provider over time; in particular, the energy provided by r_k on the interval $[t_1..t_2)$ is defined as $C_k \cdot (t_2 - t_1)$. Similarly, each activity can be considered an energy consumer, with $E(t_i) = rq_{ik} \cdot (t_2 - t_1)$ being the energy consumed in the interval $[t_1..t_2)$.

The main idea in edge finding is that, if by adding a activity t_i to a consistent set of activities Ω the overall energy consumption over the available time window is too high, then some restriction must be applied to $\mathtt{S}_i$. As a first step toward the constraint rule, note that some of the definitions introduced for activities can be extended to a set of activities Ω. Namely:

- $EST(\Omega) = \min_{t_i \in \Omega} EST(t_i)$
- $LST(\Omega) = \max_{t_i \in \Omega} LST(t_i)$
- $EET(\Omega) = \min_{t_i \in \Omega} EET(t_i)$
- $LET(\Omega) = \max_{t_i \in \Omega} LET(t_i)$
- $E(\Omega) = \sum_{t_i \in \Omega} E(t_i)$

Classical edge finder reasoning [Nui94] requires an activity t_i and a set of activities Ω; then, if by assuming t_i to end before $LET(\Omega)$, the resulting time window does not contain enough energy, we can deduce that t_i has to end after all activities in Ω. Formally:

$$E(\Omega \cup \{t_i\}) > C_k \cdot [LET(\Omega) - EST(\Omega \cup \{t_i\})] \Rightarrow \\ \Rightarrow \forall t_j \in \Omega \ : \ \mathtt{E}_j \leq \mathtt{E}_i \tag{2.4}$$

By reasoning the other way round, one can get the dual implication. In case deduction (2.4) is performed, the end time of t_i can be adjusted to the first time point after which all t_j in Ω leave enough capacity:

$$\mathtt{E}_i \geq \min\{\tau \geq EST(\Omega) \mid \forall \tau' \geq \tau \ : \sum_{\substack{t_j \in \Omega \\ \mathtt{S}_j \leq \tau' < \mathtt{E}_j}} rq_{jk} \leq C_k - rq_i\}$$

Unfortunately, computing such value requires to solve and NP-hard problem, hence one has to resort to lower bounds; for more details see [BLPN01]. Figure 2.13 show an example of edge finder filtering; activities have no obligatory region and do not pairwise exceed the capacity, hence both TTB and DSJ cannot deduce anything. However, let the set Ω contain t_1, t_2, t_3; then the interval $[0..7)$ – i.e. $[EST(\Omega)..LET(\Omega))$ – does not provide enough energy to execute $\Omega \cup t_0$ (requiring 15 resource units); hence t_0 can be forced to end after all activities in Ω have ended. Note finding the corresponding time instant is not trivial: in this case it was done by hand.

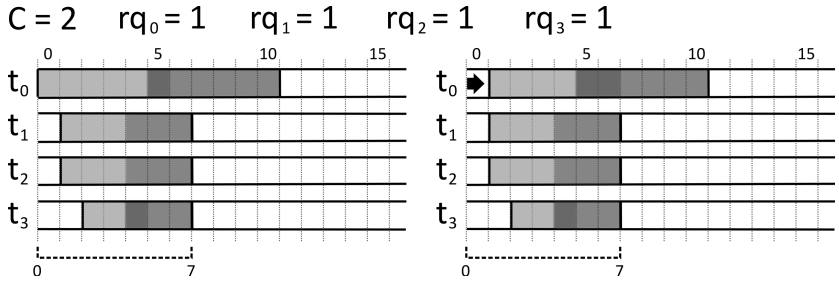


Figure 2.13: An example of Edge Finder filtering

2.3.2.4 Not-first, not-last rules (NFL)

Not-first, not-last rules (introduced in [NA96, Nui94, NA94]) share with edge finding the main idea to deduce mutual relations between a target activity t_i and a set of activities Ω. Compared to EFN, however, NFL has stronger application requirements and can result in more pruning.

In particular, if t_i can produce, together with all activities in Ω, an energy over-usage on the interval $[EST(\Omega)..LST(\Omega))$, then t_i has to start after at least one activity in Ω. Formally, let $rqer(t_i, \Omega)$ be the maximum energy consumed by t_i over the interval $[EST(\Omega)..LST(\Omega))$ in case t_i is starts as soon as possible. Then:

$$\text{and } \begin{cases} EST(\Omega) \leq EST(t_i) < EET(\Omega) \\ E(\Omega) + rqer(t_i, \Omega) > C_k \cdot [LET(\Omega) - EST(\Omega)] \end{cases} \Rightarrow \mathtt{S}_i \geq EET(\Omega) \quad (2.5)$$

with $rqer(t_i, \Omega) = rq_{ik} \cdot [\min(EET(t_i), LET(\Omega)) - EST(\Omega)]$; by reasoning the other way round one can get the dual rule. As an example, consider Figure 2.14; neither TTB nor DSJ can deduce anything. However, let $\Omega = \{t_1, t_2, t_3\}$;one can note that the time interval [0..6) does not provide enough energy to execute $\Omega \cup t_0$; since $EST(\Omega) \leq EST(t_i)$, then NFL can be applied instead of edge finding; hence t_0 is forced to start after $EET(\Omega)$. Note the resulting pruning is stronger than that performed by EFN.

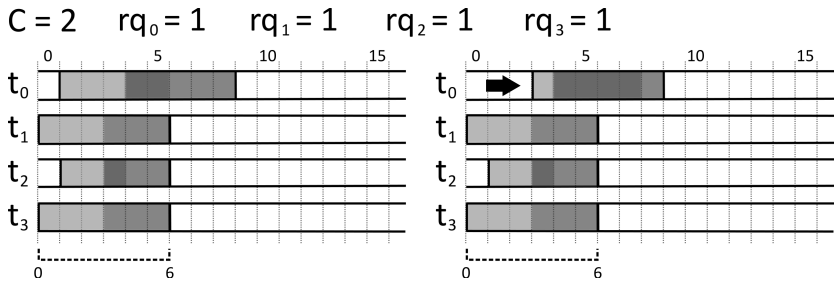


Figure 2.14: An example of Not First, Not Last filtering

2.3.2.5 Energetic Reasoning (ENR)

Additional propagation can be enabled by applying energetic reasoning on time intervals rather than to activity sets. In particular, a technique proposed in [ELT91, LEE92] considers selected relevant intervals and checks whether their best case energy usage prevents some activity t_i from being left-shifted (scheduled as soon as possible) or right-shifted (scheduled as late as possible). Formally, given a time interval $[\tau_1..\tau_2)$, we provide a notation for the following duration values:

- $\tau_2 - \tau_1$: the length of the time interval
- $d^+(t_i, \tau_1)$: the amount of time by which t_i executes after τ_1, in case the activity is left-shifted. Formally:
  $d^+(t_i, \tau_1) = \max(0, d_i - \max(0, \tau_1 - EST(t_i)))$.
- $d^-(t_i, \tau_2)$: the amount of time by which t_i executes before τ_2, in case the activity is right-shifted. Formally:
  $d^-(t_i, \tau_2) = \max(0, d_i - \max(0, LET(t_i) - \tau_2))$.

We can then introduce, over the interval $[\tau_1..\tau_2)$, the minimum energy consumed by t_i and the minimum energy consumption in general. Those are:

$$E(t_i, \tau_1, \tau_2) = rq_{ik} \cdot \min(\tau_2 - \tau_1, d^+(t_i, \tau_1), d^-(t_i, \tau_2)) \tag{2.6}$$

$$E(\tau_1, \tau_2) = \sum_{t_i \in T} E(t_i, \tau_1, \tau_2) \tag{2.7}$$

Energetic reasoning filtering then identifies time intervals $[\tau_1..\tau_2)$ and activities t_i such that, if t_i is left-shifted, an energy over-usage arises on $[t_1..t_2)$; consequently a time bound update can be performed. In detail:

$$\overbrace{E(\tau_1, \tau_2) - E(t_i, \tau_1, \tau_2) + rq_{ik} \cdot p^+(t_i, \tau_1)}^{(A)} > C_k \cdot (\tau_2 - \tau_1) \;\Rightarrow$$

$$\Rightarrow \; \mathtt{E}_i \geq \tau_2 + \left\lceil \frac{\overbrace{E(\tau_1, \tau_2) - E(t_i, \tau_1, \tau_2) + rq_{ik} \cdot p^+(t_i, \tau_1) - C_k \cdot (\tau_2 - \tau_1)}^{(B)}}{rq_{ik}} \right\rceil \tag{2.8}$$

In the above implication, expression (A) denotes the energy requirement on the time interval $[\tau_1..\tau_2)$ obtained by replacing the minimum energy usage of t_i with the left-shift usage. In case the required energy exceeds that provided by the resource on the same interval, then:

1. t_i cannot end before τ_2;
2. after τ_2, an activity t_i still requires an amount of energy; its minimum value is given by expression (B) – i.e. expression (A) minus $C_k \cdot (\tau_2 - \tau_1)$

The proof of the time bound can be find in [BLPN01]; the same text reports the precise characterization of the time intervals which need to be taken into account and show that their number is $O(|T|^2))$. An extremely important property of energetic reasoning is that ENR dominates all the filtering methods described so far; the only drawback of the approach is its non trivial computational complexity.

Figure 2.15 shows an example of energetic based filtering; no inconsistency is detected by TTB, DSJ, EFN, NFL, but one can note that the interval [2..6) does not provide enough energy to allow the left shift of t_0 ($4+2+4$ units would be required, while $2 \times 4 = 8$ are available). Hence t_0 if forced to end at $6 + (10 - 8)/1 = 8$.

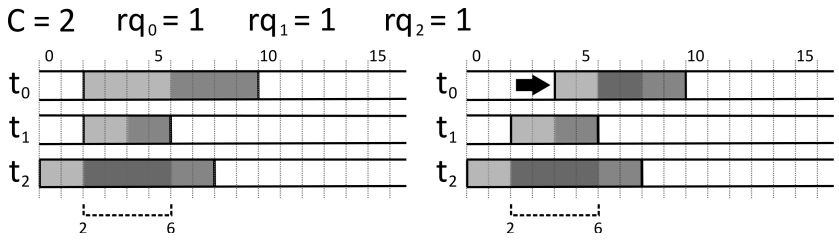


Figure 2.15: An example of Energetic Reasoning based filtering

2.3.2.6 Integrated Precedence/Cumulative Filtering

By extending the filtering algorithms to take into account more constraint one can improve the filtering capabilities. In scheduing, a particularly interesting case is to consider at the same time both resource constraints (the classical `cumulative`) and precedence relations. Work [Lab03] takes into account this combinations and provides two main algorithms for integrated precedence/cumulative filtering.

Both the proposed algorithms are based on a data structure (referred to as *precedence graph*) storing all precedence information about problem activities; in particular the graph considers precedence constraints in the initial problem formulation, as well as discovered ones. The data structure provides efficient computation of eight functions returning activity sets:

- $PSS(t_i), PSE(t_i)$: activities Possibly Starting before Start/End of t_i
- $NSS(t_i), NSE(t_i)$: activities Necessarily Starting before Start/End of t_i
- $PES(t_i), PEE(t_i)$: activities Possibly Ending before Start/End of t_i
- $NES(t_i), NEE(t_i)$: activities Necessarily Ending before Start/End of t_i

Precedence Energetic Reasoning Based on the provided precedence informations, it is possible to cast a new energetic reasoning rule; given a set of activities Ω constrained to end before t_i starts (i.e. such that $\Omega \subseteq NES(t_i)$), the resource must provide enough energy to let Ω execute before t_i can start:

$$\mathtt{S}_i \geq EET(\Omega) + \left\lceil \frac{\sum_{t_j \in \Omega} rq_{jk} \cdot d_j}{C_k} \right\rceil \tag{2.9}$$

The key advantage over standard energetic reasoning is that some activities may be constrained to start before t_i and yet this could not be deduced by reasoning on time windows alone; this is especially useful when the time windows are large. Consider for example Figure 2.16, where all the filtering rule presented so far are ineffective, due to the large time windows. However, let Ω contain t_1, t_2, t_3, all constrained to end before t_0 starts; then by combined precedence and energetic reasoning one can deduce that t_0 has to start no earlier then $\lceil 0 + 9/2 \rceil = 5$, instead of just 3 (deduced by considering precedence constraints only).

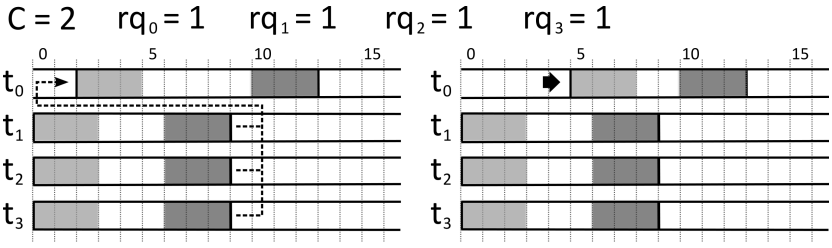


Figure 2.16: An example of Precedence Energetic filtering

Balance Constraint Additionally, the precedence graph can be used to compute upper (respectively: lower) bounds on the resource availability before the

start (respectively: after the end) of each activity t_i; this is at the base of the so called *Balance constraint* (actually a filtering rule). Here, we focus on the rule for start variables, the rule on end variables being symmetric.

In particular, an upper bound on the resource availability before the start of t_i can be computed by taking into account the usage the set Ω of activities necessarily starting before t_i ($\Omega \subseteq NSS(t_i)$) and not possibly ending before t_i (thus $PES(t_i) \cap \Omega = \emptyset$ and $NES(t_i) \cap \Omega = \emptyset$); in detail:

$$UB_{start}(r_k, t_i) = C_k - \sum_{t_j \in \Omega} rq_{jk}$$

where $\Omega = NSS(t_i) \setminus [PES(t_j) \cup NES(t_i)]$. If $UB_{start}(r_k, t_i) < 0$ then the problem is infeasible. Moreover, if the quantity $C_k - \sum_{t_j \in NSS(t_i) \setminus NES(t_i)} rq_{jk}$ is negative, then some activities in $PES(t_i)$ must end before t_i, in order to release the needed amount of resource capacity. Let this amount be $req_{start}(r_k, t_i)$; in detail:

$$req_{start}(r_k, t_i) = - \left\lceil C_k - \sum_{t_j \in NSS(t_i) \setminus NES(t_i)} rq_{jk} \right\rceil$$

Now, let $t_{j_0}, t_{j_1}, \ldots$ the sequence of activities in $PES(t_i)$, ordered by increasing EET. Let h^* be the index such that:

$$\sum_{h=0}^{h^*-1} rq_{j_h} < req_{start}(r_k, t_i) \leq \sum_{h=0}^{h^*} rq_{j_h}$$

then we know t_i cannot start before $EET(t_{h^*})$, or not enough resource capacity would be available for its execution. Hence we deduce $\mathtt{s}_i \geq EET(t_k)$. The deduction can be strengthen in case particular conditions hold: the reader can refer to [Lab03] for details.

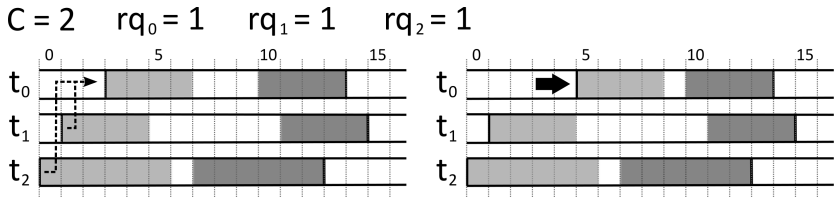


Figure 2.17: An example of Balance filtering

Figure 2.17 show an example of balance filtering; note none of the rule not taking into account precedence relations can deduce anything about t_0, due to the size of the time windows; moreover, precedence energetic reasoning is blind as well, since no activity is constrained to end before t_0. However, both t_1 and t_2 necessarily start before t_0 (due to the precedence constraints) and possibly end before t_0. As their cumulative resource consumption is 2, one can deduce by balance reasoning that t_0 has to start after the minimum between $EET(t_1), EET(t_2)$.

2.3.3 Search Strategies

Many search algorithm have been used to solve scheduling problems within a CP framework, ranging from simple tree search strategies to complex hybrid algorithms integrating OR or SAT techniques. In this section we try to outline briefly the scheme of the most widely employed search approaches.

Schedule Or Postpone Proposed in [PVG], this is a tree-search strategy focusing on the production of so called *active* schedules (see Section 2.4.3.3). At each node of the search tree an activity t_i is selected; usually, the activity with the lowest earliest start time is chosen, while latest end times are used to break ties. Then a choice point is opened; along the first branch t_i is scheduled at $EST(t_i)$, along the second branch t_i is marked as non-selectable (i.e. *postponed*) until its earliest start time is modified by propagation.

The method naturally produces schedules where no activity can be left shifted (so called active schedules). Since, if the objective function is regular, there always exist an optimal active schedule, the method is complete (unless the regularity hypothesis is violated or weird precedence constraints are used).

Precedence Constraint Posting This search method (compactly referred to as PCP) proceeds by resolving possible resource conflicts through the addition of precedence constraints. The idea is relatively old, but was very successfully applied in a CP framework by [Lab05, PCOS07]. In particular, the approach proposed in [Lab05] consists in the systematic identification and resolution of so called Minimal-Conflict-Set. A MCS for a resource r_k is a set of activities such that:

1. $\sum_{t_i \in MCS} rq_{ik} > C_k$
2. $\forall t_i \in MCS \ : \ \sum_{t_j \in MCS \setminus \{t_i\}} rq_{jk} \leq C_k$
3. $\forall t_i, t_j \in MCS$ with $i < j \ : \ \mathtt{S}_i < \mathtt{E}_j \vee \mathtt{S}_j \leq \mathtt{E}_i$ is consistent with current state of the model

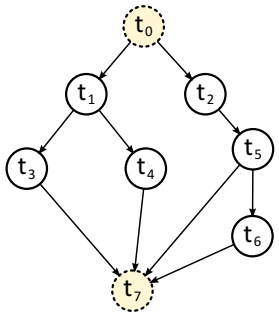
where (1) requires the set to be a conflict, (2) is the minimality condition and (3) requires activities to be possibly overlapping. A MCS can be resolved by posting a single precedence constraint between any pair of activities in the set; complete search can thus be performed by using MCS as choice points and posting on each branch a precedence constraint (also referred to as *resolver*). A deeper discussion on the Precedence Constraint Posting approach appears in Section 5.3.2.

Large Neighborhood Search (LNS) This is search process is based on interleaving relaxation and re-optimization steps; LNS was successfully applied to scheduling problems in [COS00, MVH04, GLN05]. In the most general case, the in the re-optimization step any algorithm could be adopted; in [GLN05], for example, a schedule or postpone tree search is used on the purpose. LNS is currently the default search strategy adopted ILOG CP Optimizer [LRS+08] to solve scheduling problems.

2.4 The RCPSP in Operations Research

The reference scheduling problem presented in Section 2.3.1 is an instance of the well known Resoruce Constrained Project Scheduling Problem (RCPSP). The RCPSP has been subject of deep investigation by the Operations Research community since the 70s [DH71]. In the last 40 years a large number of problem variants, benchmarks, optimal and heuristic algorithms have been provided.

The basic RCPSP formulation consists in scheduling (i.e. assigning a start time) a set of activities/tasks $T = \{t_0, t_1, \ldots\}$ connected by a set of end-to-start precedence relations $A = \{(t_i, t_j)\}$. The set of activities and precedence relations forms a graph $G = \langle T, A \rangle$, usually referred to as *project graph* in the literature. A set of *resources* $R = \{r_0, r_1, \ldots\}$ completes the problem input set; each resource has a capacity C_k and each activity t_i requires a non-negative amount rq_{ik} of resource r_k throughout its execution. The project graph usually contains a dummy source activity (similarly to Task Graphs, see Section 2.1.2 and a dummy sink activity (unlike Task Graphs). The "standard" problem objective is to minimize the makespan, and is the same as minimizing the end time of the sink activity.



d_0	0
d_1	3
d_2	2
d_3	4
d_4	1
d_5	2
d_6	1
d_7	0

C_0	3
$rq_{0,0}$	0
$rq_{1,0}$	2
$rq_{2,0}$	1
$rq_{3,0}$	2
$rq_{4,0}$	0
$rq_{5,0}$	2
$rq_{6,0}$	1
$rq_{7,0}$	0

C_0	2
$rq_{0,0}$	0
$rq_{1,0}$	1
$rq_{2,0}$	1
$rq_{3,0}$	0
$rq_{4,0}$	2
$rq_{5,0}$	1
$rq_{6,0}$	1
$rq_{7,0}$	0

Figure 2.18: An instance of a basic RCPSP; t_0 and t_7 are dummy nodes.

Constraint Programming tends to provide modular algorithm design tools (constraints, search strategies,...) which can be arbitrarily composed; conversely, the OR approach to scheduling problems focus on providing a detailed classification and characterization of the properties of each identified class. Given the variety of allocation and scheduling problem in the real world, this approach led to a pretty crowded zoo of problem types.

Covering the whole RCPSP related literature is beyond the scope of this work; for a more complete reference the reader may however refer to the surveys in [BDM+99, HB09, KP01] (and additionally to those in [HDRD98, YGO01, HL05a, KH06]). Here, the focus will be on providing a schematic overview of the considered RCPSP problem variants, the main branching rules adopted in complete approaches and the reference ILP models.

2.4.1 RCPSP Variants

RCPSP problem variants are denoted by an extension (proposed in [BDM+99]) of the popular three-field $\alpha|\beta|\gamma$ Graham's notation, introduced in [GLLK79] for machine scheduling. Despite covering the intricacies of such notation is not an objective of this work, it is useful to recall that the first parameter (α) refers to

characteristic of the resources (the so-called *resource environment*). The second parameters (β) refers to the activity characteristics and the adopted type of precedence relations; finally the third parameter (γ) describes the objective function. A similar classification scheme is be adopted here to present RCPSP subtypes.

2.4.1.1 Resource characteristics

Several types of resources are considered in the RCPSP literature, the most common being:

Renewable Resources: this is the simplest type of resource, exhibiting constant capacity over time. From another (equivalent) perspective, the resource availability is renewed at each time instant. E.g. an industrial machine or a CPU, capable of running a finite number of activities simultaneously.

Non-renewable Resources: this type of resource has a starting availability and is consumed throughout the whole scheduling horizon, until it is depleted. Project budget is a good example of non-renewable resource. In [NS03], so-called cumulative resource are also considered; those are basically non-renewable resources which can be refilled by some activities at schedule execution time. This type of resource is well known in the CP community as *reservoir*.

Partially Renewable Resources: this type of resource requires the specification of a list of sub-periods $\Pi = \{\pi_0, \pi_1, \ldots\}$ (not necessarily having the same length); the resource is renewed at the beginning of each sub-period. Note this is a generalization of both renewable and non-renewable resources.

2.4.1.2 Activity characteristics

Among the most relevant activity characteristics described in the literature, we consider:

Single Mode Activities: those are the "standard", fixed duration, non preemptive activities used as a reference in Section 2.3. Single mode activities are the most often encountered in practical problem models and solution methods.

Multi Mode Activities: this type of activity (introduced in [Elm77]) may execute in one of a set of *modes*; each mode can define a different duration and different resource usage. Multiple modes may arise (for example) when the project responsible can afford spending more money on an activity to make it finish earlier (time-cost tradeoff problems – see [DDRH00] for the single resource case, and [RDRK09] for multiple renewable resources); for this reason multi-mode activities are traditionally associated with non-renewable resources.

Multi mode activities can also be used to introduce a resource allocation step in a RCPSP framework (see [VUPB07]); this is done by defining, for each activity, modes using different renewable resources, while there is no need to take into account a non-renewable resource. Somehow surprisingly, this problem setting is extremely uncommon in the OR literature.

Preemptive Activities: by preemption, we mean that an activity may be interrupted at some point of time and recovered later on. An activity t_i may be preempted to give priority to a second, more urgent, task using the same resource; alternatively, preemption can be a side-effect of having "holes" in resource availability (e.g. due to vacation time, week-ends, etc.). Preemption is never considered with multiple modes.

From the point of view of this work, non-preemptive single and multi-mode activities with renewable resources are the most interesting RCPSP setting.

2.4.1.3 Precedence Relation Types

The β parameter in the three field classification refers to the precedence relation features; besides traditional end-to-start arcs, the most relevant classes of precedence constraints include:

Generalized Precedence Relations: in this case end-to-start relations are generalized by introducing start-to-start, end-to-end and start-to-end precedence constraints. Note that, as one may intuitively realize, all this precedence relation types can be converted to some extent one into another, as described in [BMR88]. Note however that the conversion cannot be applied in multi-mode scheduling.

Minimal and Maximal Time Lags: minimal and maximal time lags may label the precedence constraints so that, by referring as $\mathtt{S}_i$ and $\mathtt{E}_i$ to the start and end time of activity t_i, the produced schedule must satisfy $\delta_{min} \leq \mathtt{E}_j - \mathtt{S}_i \leq \delta_{max}$; where δ_{min}, δ_{max} respectively are the the minimum and maximum time lag labeling arc (t_i, t_j).

Minimal time lags are a useful modeling tool when resource requires setup time between specific sequences of activities; for example, if activities t_i and t_j have to be processed at different temperatures on an industrial oven, some setup time will be required between t_i and t_j to allow the temperature to change. From a computational point of view, basically every approach dealing with common precedence relations can be extended to take into account minimal time lags. For some recent works taking into account minimal time lags, see [HB09].

Maximal time lags, however, are another pair of shoes, as they introduce relative deadlines on the activity execution; as a consequence, finding a feasible schedule becomes NP-complete; conversely, in the basic RCPSP formulation a trivially feasible schedule can always be found by serializing all activities. This modification hinders most of the algorithms devised for the RCPSP without maximal time lags.

An interesting point is to note that a maximal time lag on an arc (t_i, t_j) can be converted into a *negative* minimal time lag on the complementary arc (t_j, t_i) (see [BDM$^+$99]). The resulting network contains a feasible schedule if and only if no cycle with positive length exists (see [BMR88]); this further stresses the analogy between project graph with time lags and Simple Temporal Networks in Artificial Intelligence (see [DMP91]), for which analogous results have been produced. Finally, note that fixed release dates and deadline on each activity t_i can be set by posting precedence constraints with time lags between the dummy start node and t_i.

Logical Dependencies: so far we have implicitly assumed that an activity t_j destination of more arcs (t_{i_k}, t_j) must execute after *all* the predecessor nodes. From a logical standpoint, let $S(t_i)$ the predicate "activity t_i has started" and let $E(t_i)$ be "activity t_i has ended"; the set of precedence constraint enforces the following logical relation:

$$S(t_j) \Rightarrow \bigwedge E(t_{i_k})$$

hence, the set of arcs can be seen as a single AND hyper-arc; by generalizing the idea, one may have OR hyper-arcs as well (often referred to as *disjunctive arcs* in the literature), forcing a t_j to execute after at least one predecessor t_{i_k}. This is especially handful in case a precedence based branching schema is adopted, as described in the forthcoming Section 2.4.3.1. Note however that due to disjunctive arcs the feasible region is no longer convex, but a union of convex polyhedra (see [Sch98]); this prevent the application of some techniques: most notably, Bound Consistency on precedence relations is no longer equivalent Generalized Arc Consistency.

2.4.1.4 Objective Functions Types

Some of the objective functions considered in Resource Constrained Project Scheduling have already been presented in Section 2.3.1.1; in particular, all the objective presented so far (completion time, tardiness, number of late jobs...) can be classified as *time based objectives* and are the most frequently occurring in practical problems. Nevertheless, many other type of functions appear in the RCPSP literature; some of them are recalled in the followings; for an extended list see [HB09].

Resource Based Objectives: those include all function measuring a resource related cost. In particular, there may be a cost to pay to provide resource capacity; this leads to objective in the form $\sum_k cost(C_k)$, where *cost* is any non-decreasing function of the resource capacity C_k. The resulting RCPSP variant goes under the name of *resource investment problem*. Alternatively, the price to pay could be associated to capacity variations; this leads to objectives such as the minimization of the maximum change, or the sum of changes; the resulting RCPSP settings takes the name or *resource leveling problem*. A comprehensive list of resource related objective functions can be found in [NSZ95]. Note that design space exploration (see Section 2.1.1) can be though as very peculiar problem with resource dependent objective.

Net-Present Value Based Objectives: the last class of objective functions mentioned in this work are based on the net present value. Basically, in many project scheduling problems a cost is associated to the execution of each activity (or some other project subpart); similarly, a reward can be associated to the end of the activities. To limit the project cost, one may be interested in minimizing the maximum value on the project graph at execution time. Many variants of such objective exist in the literature; for a list see [HB09].

2.4.2 Representations, models and bounds

This section covers some of the adopted representations, models and bounds for the RCPSP. Basically, representations tend to suggest models and models

in turn serve as a base for bound computation; hence it was quite natural to group all those aspects in a single section.

2.4.2.1 RCPSP representations

Typically, RCPSPs are described by means of graph-based representations. Different mappings between the problem concepts and the graph objects lead to different formulations; the most common choices respectively map activities to nodes and activities to arcs, and are referred to through the acronyms AON (Activity On Node) and AOA (Activity On Arc).

As an alternative, one may rely on a specific deterministic scheduling technique to provide a more compact representation; for example, it is quite common in RCPSP heuristics to encode a schedule as an *activity list*; this can be turned in an actual start time assignment (in linear time) by a priority rule scheduling algorithm. In the following, all the mentioned representations are briefly discussed; the content of this section is mainly based on [KP01].

Activity on Node representation: the AON representation is the most frequently adopted; each activity/task corresponds to a graph node; dummy source nodes and sink nodes are usually added to model project start/end. AON is quite a natural mapping choice, leads to compact representations and lets one easy model single and multi mode activities (by annotation of nodes), time lags (by annotation of arcs), classical time and resource based objectives. However, AON has some difficulties in modeling generalized precedence constraints (such as start-to-start) and net value based objectives; in fact, annotating nodes with both costs ad rewards fails to model the fact that necessary expenses occur *before* the reward is achieved.

Activity on Arc representation: in this representation type, graph nodes represent events (such as an activity start or end) and arcs represent activities. Dummy nodes are usually added to model the project start/end and dummy activities (arcs) are used to model precedence relations. Compared to the previous approach, AOA representations support generalized precedence constraints in a straightforward fashion.

AOA representations actually come in two flavors, depending on whether node events are related to (A) single activities or to (B) multiple activities. Case (B) allows a more compact representation (usually as compact as AON models) but generally fails to model correctly net value based objectives; the reason is that, if costs or rewards related to multiple activities are mapped on the same event, it is impossible to trace back exactly the activity they refer to. One may easily circumvent the issue by associating nodes to single activity events (namely "start" and "end"), with the drawback of doubling the number of nodes; this may become a problem in large projects.

Activity List Representation: here the main idea is to represent a *schedule* (not a project) as the input for a reference algorithm; priority rule based scheduling (see forthcoming Section 2.4.3.3) is often used on this purpose, as it requires the specification of a single priority value for each activity (at least in the basic version). This allows one to represent a whole schedule as a list of numbers, or a simple list of activities (sorted by their priority). The compactness and the simplicity of this representation make it an appealing choice for heuristic

approaches such as genetic algorithms or local search, by making easier the definition of the required operators.

Gantt Charts: Gantt Charts are mentioned here as they are a well known tool to represent project schedules, as well as the underlying representation for many commonly used RCPSP models. A Gantt Chart is table-like diagram where each row refers to an activity and each column to a time point; a cell i, j is either filled if activity i is running at time j, or blank otherwise.

2.4.2.2 RCPSP Reference Models

An overwhelming number of different solution approaches can be found in the RCPSP related literature; some of them are based on clear declarative models, some are not. Despite covering the whole range of adopted formulation would be impossible, many of the used models actually origin from a very limited number of basic schemes; the main ones according to the author are presented in the followings.

Time Point Based Model Time point based models have roots in graphical representations of the project network. The main idea is to introduce for each node in the graph (either activity or event) a real or integer variable telling the time instant where the event occurs; for example, one may introduce a start variable $\mathtt{S}_i$ for each activity t_i.

Time-valued variables allow for straightforward modeling of precedence relations; for example a simple precedence relation (t_i, t_j) can be represented as $\mathtt{S}_i + d_i \leq \mathtt{S}_j$, in the hypothesis that all activities have fixed durations d_i. In order to trace resource usage in time we can assume to have access to a $T(\tau)$ function denoting the set of activities executing at time τ:

$$T(\tau) = \{t_i \in T \mid \mathtt{S}_i \leq \tau < \mathtt{S}_i + d_i\}$$

Then a time point based model for a basic RCPSP problem (single mode, time-based objective, renewable resources) can be formulated as follows:

$$\begin{aligned}
\text{(M0)} \qquad & \text{min: } F(\mathtt{S}) && && (2.10)\\
& \text{subject to: } \mathtt{S}_j - \mathtt{S}_i \geq d_i && \forall (t_i, t_j) \in A && (2.11)\\
& \sum_{t_j \in T(\mathtt{S}_i)} rq_{jk} \leq C_k && \forall r_k \in R, \forall t_i \in T && (2.12)\\
& \text{with: } \mathtt{S}_i \in \mathbb{N}^+ && \forall t_i \in T && (2.13)
\end{aligned}$$

where T is the set of project task/activities t_i, A is the set of precedence constraints, R is the set of resources r_k with capacity C_k; rq_{ik} is the amount of resource r_k required by activity t_i for its execution. Set $\mathbb{N}^+$ is assumed to contain element 0. Despite its simplicity, model (M0) serves as a basis for many practical models. From a classical OR perspective, M0 presents two major problems:

1. the $T(\tau)$ function is strongly non linear
2. the argument of $T(\tau)$ in constraints (2.12) is a variable (namely, S_i)

Generally, we can say that the key difficulty with time point based models is to deal with resource constraints. This provides motivation for the class of time indexed models.

Bounds: the LP relaxation of time point based models allows trivial access to the longest path on the project graph (a.k.a. *critical path*), providing a simple but effective lower bound on the makespan. More advanced bounding techniques try to partially take into account resources in the bound computation; an early example can be found in [SDK78], for more recent makespan bounds based on the critical path, see [BJK93, BC09a].

Time Indexed Model Gantt charts are the implicit foundation of time indexed models; here the main idea is to have binary variables assessing the presence of each activity t_i at each time index (see [PWW69] for an early reference). For example (see [BC09b]), let us assumes to have for each activity a variable $\mathtt{S}_{i\tau}$, (telling whether $t_i$has started at time τ) and a variable $\mathtt{E}_i$, telling whether t_i has ended at time τ. Some basic constraints can be used to enforce meaningful assignment of the S, E variables:

$$\mathtt{S}_{i,\tau-1} \leq \mathtt{S}_{i,\tau} \qquad \forall t_i \in T, \forall \tau = 1, \ldots eoh \tag{2.14}$$

$$\mathtt{E}_{i,\tau-1} \leq \mathtt{E}_{i,\tau} \qquad \forall t_i \in T, \forall \tau = 1, \ldots eoh \tag{2.15}$$

$$\mathtt{E}_{i,\tau} \leq \mathtt{S}_{i,\tau} \qquad \forall t_i \in T, \forall \tau = 0, \ldots eoh \tag{2.16}$$

intuitively, once an activity starts, the corresponding S variable cannot assume the value 0 (constraints (2.14)); a similar restriction holds of course for end variables (constraints (2.15)). Basically, constraints (2.14) and (2.15) force "1" values to be contiguous. Finally, an activity cannot end if it has not yet started (constraints 2.16). Activity durations can be modeled as follows:

$$\sum_{\tau=0}^{eoh} \mathtt{S}_{i\tau} - \sum_{\tau=0}^{eoh} \mathtt{E}_{i\tau} = d_i \qquad \forall t_i \in T \tag{2.17}$$

where *eoh* (End Of Horizon) is an upper bound on the length of a feasible schedule. Constraints 2.17 require the number of S variables set to 1 to exceed the number of E variables set to 1 by at least d_i units. Constraints to enforce resource capacity limits are immediately formulated as:

$$\sum_{t_i \in T} rq_{ik} \left(\mathtt{S}_{i\tau} - \mathtt{E}_{i\tau}\right) \leq C_k \qquad \forall r_k \in R \tag{2.18}$$

in practice, at every time index τ the sum of the requirements of the currently running activities must not exceed the resource capacity. Activities currently in execution are identified as they have started and have not yet ended. A simple precedence relation (t_i, t_j) with no minimum time lag can be modeled as:

$$\mathtt{S}_{j\tau} \leq \mathtt{E}_{i\tau} \qquad \forall \tau = 0, \ldots eoh \tag{2.19}$$

that is, t_j can start only once t_i has ended. Minimum and maximum time lags can be introduced by quantitatively reasoning on S and E variables, analogously to duration constraints. A simple time indexed model may therefore look as

follows:

$$
\begin{array}{lrll}
\text{(M1)} & \text{min:} & F(\mathtt{S},\mathtt{E}) & \\
& \text{subject to:} & \mathtt{S}_{i,\tau-1} \leq \mathtt{E}_{i,\tau} & \forall t_i \in T, \forall \tau = 1, \ldots eoh \\
& & \mathtt{E}_{i,\tau-1} \leq \mathtt{E}_{i,\tau} & \forall t_i \in T, \forall \tau = 1, \ldots eoh \\
& & \mathtt{E}_{i,\tau-1} \leq \mathtt{S}_{i,\tau} & \forall t_i \in T, \forall \tau = 0, \ldots eoh \\
& & \sum_{\tau=0}^{eoh} \mathtt{S}_{i\tau} - \sum_{\tau=0}^{eoh} \mathtt{E}_{i\tau} = d_i & \forall t_i \in T \\
& & \sum_{t_i \in T} rq_{ik} \left(\mathtt{S}_{i\tau} - \mathtt{E}_{i\tau}\right) \leq C_k & \forall r_k \in R \\
& \text{with:} & \mathtt{S}_{i\tau}, \mathtt{E}_{i\tau} \in \{0,1\} & \forall t_i \in T, \forall \tau = 0, \ldots eoh
\end{array}
$$

where we assume the objective function is time based. Once again, this relatively simple model outlines the main ideas behind many practical approaches. From a computational standpoint, (M1) presents two major issues:

1. the number of variables depends on the time horizon;
2. the LP relaxation can be very weak

Bounds: makespan bounds based on time indexed models are usually obtained through the applications of relaxation techniques. In particualar, for the presented model, the LP relaxation provides quite a weak bound; as an alternative, a lagrangian relaxation based bound is proposed in [BC09b]. Better results are obtained in [MMRB98] by allowing preemption, partially relaxing precedence constraints and, most notably, by scheduling *sets* of activities not exceeding the resource capacities instead of single activities; those are referred to as *feasible sets*. The resulting problem has an exponential number of variables and can be solved via column-generation. The technique is improved in [BK00] with the addition of constraint programming based pre-processing and by taking into account time windows. Both in [MMRB98] and [BK00] pure time indexed variables are replaced by variables telling the number of time units each feasible set executes.

2.4.3 Algorithmic Techniques

Covering the whole set of algorithms used to tackle the Resource Constrained Project Scheduling Problem is far out of the scope of this chapter. Nevertheless, it is worthwhile to look through some specific techniques often used as low level components of more complex algorithms. In particular, in the following we will cover branching schemes, dominance rules and priority rule based scheduling.

2.4.3.1 Branching Schemes

Many Branch and Bound have been proposed for the RCPSP, adopting each different branching strategies. The following is an attempt to provide a fairly comprehensive overview of the main ideas. Most of the material for this section comes from [BDM+99].

Precedence Tree Branching: this branching strategy, first defined in [PSTW89] consists in scheduling at each step of the search tree an activity whose predecessors have all been *scheduled*. Each node of the search tree is associated with a set of activities eligible for scheduling; on backtrack, different activities are chosen, so that each path from the root of the search tree to the leaves corresponds to a possible linearization of the partial order induced on T by the precedence graph.

Delay Alternatives: in this branching method, introduced in [CAVT87, DH92], each node of the search ree is associated to a time instant τ. A clear distinction is then made between completed activities at time τ (say $CT(\tau)$) and activities in process (say $PT(\tau)$); eligible activities for scheduling are those whose predecessors have all *completed* execution. Then an attempt is made to schedule all eligible activities, so that they are all added to the set of activities in process. Of course this may cause a conflict; in such a case the method branches by *withdrawing* from execution so called *delay alternatives*. Those are:

1. activities in process (that is, in $PT(\tau)$),
2. such that, if they are removed, no resource conflict occurs

This method differs from the precedence tree based one in two regards: (1) one branches on *sets* of activities and (2) scheduled activities may be withdraw from execution. The applicability of the approach requires some hypothesis (e.g. constant resource capacity).

Extension Alternatives: this method, proposed in [SDK78], closely resembles the previous one as each search node corresponds to a time instant τ, for which sets $CT(\tau)$ and $PT(\tau)$ are identified. However, the idea here is to branch on sets of activities which can be started without violating resource constraints (so called *extension alternatives*).

Scheduling Schemes: an interesting branching scheme proposed in [BKST98] is based on the idea to branch on *pairs* of activities t_i, t_j, by forcing them to follow a specific scheduling *scheme*. This can be either parallel execution (in notation $t_i \| t_j$), or one of the two possible sequential executions ($t_i \rightarrow t_j$ or $t_j \rightarrow t_i$).

Minimal Forbidden Sets: the idea of Minimal Forbidden Sets was introduced in [IR83b, IR83a]; those are minimal size sets of activities causing a resource overusage in case they overlap. Minimal Forbidden Sets are known in Constraint Programming as Minimal Critical Sets [Lab05], or Minimal Conflict Sets (see Section 2.3.3). Once an MCS is identified a basic branching strategy consists in enumerating all possible resolvers (i.e. pairwise precedence relations). A more advanced strategy, proposed in [Sch98] consists in posting disjunctive precedence constraints, i.e. an activity t_i in the conflict set has to come after *any* another activity in the set; this greatly reduces the size of the search tree, at the price of altering the problem structure (see Section 2.4.1.3) and making it less amenable for propagation.

2.4.3.2 Dominance Rules

Scheduling problems quite often happen to have more than one optimal solution; it is indeed quite common to have large numbers of equivalent solutions in terms of optimality. Dominance rules exploit this property to narrow the search space

by forcing the selection of a specific schedule. Devising such a rule requires to prove that among the set of optimal solutions, there exist a schedule satisfying a specific property; such property is usually in the form of an implication (e.g. $A(\mathtt{S}) \Rightarrow B(\mathtt{S})$, where $\mathtt{S}$ is the set of start time variables) and can then be posted as a new problem constraint. For example, the three schedules in Figure 2.19 all have the same makespan; one can see that restricting the attention to the rightmost one (where thereis no "hole" between t_2 and t_3) would be sufficient.

Many dominance rules have been proposed in the scheduling literature; for some examples see [BDM+99, HD98, SD98, DH92, BP00]. Several of such techniques can in fact be assimilated to constraint propagation; in the following, we summarize some of the rules which are not subsumed by existing filtering algorithms.

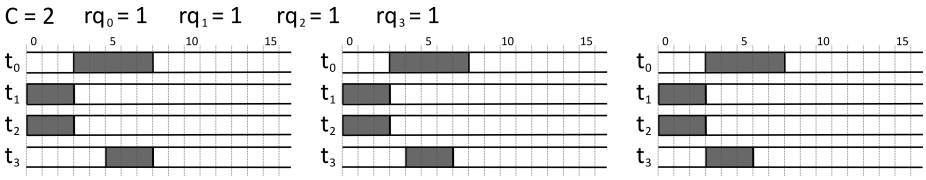


Figure 2.19: Three schedules having the same makespan

Left Shift Rule: If an activity that has been started at the current level of the branch and bound tree can be left-shifted, then the current partial schedule need not to be completed; note this rule effectively solves the issue highlighted in Figure 2.19. This rules is often implicitly embedded in many CP search algorithms for scheduling problems.

Order Swap Rule: Consider a scheduled activity the finish start of which is less than or equal to any start time that may be assigned when completing the current partial schedule. If an order swap on this activity together with any of those activities that finish at its start time can be performed, then the current partial schedule needs not to be completed.

Cutset Rule: This rule is introduced in [SD98] and is based on the definition of a cutset ($CS(\sigma)$) as the set of activities scheduled in a given partial schedule σ. The cutset rule consists in memorizing evaluated partial schedules during search; if at any point of time a partial schedule σ is built such that:

1. the cutset of σ is the same as that of a stored partial schedule $\bar{\sigma}$
2. the minimal time τ at which an activity can be scheduled in σ is grater or equal than the maximal finish time of activities in $\bar{\sigma}$
3. all the leftover capacities in σ are lower or equal than those in $\bar{\sigma}$

then the current partial schedule needs not to be completed. One can note how the cutset rule is actually a form of no-good learning (see Section 2.2.2.1).

Incompatible Set Decomposition Rule: This rule was presented by [BP00] in the context of a CP scheduling algorithm. The rule generalizes the single incompatibility rule introduced in [DH92] and requires the definition of a directed graph $\langle T, A \rangle$ as follows:

- T is the set of tasks/activities

- both arcs (t_i, t_j) and (t_j, t_i) are in A if t_i is compatible with t_j (i.e. they can execute in parallel according to both precedence and resource constraints).
- arc (t_i, t_j) is in A is t_i comes before t_j

then all connected components T_i in $\langle T, A \rangle$ are identified; it can be shown that they form a partially ordered set and hence can be "sequenced" in a number of ways, thus obtaining total orders. Let a valid total order be $T_0, T_1, \ldots$; then the rule states that all activities in T_i must end before any activity in T_{i+1} starts. If applicable, the rule defines a very strong problem decomposition.

2.4.3.3 Priority Rule Based Scheduling

Priority rule based scheduling is the oldest class of RCPSP heuristics to appear; nevertheless, it still retains a lot of importance. This is due to three main reasons:

1. the method is intuitive, easy to use, to tune and to implement; this makes it a very popular algorithm, especially when solving a scheduling problem is not the core problem tackled.
2. the method is computationally fast, making it an ideal choice for integration within complex AI/OR approaches or metaheuristics.
3. the method is effective (in particular in its multi-pass version); this is especially true when the project graph contains many precedence relations.

A priority rule based scheduling approach is made up of two components, a *schedule generation scheme* and a *priority rule*. While a huge number of priority rules (either generic or problem specific) has been proposed in the literature, basically two main scheduling schemes can be distinguished: the so called *serial* and the *parallel* method. Both generate a schedule by extending a feasible partial schedule (i.e. a schedule where only some activities are given a start time) in a stage-wise fashion. At each stage the priority rule is used to choose an activity to perform the extension.

In the following, we briefly present the two possible schedule generation schemes; for a more detailed description of the priority based approach see [Kol96]. The same paper proves the the serial method produces so called active schedules (see Section 2.3.3), while the parallel methods results in non-delay schedules [SKD95]; note that if a regular objective function is minimized (see Section 2.3.1.1) the set of active schedules is guaranteed to contain an active schedule, but not necessarily a non-delay schedule. For more general survey of scheduling heuristics and metaheuristics the reader may refer to [KH06].

The Serial Method: The serial method was proposed in [Kel63]; each stage here is characterized by a set of *scheduled activities* and a *decision set*, containing activities whose predecessors have all been scheduled. The priority rule is used to select an activity from the decision step; this is scheduled at earliest possible precedence *and* resource feasible time. Then, the process is reiterated. The serial method is known in many contexts as list scheduling (see Section 2.1.3).

The Parallel Method: The schedule generation scheme usually referred to as "parallel" is introduced in [BB82]. In such method, each generation stage is associated to a time instant τ, at which the set of completed tasks $CT(\tau)$ and the

set of processing tasks $PT(\tau)$ can be identified. The decision set contains activities which are both precedence *and* resource feasible at time τ; among those activities a single one is selected by means of the priority rule. The main difference compared to the previous approach is that resources and limited capacities are taken into account to build the decision set, rather then in determining the start time to be assigned.

2.5 Hybrid Methods for A&S Problems

Mixed resource allocation and scheduling problems, such as those treated in this work, are well known to be tough to deal with. Pure OR and CP approaches can claim individual successes on specific cases, but neither of the approaches clearly wins on more general problems. This is a consequence of the intrinsically hybrid nature of resource allocation and scheduling, resulting from the combination of an assignment and a scheduling subproblems. While MILP models stand out as natural candidates for the assignment part, CP is much more effective in dealing with the large domain variables and complex non-linear constraints often found in scheduling.

This raises interest in hybrid algorithmic techniques, to take advantage of the mutual strengths of heterogeneous methods and compensate for their weaknesses. Constraint Programming in particular offers an ideal framework for the development of hybrid algorithms, for a threefold reason:

1. typical CP engines support easy development of *many types of search algorithms*; in fact, the only usual requirement for the search method is to proceed in stepwise fashion by posting new constraints to the model. The actual implementation of each step is left to the user, as well as the decision whether to take advantage or not of the available backtracking support.
2. the *neat separation between search and model* allows one to exploit filtering and propagation, whatever the search method is.
3. constraints have a declarative definition and a procedural implementation; this is made possible by allowing them to interact in modular fashion through the variable domains (the so-called *domain store*); hence *each constraint can perform filtering by using any type of algorithm.*

Integration within a CP Framework Global constraints based on matching theory [Rég94] and network flow algorithms [Rég96] have made their appearance since the early establishment of CP. More recently, *optimization constraints* [FLM02b] were introduced; those make wide use of OR techniques (algorithms and models) to enforce bound consistency on a cost variable and/or perform filtering based on reduced costs [FLM99]; reduced cost can be used to guide search as well [MH04]. The effectiveness of the approach is showcased in [FLM02a].

Moreover, *lazy clause generation* is worth mentioning in this context; this consists in embedding a SAT solver (as a constraint) within a CP framework [FS09]; CP filtering algorithms, beside pruning variable domains, act as clause generators for the SAT model. The ability of the SAT solver to identify and record no-goods is used to perform filtering, backjumping and to guide the CP search. The approach proved very effective on scheduling problems [SFSW09].

This work is entirely dedicated to practical applications of hybrid allocation and scheduling methods. In particular, in many of the presented case studies the integration between heterogeneous approaches is realized via a technique known as Logic Based Benders' Decomposition (LBD), described in detail in the following section.

2.5.1 Logic Based Benders' Decomposition

Logic Based Benders' Decomposition (formalized by Hooker in [HO03]), is a generalization of Benders' Decomposition (BD, see [Ben62]). This is a classic OR technique, based on the idea of learning from one's mistakes; namely, the method solves a problem by enumerating values of a subset of the variables (so called *primary variables*). For each set of values enumerated, it solves the subproblem that results from fixing the primary variables to these values. The solution of the subproblem is used to generate a *Benders' cut* that the primary variables must satisfy in all subsequent solutions enumerated. The classical Benders' cut is a linear inequality based on Lagrange multipliers obtained from a solution of the subproblem dual. The next set of values for the primary variables is obtained by solving the master problem, which contains all the Benders cuts so far generated. The process continues until the master problem and subproblem converge in value.

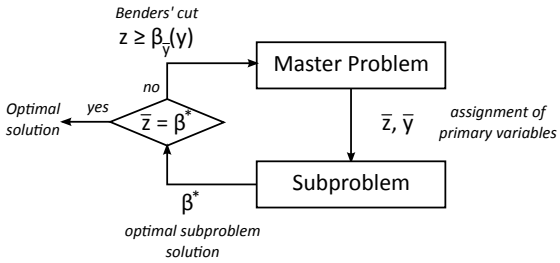


Figure 2.20: Structure of the Logic based Benders' Decomposition Approach

The generalization of the subproblem dual, used for cut generation, is the key step to enable the application the classical Benders' scheme to a broader class of problems and to non-LP techniques. This is done in [HO03] by observing that the solution of the dual problem in BD is just a mean to *infer* a bound from the constraints. This leads to the definition of a so-called inference dual, not requiring the linearity of the subproblem.

The Inference Dual Formally, let us consider a general optimization problem in the form:

$$(P0) \qquad \text{min:} \quad f(x) \tag{2.20}$$

$$\text{subject to:} \quad x \in S \tag{2.21}$$

$$\text{with:} \quad x \in D \tag{2.22}$$

where $f(x)$ is a generic function of the set x of variables, while D is their domain. The constraints are given by simply requiring x to belong to a set $S \subseteq D$. Then,

stating an inference dual requires to define an implication rule on the domain D. Let this be $P \xrightarrow{D} Q$, then the *inference dual* is:

$$(D0) \qquad \text{max:} \qquad \beta \tag{2.23}$$

$$\text{subject to:} \quad x \in S \xrightarrow{D} f(x) \geq \beta \tag{2.24}$$

The dual seeks the largest β for which $f(x) \geq \beta$ can be inferred from the constraint set. In other words, the dual problem is to find the strongest possible bound on the objective function value. The optimal value of the inference dual is the same as the original problem (i.e. *strong duality* holds). Classical LP duality if a specific case of inference duality.

Structure of the LBD approach Benders decomposition views elements of the feasible set as pairs (x, y) of objects that belong respectively to domains D_x, D_y. Problem (P0) can then be stated:

$$(P1) \qquad \text{min:} \qquad f(x, y) \tag{2.25}$$

$$\text{subject to:} \quad (x, y) \in S \tag{2.26}$$

$$\text{with:} \quad x \in D_x, y \in D_y \tag{2.27}$$

A general LBD approach (see Figure 2.20) begins by fixing y to some value $\bar{y} \in D_y$, for example by means of an initial heuristic (the master problem, reported in the figure, will be introduced later). This immediately leads to the following *subproblem*:

$$(SP) \qquad \text{min:} \qquad f(x, \bar{y}) \tag{2.28}$$

$$\text{subject to:} \quad (x, \bar{y}) \in S \tag{2.29}$$

$$\text{with:} \quad x \in D_x \tag{2.30}$$

Rather than solving the subproblem directly, one can focus on the corresponding inference dual (known to have the same optimal value).

$$(DP) \qquad \text{max:} \qquad \beta \tag{2.31}$$

$$\text{subject to:} \quad (x, \bar{y}) \in S \xrightarrow{D} f(x, \bar{y}) \geq \beta \tag{2.32}$$

The dual problem is to find the best possible lower bound β^* on the optimal cost that can be inferred from the constraints, assuming y is fixed to $\bar{y}$. Basically, due to strong duality, the optimal value of the (primal) subproblem is the same as the tightest bound on $f(x, \bar{y})$ which can be inferred from the constraint. Once such a value β^* is available, the key step of the process is to identify a bounding function $\beta_{\bar{y}}(y)$ such that:

$$\beta_{\bar{y}}(y) = \begin{cases} \beta^* \text{ if } y = \bar{y} \\ \beta'(y) \leq f(x, y) \text{ otherwise} \end{cases}$$

the subscript $\bar{y}$ of the bounding function denotes the y values used for its construction; $\beta_{\bar{y}}(y)$ equals the tightest lower bound when $y = \bar{y}$, otherwise it provides a valid (likely more loose) bound. If the subproblem is infeasible, the dual

is unbounded and $\beta^* = \infty$, which prevents the method from proposing again the same $\bar{y}$ value. By using the bounding functions built iteration by iteration, a *master problem* can be defined:

$$
\begin{array}{lllll}
(MP) & \text{min:} & z & & (2.33) \\
 & \text{subject to:} & z \geq \beta_{\bar{y}_k} & \forall k = 0, 1, \ldots & (2.34) \\
 & \text{with:} & y \in D_y & & (2.35)
\end{array}
$$

where $\bar{y}_0, \bar{y}_1 \ldots$ are the trial values hitherto obtained. Let the solution of the master problem be $(\bar{z}, \bar{y})$, then the $\bar{y}$ value can be used to prime the following subproblem iteration. The method ends when the master problem and subproblem value converge, that is $\bar{z} = \beta^*$.

Discussion and References The outlined method is complete, i.e. guaranteed to provide the optimal solution. Provided D_y is finite, convergence is necessarily achieved in a finite number of steps. The LBD method has two major advantages inherited from BD, namely:

1. breaking the dependence between the master and subproblem variables (respectively, y and x) can be done by taking advantage of the structure of the problem. This can greatly simplify the problem model, or break global constraints and enable the decoupling of the subproblem.
2. Properly devised cuts can clear a wide portion of the search space, thus boosting convergence.

Furthermore, a peculiar advantage of Logic Based Benders' Decomposition is the possibility to use heterogeneous techniques to solve the master and the subproblem. This broadens the applicability of the method (as linear models are no longer a requirement) and may enable impressive speed-ups. In the literature, LBD has been applied to 0-1 programming and SAT [HO03], circuit verification [HY94], automated vehicle dispatching [DLRV03], steel production planning [HG01], real time multiprocessor scheduling [CHD$^+$04], batch scheduling in a chemical plant [MG04].

Most notably, the Logic based Benders Decomposition approach has been used so solve allocation and scheduling problems [JG01, HG02, Hoo05b, Hoo05a, Hoo07]. Here the whole problem is usually decoupled into an allocation master problem and a scheduling subproblem; LBD allows the use of MILP for the master and CP for the subproblem, so that each technique is effectively employed for what it is best for.

LBD has of course a fundamental drawback; in fact, by decomposing the problem in two stages one incurs the risk of loosing valuable information (e.g. if master and problem variables are connected by tight constraints); hence, extra care should be put in breaking the problem in a proper manner. As a partial solution to this issue, a *subproblem relaxation* may be embedded in the master problem; in the context of the abstract framework described above, such a relaxation may be inserted as bounding function $g(y)$ on the objective variable

z in the master problem; therefore, (MP) becomes:

$$
\begin{aligned}
(MP) \qquad & \text{min:} && z && && (2.36)\\
& \text{subject to:} && z \geq \beta_{\bar{y}_k} && \forall k = 0, 1, \ldots && (2.37)\\
& && z \geq g(y) && && (2.38)\\
& \text{with:} && y \in D_y && && (2.39)
\end{aligned}
$$

where constraint (2.38) is the subproblem relaxation; the actual formulation has to be given case by case; quite often for example, the subproblem relaxation is given as a constraint not involving the objective function. Such a relaxation can have a deep impact on the performance (see [Hoo05b]) and enables the use of the master problem to compute also the y variable assignment $\bar{y}$ at the first iteration.

Chapter 3

Hybrid A&S methods in a Deterministic Environment

3.1 Introduction

This chapter tackles Allocation and Scheduling problems in a deterministic environment; having controlled activity durations, resource capacity and graph structure is a feasible hypothesis whenever actual variations have limited extent, or when conservative assumptions can be made. As a matter of fact, deterministic approaches are very often used in practice, last but not least for their (much) higher tractability compared to stochastic models.

In particular, here we tackle a specific problem, namely optimal mapping and scheduling of software applications on the Cell BE processor. Cell BE is multicore CPU by Sony, IBM and Toshiba, providing both scalable computation power and flexibility; it is already being adopted for many computational intensive applications like aerospace, defense, medical imaging and gaming. Despite of its merits, it also presents many challenges, as it is now widely known that is very difficult to program the Cell BE in an efficient manner. Hence, the creation of an efficient software development framework is becoming the key challenge for this computational platform.

Contribution Efficient programming requires to explicitly manage the resources available to each core, as well as the allocation and scheduling of activities onto them, the storage resources, the movement of data and synchronizations, etc. Our aim is to provide optimization methods to take care of several of such decision, thus setting the programmer free from their burden. We describe three approaches designed to deal with mapping and scheduling over Cell. In the first one, Logic Based Benders' Decomposition is used to achieve a robust and flexible hybrid solver; in particular, we investigate the recursive application of decomposition to allow the efficient solution of otherwise overly complex subproblems. Moreover, the use of NP-hard relaxations in the cut generation step is introduced, and motivated with empirical consideration and experimental results. The second proposed approach is CP based and leverages Priority Based Scheduling Ideas in the search strategy. Finally, we present a hybrid solver combining the former two to complement their strength.

Outline For the above mentioned allocation and scheduling problem we have developed three different approaches: each of them proved to have advantages and disadvantages for different classes of instances. In the following, we presente a solver based on a three-stage Logic based Benders Decomposition (LBD, section 3.3) and a pure CP one (section 3.4); a third hybrid algorithm follows (section 3.5) which tries to combine the strengths of the two approaches. Experimental results are reported in Section 3.6, while concluding remarks are in Section 3.7.

Publications This work is part of the CellFlow tool prototype developed by the MICREL lab, at University of Bologna, and has bee published in the conference papers [BLM+08, BLMR08, RLMB08]; a latter publication has been accepted for publication on the international journal "Annals of Operations Research".

3.2 Context and Problem Statement

We focus on a well-known multicore platform, namely the IBM Cell BE processor; Cell has already demonstrated impressive performance ratings in computationally intensive applications and kernels mainly thanks to its innovative architectural features [PFF+07, BAM07, OYS+07, LkKC+06]. In particular, here we address the problem of allocating and scheduling its processors, communication channels and memories. The application that runs on top of the target platform is abstracted as a Task Graph. Each task is labeled with its execution time, memory and communication requirements. The optimization metric we take into account is the application execution time that should be minimized. The Cell BE architecture is described in more details in Section 3.2.1, while Section 3.2.2 is devoted to the target application and Section 3.2.3 concludes with the problem statement.

3.2.1 CELL BE Architecture

In this section we give a brief overview of the Cell hardware architecture, focusing on the features that are most relevant for our optimization problem. Cell is a non-homogeneous multicore processor [PAB+05] which includes a 64-bit PowerPC processor element (PPE) and eight synergistic processor elements (SPEs), connected by an internal high bandwidth Element Interconnect Bus (EIB) [KPP06]. Figure 3.1 shows a pictorial overview of the Cell Broadband Engine Hardware Architecture.

The PPE is dedicated to the operating system and acts as the master of the platform, while the eight synergistic processors are optimized for compute-intensive applications. The PPE is a multi-threaded core and has two levels of on-chip cache, however, the main computing power of the Cell processor is provided by the eight SPEs. The SPE is a compute-intensive coprocessor designed to accelerate media and streaming workloads [FAD+05]. Each SPE consists of a synergistic processor unit (SPU) and a memory flow controller (MFC). The MFC includes a DMA controller, a memory management unit (MMU), a bus interface unit, and an atomic unit for synchronization with other SPUs and the PPE.

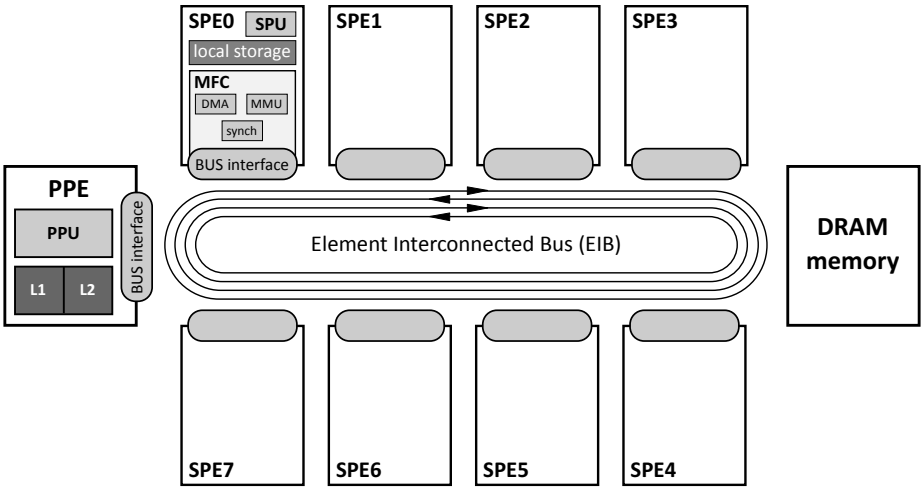


Figure 3.1: Cell Broadband Engine Hardware Architecture.

Efficient SPE software should heavily optimize memory usage, since the SPEs operate on a limited on-chip memory (only 256 KB local store) that stores both instructions and data required by the program. Additional data can be stored on a larger on-chip DRAM. The local memory of the SPEs is not coherent with the PPE main memory, and data transfers to and from the SPE local memories must be explicitly managed by using asynchronous coherent DMA commands. PPE-SPE and SPE-SPE communication takes place through a four ring Element Interconnected Bus; moreover, each ring supports multiple transactions, provided the two source-to-destination paths do not overlap (e.g. data transfers SPE0-SPE2 and SPE4-SPE7 can occur simultaneously). The whole EIB provides extremely high communication bandwidth.

The abstract platform A good architecture abstraction should take into account all (and only) the platform features which set non trivial constraints on the application execution. Failing to model a relevant element results in poor predictive performance, while taking into account negligible factors introduces unnecessary complication.

In the case at hand, since the core of the Cell computation performance is provided by the SPEs, the PPE is neglected in our abstract architecture description. The on-chip DRAM, having typically sufficient capacity to store all application data, can be disregarded as well. The single-thread SPUs within the SPEs are effectively modeled by renewable unary resources (let those be $SPE = spe_0, spe_1, \ldots$); the limited size local storage in this specific case is statically partitioned prior to the execution and can therefore be modeled by a finite capacity reservoir (with an exception, described later); the capacity of the device is referred to in the following as MC_k. The Element Interconnected Bus is disregarded, due to the very high provided bandwidth, and due to the choice to focus (for this work) on computation intensive applications. The presence of DMA controllers is not taken into account, but we plan to relax this restriction in the near future.

3.2.2 The Target Application

A target application to be executed on top of the hardware platform consists of a set of interdependent processes (tasks); in particular, task dependencies are due to data communications, performed by writing/reading a shared queue (in practice a FIFO buffer). Tasks can handle several input/output queues and we assume no cyclic dependence exists within a single iteration of the application.

Task execution is structured in three phases (see Figure 3.2): all input communication queues are read (Input Reading), task computation activity is performed (Task Execution) and finally all output queues are written (Output Writing). Each operation consists of an atomic activity and read operations are blocking, i.e. the task hangs on a reading operation if input data are not yet available; during the resulting idle time the processor is *not* released. Queues are read and written in a fixed order.

Figure 3.2: Three phases behavior of Tasks.

Each task t_i has an associated memory requirement representing storage locations required for computation data and for processor instructions. Such data can be either allocated on the local storage of the SPE where t_i runs, or in the shared memory (DRAM in figure 3.1). Clearly the duration of each task execution is related to the corresponding program data allocation; in particular, a local allocation results in lower execution time as accessing the DRAM has higher latency and requires the use of the EIB. Similarly, communication buffers can be allocated either on a local storage or on the on chip DRAM; in particular, in case of local allocation the buffer must be on the memory device of a SPE where either the producer or the consumer task run. Accessing the local storage of an SPE_j from a different SPE_i requires the use of the bus, but is nevertheless faster than accessing the on chip DRAM.

Note that, due to implementation issues, in case of DRAM allocation, when a task executes both the input/output buffers and the computation data must be temporary copied on the local memories. Such devices must therefore have enough free space for the copies; this is the aforementioned exception to the otherwise fully static partitioning of the local storage.

The Abstract Application The target application can be abstracted as a Task Graph $\langle T, A \rangle$ (see Section 2.1.2). In particular, it is natural to map each process to graph node/task t_i, while acyclic dependencies due to data communication are captured by the arcs $a_h = (t_i, t_j)$. Figure 3.3 shows a sample Task Graph, representing a software radio application.

Correctly modeling the three phase behavior is crucial for the accuracy of the model, and requires to split each task into a set of activities (with the meaning the term has in scheduling – see Sections 2.3.1 and 2.4). In detail, we introduce an activity ex_i for the execution phase and one activity for each queue read/write operation (rd_h, wr_h – see Figure 3.4).

As queues are read/written in fixed order, we assume the queue corresponding to arc a_h will be accessed before arc a_{h+1} and so on. In order to model the blocking read semantic, different rd_h activities are connected by loose standard

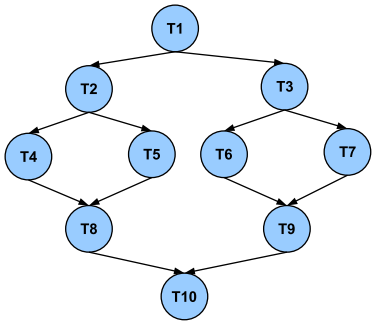


Figure 3.3: Task graph for a software radio application

end-to-start constraint; the execution activity is constrained to start after the last read operation and the write operations must follow immediately. The fact that the processor is not released in the (possible) idle time between read operations is modeled by introducing a dummy activity cv_i starting with the first read operation and ending with the last write operation; cv_i is the only activity actually requiring the SPE resource.

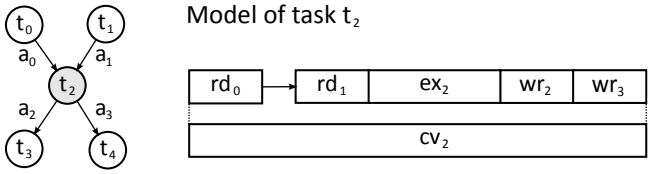


Figure 3.4: Model of the three phase behavior

We denote as $mem(t_i)$ the computation memory requirement of task t_i and as $comm(a_h)$ the size of the communication buffer associated to arc a_h. The impact of memory allocation choices on the durations can be taken into account by measuring the execution time corresponding to each mapping configuration; consequently we have for the execution activity a minimum duration $d(ex_i)$ in case the computation data are on the device store of the SPE where t_i runs (let this be $spe(t_i)$); conversely, the duration is maximum (say $D(ex_i)$) in case of DRAM allocation. Similarly, read and write activities corresponding to arc $a_h = (t_i, t_j)$ are associated with three duration values, namely:

1. a minimum value $d(rd_h)$ (resp. $d(wr_h)$) in case the buffer is on $spe(t_j)$ (resp. $spe(t_i)$).
2. an intermediate value $d^+(rd_h)$ (resp. $d^+(wr_h)$) in case the buffer is on $spe(t_i)$ (resp. $spe(t_j)$).
3. a maximum value $D(rd_h)$ (resp. $D(wr_h)$) in case the buffer is on the DRAM

3.2.3 Problem definition

The mapping and scheduling problem can now be formally stated; specifically, given:

1. an input application, described by a Task Graph $\langle T, A \rangle$ labeled with the attributes as in Section 3.2.2,
2. a target instantiation of the Cell platform, described by the number of available spes (denoted as $|SPE|$) and by the capacity of each local storage (denoted as MC_k),

the problem consist in mapping each task $t_i \in T$ to a SPE spe_k and providing a schedule (i.e. an assignment of start times to activities rd_h, ex_i, wr_h). Tasks are not subject to specific deadlines (although those can be easily introduced) and the objective is to minimize the application completion time (makespan). Of course memory capacity constraints cannot be violated at any point of time.

3.3 A Multistage LBD Approach

The problem we have to solve is a scheduling problem with alternative resources and allocation dependent durations. A good way of facing these kind of problems is via Benders Decomposition, and its Logic-based extension, as described in Section 2.5.1. In the Logic based Benders' Decomposition (LBD) approach the problem at hand is first decomposed into a *master problem (MP)* and a *subproblem (SP)*; then the method works by iteratively solving MP and feeding SP with the partial solution from MP. If such partial solution cannot be extended to a complete one a cut (Benders' cut) is generated forcing MP to yield a different partial solution. In case the extension is successful we have a feasible solution: this is stored and (possibly different) cuts are generated to guide MP towards a better partial solution. The efficiency of the technique greatly depends on the possibility to generate strong cuts.

Previous papers have shown the effectiveness of the method for similar problems (see Section 2.5.1). The allocation is in general effectively solved through Integer Linear Programming, while scheduling is better faced via Constraint Programming. In our case, the scheduling problem cannot be divided into disjoint single machine problems since we have precedence constraints linking tasks allocated on different processors, which also makes it much more difficult to compute effective Benders' cuts.

Effectiveness of Pure LBD Following this idea, in [BLM+08], we have implemented a Logic Based Benders' Decomposition based approach, similarly to [BBGM05b, BBGM06] (see figure 3.5A). In this case the master problem is the allocation of tasks to SPEs and memory requirements to storage devices (SPE & MEM stage), and the subproblem is computing a schedule with fixed resource assignment and fixed task durations (SCHED stage). We have experimentally experienced a number of drawbacks, the main one being the fact that for the problem at hand a two stage decomposition produces two unbalanced components. The allocation part is extremely difficult to solve while the scheduling part is indeed easier: on a test bench of 290 instances the average solution time ratio ranges between 10^3 to 10^4. Mainly due to that reason, the approach scales poorly on our problem.

The Multi-stage Approach We therefore have experimented a multi-stage decomposition, which is actually a recursive application of standard Logic based

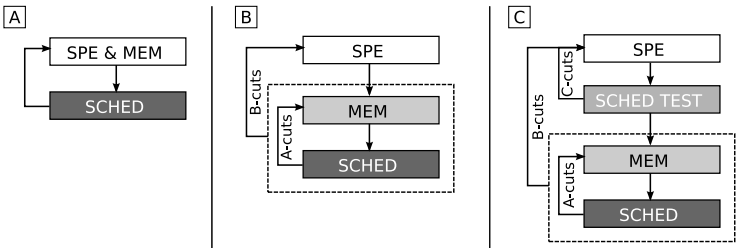


Figure 3.5: Solver architecture: (A) Standard LBD approach; (B) Two level Logic based Benders' Decomposition; (C) Schedulability test added

Benders' Decomposition (LBD), that aims at obtaining balanced and lighter components. The allocation part should be decomposed again in two subproblems, each part being easily solvable.

In figure 3.5B, at level one, the SPE assignment problem (SPE stage) acts as the master problem, while memory device assignment and scheduling as a whole are the subproblem. At level two (the dashed box in figure 3.5B) the memory assignment (MEM stage) is the master and the scheduling (SCHED stage) is the corresponding subproblem. In particular, the first step of the solution process is the computation of a task-to-SPE assignment; then, based on that assignment, in the second step (MEM stage) allocation choices for all memory requirements are taken. Deciding the allocation of tasks and memory requirements univocally defines execution and communication durations. Finally, a scheduling problem (SCHED stage) with fixed resource assignments and fixed durations is solved.

When the SCHED problem is solved (no matter if a solution has been found), one or more cuts (A-cuts in figure 3.5B) are generated to forbid (at least) the current memory device allocation and the process is restarted from the MEM stage; in addition, if the scheduling problem is feasible, *an upper bound on the value of the next solution is also posted*. When the MEM & SCHED subproblem ends (either successfully or not), more cuts (B-cuts) are generated to forbid the current task-to-SPE assignment. When the SPE stage becomes infeasible the process is over, converging to the optimal solution for the problem overall.

A First Difference with Classical LBD Note that the mentioned termination condition differs from that given in Section 2.5.1, requiring the master problem objective $\bar{z}$ to equal the corresponding subproblem optimal value β^*. Observe however that unlike in classical LBD, in the present approach, at every SP-feasible iteration a new MP constraint (an upper bound) is posted, requiring the next solution to improve the best one so far:

$$z < \beta^* \tag{3.1}$$

where we remind the NP is a minimization problem. One can see that $\bar{z} = \beta^*$ and constraint (3.1) imply a failure. Conversely, if β^* is not optimal, adding (3.1) keeps the MP feasible (as z is a lower bound on $f(x, y)$ and hence $z \leq \beta^*$). Therefore, *if the subproblem finds a solution with value* β^*, we have:

$$\beta^* \text{ optimal} \Leftrightarrow \text{next MP iteration infeasible}$$

which motivates our termination condition. If no solution is found by SP, no new constraint is added and the method behave as classical LBD. Indeed our scheme presents a second major difference compared to classical LBD, but this is described later on in Section 3.3.1.2.

Schedulability Test We found that quite often the SPE allocation choices are by themselves very relevant: in particular, a bad SPE assignment is sometimes sufficient to make the scheduling problem infeasible. Thus, after the task to processor allocation, we can perform a first schedulability test as depicted in figure 3.5C. In practice, if the given allocation with minimal durations is already infeasible for the scheduling component, then it is useless to complete it with a memory assignment that cannot lead to any feasible solution overall. In this case a cutting plane (C-cuts) is generated to avoid the generation of the same task-to-SPE assignment in the next iterations.

In the following each step of the multi-stage process is described in detail.

3.3.1 SPE Allocation

The computation of a task-to-SPE assignment is tackled by means of Integer Linear Programming (ILP). Given the Task Grah graph $\langle T, A\rangle$ and the platform with $SPE = \{spe_0, spe_1, \ldots\}$ the ILP model we adopted is very simple: this is a first visible advantage of the multi-stage approach. We introduce a decisional variable $\mathtt{T}_{ik} \in \{0, 1\}$ such that $\mathtt{T}_{ik} = 1$ if task i is assigned to SPE k. Then, the model to be solved is:

$$\begin{array}{lll}
\text{min:} & z & \\
\text{subject to:} & z \geq \displaystyle\sum_{t_i \in T} \mathtt{T}_{ik} & \forall spe_k \in SPE \qquad (3.2) \\
& \displaystyle\sum_{spe_k \in SPE} \mathtt{T}_{ik} = 1 & \forall t_i \in T \qquad (3.3) \\
\text{with:} & \mathtt{T}_{ik} \in \{0, 1\} & \forall t_i \in T, spe_k \in SPE
\end{array}$$

Constraints (3.3) state that each task can be assigned to a single SPE; constraints (3.2) are needed to express the objective function. Note that the actual makespan depends both on the resource allocation and on the decisions that will be taken in the scheduling stage; hence, here we adopt an easy to optimize objective function that tends to spread tasks as much as possible on different SPEs, which often provides good makespan values pretty quickly. Constraints (3.2) force the objective variable z to be greater than the number of tasks allocated on any PE.

The allocation model also features quite standard symmetry breaking ordering constraints to remove permutations of SPE having the same memory capacity MC_k.

3.3.1.1 Subproblem Relaxation

Constraints on the total duration of tasks on a single SPE were also added to a priori discard trivially infeasible solutions; this methodology in the LBD context is often referred to as "adding a subproblem relaxation" (see Section 2.5.1), and

is crucial for the performance of the method. In practice the model also contains the constraints:

$$\sum_{t_i \in T} d(t_i) \cdot \mathtt{T}_{ik} < mk^* \qquad \forall spe_k \in SPE \tag{3.4}$$

where $d(i)$ is the minimum possible duration of task i (reading and writing phases included), and mk^* is initially ∞ and the updated regularly to store the makespan value of the best solution found. Basically, this is a packing based bound, obtained by disregarding all precedence constraint and the considering minimum durations.

The longest path bound (see Section 2.4.2.2) provides a second, easy to plug, relaxation, obtained by disregarding *resources*. However, since the length of the critical path with no resource constraints is not affected by the SPE allocation, the longest path relaxation is employed in a pre-processing step.

3.3.1.2 A Second Difference with Classical LBD

A second relevant difference with the standard LBD scheme is given by the choice of the MP objective function, which in the SPE stage is not a lower bound on the overall problem objective. In principle, such a modification poses some issue for two steps of the iterative process, namely:

1. the termination condition $\bar{z} = \beta^*$,
2. the Benders' cut $z \geq \beta_{\bar{y}}(y)$

as for issue (1), the introductory part of Section 3.3 has already pointed out how the termination condition can be replaced by an equivalent one, not based on the equality of the MP and the SP objective. In order to show how we dealt with issue (2), we first recall the structure of the bounding function $\beta_{\bar{y}}(y)$ used in classical LBD:

$$\beta_{\bar{y}}(y) = \begin{cases} \beta^* \text{ if } y = \bar{y} \\ \beta'(y) \leq f(x, y) \text{ otherwise} \end{cases}$$

Now, the conjunction $z \geq \beta_{\bar{y}}(y) \wedge z < \beta^*$ has the effect of forbidding $\bar{y}$ and every y such that $\beta'(y) \geq \beta^*$; additionally, the provided bound $\beta'(y)$ may cut more solutions in the subsequent iterations. If we forgo the latter point, we could use a no-good constraint to forbid the y assignments removed by $z \geq \beta_{\bar{y}}(y) \wedge z < \beta^*$. Consequently, as the no-good does not need to involve the objective, any cost function can be used in MP. Note that the method remains complete as long as the assignment $\bar{y}$ is forbidden. In case SP is infeasible, the same reasoning holds by setting $\beta^* = \infty$.

The method described has a main advantage and a main disadvantage:

pro: any type of cost function (e.g. providing better initial solutions, or easier to optimize) can be used in the Master Problem

con: no lower bound $\beta'(y)$ is posted together with the cut

In the case at hand, the high level of decomposition (three stages) and the presence of general precedence constraints make it very hard to devise good bounding functions (see Section 2.4.2.2). In particular, most of the bounds one can think of are subsumed by the considered subproblem relaxation. In such a situation, the cost of loosing $\beta'(y)$ is negligible, while the advantages of a free choice of the MP objective can still be enjoyed.

Discussion The use of multistage Benders decomposition enables the complex resource allocation problem to be split into drastically smaller SPE and MEM components. However, adding a decomposition step hinders the definition of high quality heuristics in the allocation stages and makes the coordination between the subproblems a critical task. We tackle these issues by devising effective Benders' cuts and using poorly informative, but very fast to optimize, objective functions in the SPE and MEM stages. In practice the solver moves towards promising part of the search space by learning from its mistakes, rather than taking very good decisions in the earlier stages. Experimental results (reported in Section 3.6) show how in our case this choice pays off in terms of computation time, compared to using higher quality (but harder to optimize) heuristic objective functions, or less expensive (but weaker) cuts.

3.3.2 Schedulability test

We modified the solver architecture by inserting a schedulability test between the SPE and the MEM stage, as depicted in figure 3.5C. In practice, once a SPE assignment is computed, the system checks the existence of a feasible schedule using the model described in section 3.3.4, with all activity durations (execution, read, write) set to their minimum. If no schedule is found, then cuts (C-cuts in in figure 3.5C) that forbid at least the last SPE assignment are generated. Once a feasible schedule is found, the task-to-SPE assignment is passed to the memory allocation component.

3.3.3 Memory device allocation

Once tasks are assigned to processing elements, their memory requirements and communication buffers must be properly allocated to storage devices. Again we tackled the problem by means of Integer Linear Programming.

Given a task-to-SPE assignment we have that each t_i is assigned to a SPE, also referred to as $spe(t_i)$. For each task we introduce a boolean variable $\mathtt{M}_i$ such that $\mathtt{M}_i = 1$ if computation data of t_i are on the local memory of $spe(t_i)$. Similarly, for each arc/communication queue $a_h = (t_i, t_j)$, we introduce two boolean variables $\mathtt{W}_h$ and $\mathtt{R}_h$ such that $\mathtt{W}_h = 1$ if the communication buffer is allocated on the SPE $spe(t_i)$ (that of the producer), while $\mathtt{R}_h = 1$ if the buffer is allocated on the SPE $spe(t_j)$ (that of the consumer).

$$\mathtt{M}_i \in \{0,1\} \qquad \forall t_i \in T$$
$$\mathtt{W}_h \in \{0,1\}, \mathtt{R}_h \in \{0,1\} \qquad \forall a_h \in A$$

Note that, if for an arc $a_h = (t_i, t_j)$ it holds $spe(t_i) \neq spe(t_j)$, then either the communication buffer is allocated on the remote DRAM memory, or it is local to the producer or local to the consumer; if instead $spe(t_i) = spe(t_j)$, than the communication buffer is either on the DRAM, or it is local to *both* the producer and the consumer. More formally, for each arc $a_h = (t_i, t_j)$:

$$\mathtt{R}_h + \mathtt{W}_h \leq 1 \qquad \text{if } spe(t_i) \neq spe(t_j) \tag{3.5}$$
$$\mathtt{R}_h = \mathtt{W}_h \qquad \text{if } spe(t_i) = spe(t_j) \tag{3.6}$$

Constraints on the capacity of local memory devices can now be defined in terms of `M`, `W` and `R` variables. In order to do it we first take into account the memory

needed to store all data *permanently* allocated on the local device of SPE k, by defining:

$$base(spe_k) = \sum_{\substack{a_h=(t_i,t_j)\\ spe(t_j)=spe_k}} comm(h)\cdot \mathtt{R}_h + \sum_{spe(t_i)=spe_k} mem(i)\cdot \mathtt{M}_i + \sum_{\substack{a_h=(t_i,t_j)\\ spe(t_i)=k\\ spe(t_i)\neq spe(t_j)}} comm(h)\cdot \mathtt{W}_h$$

where $mem(i)$ is the amount of memory required to store internal data of task t_i and $comm(r)$ is the size of the communication buffer associated to arc a_r.

Due to implementation issues, however, during execution of task t_i all its data should be locally transferred on $spe(t_i)$ even if they are permanently allocated on the remote DRAM. Therefore, beside the "base usage" $base(spe_k)$ we need to allocate space for transferring data for executing the task. We take into account this behavior by posting, $\forall spe_k \in SPE$ and $\forall t_i$ such that $spe(t_i) = k$, the constraints:

$$\begin{aligned} base(spe_k) + \sum_{a_h=(t_j,t_i)} (1-\mathtt{R}_h)\cdot comm(a_h) + \\ (1-\mathtt{M}_i)\cdot mem(t_i) + \sum_{a_h=(t_i,t_j)} (1-\mathtt{W}_h)\cdot comm(a_h) \leq MC_k \end{aligned} \tag{3.7}$$

Constraints (3.7) force to spare on the local memory of each SPE enough storage to enable the execution of the task copying the largest amount of data. Reductions due to already locally allocated data are taken into account.

3.3.3.1 Cost Function and Subproblem Relaxation

The objective function to be minimized is a lower bound on the makespan, given by the subproblem relaxation. The basic idea is once again (see Section 3.3.1.1) to use a packing and a path based bound. More in detail, we observe that (1) the length of the longest path and (2) the total duration of tasks on a single SPE must be lower than the best makespan found so far. Both the mentioned features are affected by memory allocation choices.

We first define for each task t_i a real valued variable $\mathtt{ED}_i$ representing its execution time, and for each arc a_h two variables $\mathtt{RD}_h$ and $\mathtt{WD}_h$ representing the time needed to read and to write the corresponding buffer. In particular, $\forall t_i \in T$:

$$\mathtt{ED}_i = D(ex_i) - [D(ex_i) - d(ex_i)]\cdot \mathtt{M}_i$$

Where we remember that $D(ex_i)$ is the duration of the execution phase of t_i when the computation data are allocated on remote memory, and $d(ex_i)$ is the duration with local data. Also for each arc $a_h = (t_i, t_j)$ we introduce:

$$\mathtt{RD}_h = \begin{cases} D(rd_h) - [D(rd_h) - d(rd_h)]\cdot \mathtt{R}_h & \text{if } spe(t_i) = spe(t_j) \\ D(rd_h) - [D(rd_h) - d^+(rd_h)]\cdot \mathtt{W}_h + & \\ \quad - [D(rd_h) - d(rd_h)]\cdot \mathtt{R}_h & \text{if } spe(t_i) \neq spe(t_j) \end{cases} \tag{3.8}$$

$$\mathtt{WD}_h = \begin{cases} D(wr_h) - [D(wr_h) - d(wr_h)]\cdot \mathtt{W}_h & \text{if } spe(t_i) = spe(t_j) \\ D(wr_h) - [D(wr_h) - d^+(wr_h)]\cdot \mathtt{R}_h + & \\ \quad - [D(wr_h) - d(wr_h)]\cdot \mathtt{W}_h & \text{if } spe(t_i) \neq spe(t_j) \end{cases} \tag{3.9}$$

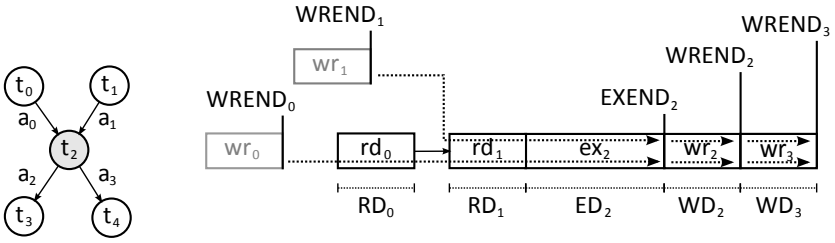


Figure 3.6: Schema of path based scheduling relaxation in MEM stage

Where for each a_h the values $D(rd_h)$, $d^+(rd_h)$ and $d(rd_h)$ are the time needed to read the associated buffer in case it is allocated on remote memory ($\mathtt{R}_h = \mathtt{W}_h = 0$), on the local memory of another SPE ($\mathtt{R}_h = 0, \mathtt{W}_h = 1$), or on the local memory of current SPE ($\mathtt{R}_h = 1$). Values $D(wr_h)$, $d^+(wr_h)$ and $d(wr_h)$ have the same meaning with regard to writing the buffer.

We use the introduced duration variables to define two scheduling relaxations, respectively based on the a packing problem obtained by disregarding precedence relations, and on the longest path obtained by disregarding resource constraints. Both relaxations set lower bounds on an additional variable MK representing the value of the makespan approximation, and constrained to be lower than the best actual makespan found to far:

$$\mathtt{MK} < mk^*$$

Minimizing MK is the objective of the memory allocation stage. The packing based relaxation consists in a constraint for each $spe_j \in SPE$; in practice, the total duration of the tasks on each processor sets a bound on the makespan relaxation (as those tasks must execute sequentially):

$$\sum_{t_i \in T} \mathtt{ED}_i + \sum_{\substack{a_h=(t_i,t_j) \\ spe(t_i)=spe_k}} \mathtt{WD}_h + \sum_{\substack{a_h=(t_i,t_j) \\ spe(t_j)=spe_k}} RD_h \leq \mathtt{MK}$$

The second scheduling relaxation sets a on MK a lower bound based on the length of the path on the graph. On computation purpose, we introduce real valued "end" variables for each execution phase ($\mathtt{EXEND}_i$, see figure 3.6) and buffer writing operation ($\mathtt{WREND}_h$). Each task reads and writes buffers in a pre-specified fixed ordered, and thus each $\mathtt{EXEND}_i$ variable is constrained to be greater than the maximum of (see Figure 3.6):

- the $\mathtt{WREND}_h$ variable of each predecessor arc a_h,
- plus the time required to read the corresponding buffer (given by $\mathtt{RD}_h$),
- plus the time to perform all the read operations after the h$^{\text{th}}$ one (given by the respective RD variables),
- plus the task duration (given by $\mathtt{ED}_i$)

Figure 3.6 shows the variables taking part to the longest path based relaxation; in particular, duration variables are reported under the activities, while end variables are above the corresponding ex and wr activities; the path corresponding to the constraints setting a lower bound on the end variables are

shown as dashed arrows. The $\texttt{EXEND}_i$ variables further constrain the makespan lower bound MK:

$$\forall t_i \in T \ : \ \texttt{EXEND}_i \leq \texttt{MK}$$

The MK variable, being the objective of a minimization problem, will be set by the optimizer to the maximum between the length of the longest path and the maximum total duration of tasks allocated on a single SPE.

3.3.4 Scheduling subproblem

Once SPE and memory allocation choices have been made, the problem reduces to disjunctive scheduling (as all resource are unary and renewable) with general precedence constraints. We adopt quite a conventional CP model (with start/end variables for each activity – ses Section 2.3.1), based on the decomposition of each task in set of activities shown in Section 3.2.2.

Variables: We introduce for each activity ex_i, rd_h, wr_h start and end variables, namely $\texttt{XS}_i, \texttt{XE}_i$, $\texttt{RS}_h, \texttt{RE}_h$, $\texttt{WS}_h, \texttt{WE}_h$ (where "X" refers to eXecution, "R" to read and "W" to write). All variables are in the integer range $[0..eoh)$, where the end of horizon eoh is set to the sum of the worst case execution time at the first iteration, and to the best makespan value mk^* afterwards.

Finally, start / end variables are introduced for the macro activities cv, namely $\texttt{CS}_i, \texttt{CE}_i$ covering the whole task execution; moreover those macro activities have non-fixed duration, modeled by using a variable $\texttt{CD}_i$, lower bounded by the sum of the durations of activities related to t_i and with no upper bound (in practice eoh is used). The objective function to minimize is the makespan.

Temporal Constraints: Read and write operations are performed in a fixed sequece; let $rd_{h_0} \dots rd_{h_{r-1}}$ be the sequence of reading activities for task t_i and $wr_{r_r}, \dots, wr_{h_{k-1}}$ the sequence of writing activities, then:

$$\begin{aligned}
&\forall j = 0, \dots, r-2 && \texttt{RE}_{h_j} = \texttt{RS}_{h_{j+1}} \\
& && \texttt{RE}_{h_{r-1}} = \texttt{XS}_i \\
& && \texttt{XE}_i = \texttt{WS}_{h_r} \\
&\forall j = r, \dots, k-2 && \texttt{WE}_{h_j} = \texttt{WS}_{h_{j+1}}
\end{aligned}$$

Each communication buffer must be written before it can be read. Thus, for each precedence constraint $a_h = (t_i, t_j)$ in the Task Graph, we post:

$$\forall a_h = (t_i, t_j) \in A \qquad \texttt{WE}_h \leq \texttt{RS}_h$$

All activities (except for the macro activities) have fixed durations, assigned during the SPE and memory allocation stages. Moreover, the macro activity must cover the whole extent of task t_i execution; this is enforced by posting, $\forall t_i \in T$:

$$\texttt{CS}_i = \texttt{RS}_{h_0} \qquad \texttt{CE}_i = \texttt{WE}_{h_{k-1}} \tag{3.10}$$

Resource Constraints: Processing elements are modeled as unary resources, by means of cumulative constraints; in particular, let $[RQ(k)]$ be an array such that $RQ(k)_i = 1$ if $spe(t_i) = spe_k$, otherwise $RQ(k)_i = 0$; then we can post:

$$\texttt{cumulative}(\,[\texttt{CS}_i]\,,[\texttt{CD}_i],[RQ(k)],1) \qquad\qquad \forall spe_k \in SPE \tag{3.11}$$

where $[\texttt{CS}_i]$ is an array with the start variables of the macro activity corresponding to taks t_i and $[\texttt{CD}_i]$ is the array with their duration variables. Time-table and precedence based energetic reasoning are the adopted filtering techniques (see Section 2.3.2).

3.3.5 Benders' Cuts

Benders cuts are used in the Logic based Benders Decomposition schema to control the iterative solution method and are extremely important for the success of the approach. In a multi stage Benders Decomposition approach we have to define Benders cuts for each level; as discussed in section 3.3.1 we chose to focus on generating strong valid cuts, rather than on trying to devise a tight makespan approximation in the early allocation stages. In the followings, both level 1 and level 2 cuts are specified.

Level 2 Cuts Let σ be a solution of the MEM stage, that is an assignment of memory requirements to storage devices. If $\texttt{X}$ is a variable, we denote as $\sigma(\texttt{X})$ the value it takes in σ. The level 2 cuts we used are:

$$\sum_{\sigma(\texttt{M}_i)=0} \texttt{M}_i + \sum_{\sigma(\texttt{R}_h)=0} \texttt{R}_h + \sum_{\sigma(\texttt{W}_h)=0} \texttt{W}_h \geq 1 \tag{3.12}$$

This forbids the last solution σ and all solutions one can obtain from σ by remotely allocating one or more requirements previously allocated locally: this would only yield longer task durations and worse makespan. In practice we ask for at least one previously remote memory requirement to be locally allocated.

Level 1 Cuts Similarly, level 1 cuts (B-cuts in figure 3.5B), between the SPE and the MEM & SCHED stage must forbid (at least) the last proposed SPE assignment. Again, let σ be a solution to the SPE allocation problem. A valid B-cut is in the form:

$$\sum_{\sigma(\texttt{T}_{ik})=1} (1 - \texttt{T}_{ik}) \geq 1 \tag{3.13}$$

Note, that since the processing elements are symmetric resources, we can forbid both the last assignment and all its permutations. This can be done by replacing cut (3.13) with a family of cuts, which are used in the solver. For each processing element spe_k we introduce a variable $\texttt{TS}_k \in \{0,1\}$ such that $\texttt{TS}_k = 1$ iff all and only the tasks assigned to SPE k in σ are on a single SPE in a new solution. This is enforced by the constraints:

$$\forall spe_k, spe_r \in SPE \qquad \sum_{\sigma(\texttt{T}_{ir})=1} (1 - \texttt{T}_{ir}) + \sum_{\sigma(\texttt{T}_{ir})=0} \texttt{T}_{ir} + \texttt{TS}_k \geq 1$$

We can then forbid the assignment σ and all its permutations by posting the constraint:

$$\sum_{spe_k \in SPE} \mathtt{TS}_k \leq |SPE| - 1 \tag{3.14}$$

With the aid of auxiliary variables a polynomial number of constraints is sufficient to forbid all permutations of the last assignment.

3.3.5.1 Cut refinement

The level 1 and level 2 cuts we have just presented are sufficient for the method to work, but they are too weak to make the solution process efficient enough; we therefore need stronger cuts. For this purpose we have devised a refinement procedure (described in Algorithm 1) aimed at identifying a subset of assignments which are responsible for the infeasibility. We apply this procedure to (3.12) and (3.13). The described cut refinement method has some analogies with the one proposed in Cambazard and Jussien [CJ05], where explanations are used to generate logic based Benders cuts; many similarities finally exist between this algorithm and those described in [dSNP88] and [Jun04].

Algorithm 1: Refinement procedure

Data: the set of all master problem decision variables in the original cut (let this be $\mathtt{X}_{i_k}$)
Result: an index lb such that variables $\mathtt{X}_{i_0}, \ldots \mathtt{X}_{i_l b}$ are responsible for the infesibility
sort the $\mathtt{X}$ set in non-increasing order according to a relevance score
set $lb = 0$, $ub = |\mathtt{X}|$, $n = lb + \lfloor \frac{ub-lb}{2} \rfloor$
while $ub > lb$ **do**
  feed subproblem with current MP solution
  relax subproblem constraints linked to variables $\mathtt{X}_{i_n}, \ldots \mathtt{X}_{i_{|\mathtt{X}|-1}}$
  solve subproblem to feasibility
  if $feasible$ **then**
    set $lb = n + 1$
  else
    set $ub = n$
  restore relaxed subproblem constraints
return lb

Algorithm 1 refines a cut produced for the master problem, given that the correspondent subproblem is infeasible with the current master problem solution. Note that in the proposed LBD approach, one may add the makespan upper bound ($makespan < mk^*$) to the subproblem to obtain a valid infeasible SP (as $mk^* < mk^*$ produces a failure).

An sample algorithm execution is shown in Figure 3.7, where $\mathtt{X}_{i0}, \ldots \mathtt{X}_{i5}$ are variables involved in the Benders cut we want to refine. In first place, all master problem variables in the original cut (let them be in the X set) are sorted according to some relevance criterion: least relevant variables are at the end of the sequence (Figure 3.7-1). The algorithm iteratively updates a lower bound

(lb) and an upper bound (ub) on the number of decision variables which are responsible for the infeasibility; initially $lb = 0$, $ub = |\mathtt{X}|$. At each iteration an index n is computed and all subproblem constraints linked to decision variables of index greater or equal to n are relaxed; in Figure 3.7-1 it is $n = 0 + \lfloor \frac{0+6}{2} \rfloor = 3$. Then, the subproblem is solved: if a feasible solution is found we know that at least variables from $\mathtt{X}_{i_0}$ to $\mathtt{X}_{i_n}$ are responsible of the infeasibility and we set the lower bound to $n + 1$ (figure 3.7-2). If instead the problem is infeasible (see figure 3.7-3), we know that variables from $\mathtt{X}_{i_0}$ to $\mathtt{X}_{i_{n-1}}$ are sufficient for the subproblem to be infeasible, and we can set the upper bound to n. The process stops when $lb = ub$. At that point we can restrict the original cut to variables from $\mathtt{X}_{i_0}$ to $\mathtt{X}_{i_{n-1}}$.

Cut Refinment Specification In order to apply Algorithm 1 to an actual cut one must specify (i) the variables in the `X` set; (ii) the relevance criterion to sort the `X` set; (iii) how to relax constraints related to a variable in `X`.

In case of level 2 cuts, the `X` set contains all `M`, `R` and `W` variables in the current cut (3.12); the relevance score is the difference between the current duration of the activity they refer to in the scheduling subproblem (resp. execution, buffer reading/writing) and the minimum possible duration of the same activity. Relaxing constraints linked to `M`, `R` and `W` variables means to set the duration of the corresponding activities to their minimum value.

Level 1 cuts are handled similarly. In that case the `X` set contains variables $\mathtt{T}_{ik}$ such that $\mathtt{T}_{ik} = 1$ in the current solution; the relevance score is the inverse of the task duration, since the longer the task, the less likely it is discarded by the refinement procedure. Relaxing the constraints linked to a $\mathtt{T}_{ik}$ means to relax all the consequences of the assignment, that is:

a. the duration of all reading, writing and execution activities related to task t_i must be set to the minimum possible value

b. buffer allocation constraints of type (3.5) and (3.6) related to t_i must be removed (as t_i is no longer assigned to a SPE)

c. all types of memory requirements of the task (buffers and computation data) must be set to 0 (as task t_i is no longer using the storage device of any SPE)

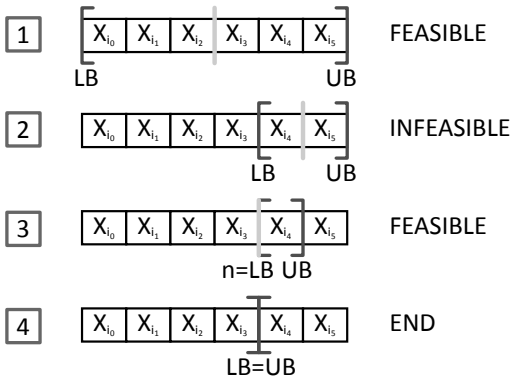


Figure 3.7: Refinement procedure: an example

Note that the refinement of level 2 cuts requires to repeatedly solve (relaxed) scheduling problems, which are by themselves NP-hard; the situation is even worse for level 1 cuts, since the subproblem is in this case MEM & SCHED, which is iteratively solved. This is expensive, but still perfectly reasonable as long as the time to solve the master problem (let this be T_M) is higher than the time to solve the subproblem (T_S), and the time spent to produce cuts saves many master problem iterations. For example, if $T_M = \alpha \cdot T_S$ holds, we can reasonably solve T_S up to α times in the worst case (i.e. if only one iteration is saved). In [BLM+08] we have experimentally found that the effort spent in strengthening the cuts actually pays off.

Finally, the described refinement procedure finds the minimum set of *consecutive* variables in $\mathtt{X}$ which cause the infeasibility of the subproblem, *without changing the order of the sequence*. Note however that it is possible that some of the variables from $\mathtt{X}_{i_0}$ to $\mathtt{X}_{i_{n-1}}$ are not actually necessary for the infeasibility. To overcome this limitation Algorithm 1 is used within an iterative conflict detection procedure, such as the one described in [dSNP88] or [Jun04] (QUICKXPLAIN), to find a minimum conflict set. In particular, we implemented the iterative method proposed in [dSNP88] and used the refinement procedure described above to speed up the process: this enables the generation of even stronger (but more time consuming) cuts, used in the experiments reported in section 3.6.

3.4 A Pure CP Approach

The effectiveness of Priority Based Scheduling (or list scheduling, see Section 2.4.3.3) on many Embedded System scheduling problem provided motivation for the development of a second approach. Ideally, we wish to exploit the ability of PRB to produce pretty good solutions in a fraction of second, while retaining completeness at the same time. We came up with a CP solver adopting a PRB like search strategy, interleaving allocation and scheduling decisions. The possible allocation choices are modeled by means of the following variables:

$$\begin{aligned}
&\mathtt{TPE}_i \in \{0, \ldots |SPE| - 1\} && \forall t_i \in T \\
&\mathtt{M}_i \in \{0, 1\} && \forall t_i \in T \\
&\mathtt{APE}_r \in \{-1, \ldots, ...|SPE| - 1\} && \forall a_h \in A
\end{aligned}$$

$\mathtt{TPE}_i$ is the processing element assigned to task t_i. Similarly, if $\mathtt{APE}_h = k$ then the communication buffer related to arc a_h is on the local memory of the processing element spe_k, while if $\mathtt{APE}_h = -1$ the communication buffer is allocated on the remote memory. Finally, $\mathtt{M}_i$ is 1 if program data of task t_i are allocated locally to the same processor of task t_i. The architectural constraints on the allocation of the buffer for arc a_h connecting tasks t_i and t_j translate to:

$$\mathtt{APE}_h = \mathtt{TPE}_i \vee \mathtt{APE}_h = \mathtt{TPE}_j \vee \mathtt{APE}_h = -1$$

that is, a communication buffer can be allocated either on the local memory of the source task, or that of the target task, or on the remote memory. In order to improve propagation, the following set of redundant constraints is also posted:

$$\mathtt{TPE}_i \neq k \wedge \mathtt{TPE}_j \neq k \Rightarrow \mathtt{APE}_h \neq k \qquad \forall spe_k \in SPE$$

From a scheduling standpoint, the model is very similar to the one presented in section 3.3.4. A set of execution (ex_i), reading (rd_r) and writing (wr_r) activities are used to model the execution phases of each task and a corresponding starting time variable is defined for each of them; however, here activity durations have to be decided during search. Hence the CP model has the following set of variables:

$$
\begin{aligned}
&\mathtt{XS}_i, \mathtt{XE}_i \in [0..eoh] && \forall t_i \in T \\
&\mathtt{RS}_h, \mathtt{RE}_h \in [0..eoh] && \forall a_h \in A \\
&\mathtt{WS}_h, \mathtt{WE}_h \in [0..eoh] && \forall a_h \in A \\
&\mathtt{ED}_i \in [d(ex_i)..D(ex_i)] && \forall t_i \in T \\
&\mathtt{RD}_h \in [d(rd_h)..D(rd_h)] && \forall a_h \in A \\
&\mathtt{WR}_h \in [d(wr_h)..D(wr_h)] && \forall a_h \in A
\end{aligned}
$$

where $\mathtt{XS}_i, \mathtt{XE}_i$ are the start and end variable of ex_i, while $\mathtt{RS}_h, \mathtt{RE}_h, \mathtt{WS}_h, \mathtt{WE}_h$ the start and end variables respectively for rd_h and wr_h. Variable $\mathtt{ED}_i$ represents the duration of the execution of task t_i, $\mathtt{WD}_h$ and $\mathtt{RD}_h$ respectively are the time needed to write and read buffer h. Durations are linked to the allocation choices; this is modeled by the following constraints:

$$
\begin{aligned}
&\forall t_i \in T: && \mathtt{ED}_i = d(ex_i) + [D(ex_i) - d(ex_i)] \cdot (1 - \mathtt{M}_i) \\
&\forall a_h \in A: && \mathtt{WD}_i = d(wr_h) + [D(wr_h) - d^+(wr_h)] \cdot (\mathtt{APE}_h = -1) + \\
& && \quad + [d^+(wr_h) - d(wr_h)] \cdot (\mathtt{APE}_h \neq \mathtt{TPE}_i) \\
&\forall a_h \in A: && \mathtt{RD}_i = d(rd_h) + [D(rd_h) - d^+(rd_h)] \cdot (\mathtt{APE}_h = -1) + \\
& && \quad + [d^+(rd_h) - d(rd_h)] \cdot (\mathtt{APE}_h \neq \mathtt{TPE}_j)
\end{aligned}
$$

where the source of each arc is referred to as t_i ad the destination as t_j. The precedence constraints are the same described in Section 3.3.4; in particular the reading operations are performed before the execution, and all writing operations start immediately after. All resource constraints are triggered when the TPE allocation variables are assigned; in particular if $\mathtt{TPE}_i = k$, then the macro activity cv_i requires 1 unit of spe_k. The resource capacity constraint is enforced by timetabling and precedence energetic filtering, as in Section 3.3.4.

3.4.1 Search strategy

The model is solved by means of a dynamic search strategy where resource allocation and scheduling decisions are interleaved.

We chose this approach since most resource filtering methods are not able to effectively prune start and end variables as long as the time windows are large and no task (or just a few of them) has an obligatory region: in particular it is difficult, before scheduling decisions are taken, to effectively exploit the presence of precedence relations and makespan bounds. In our approach, tasks are scheduled immediately after they are assigned to a processing element, this triggers updates to the time windows for all tasks linked by precedence relations.

A considerable difficulty in our specific case is set by the need to assign each task and arc both to a processing element and to a storage device: this makes the number of possible choices too big to completely define the allocation of each task before it is scheduled. Therefore we chose to postpone the memory allocation stage after the main scheduling decisions are taken, as depicted in figure 3.8A.

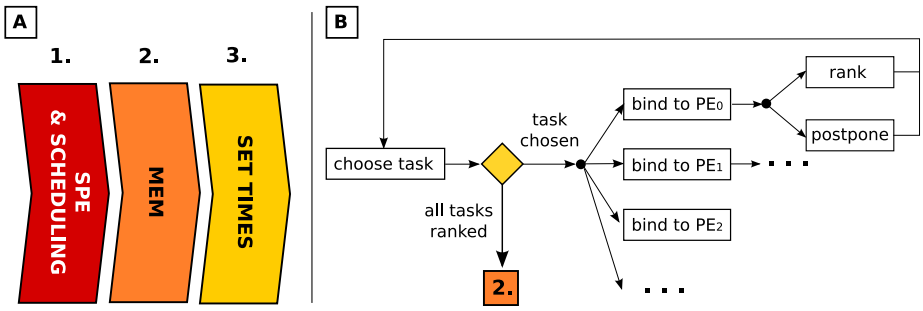


Figure 3.8: A: Structure of the dynamic search strategy; B: Operation schema for phase 1

Since task durations directly depend on memory assignment, scheduling decisions taken in phase 1 of Figure 3.8 had to be relaxed to enable the construction of a *fluid* schedule with variable durations. In practice we adopted a Precedence Constraint Posting approach (see Section 2.3.3), by just adding precedence relations to fix the order of tasks at the time they are assigned to SPEs: they will be given a start time (phase 3 in figure 3.8A) only once the memory devices are assigned. Note this time setting step has polynomial complexity, since the task order is already decided. Figure 3.9B shows an example of fluid schedule for activities in figure 3.9A; tasks have variable durations and precedence relations have been added to fix the order of the tasks on each SPE; Figure 3.9D shows a corresponding schedule where all durations are decided (a grey box means the minimum duration is used, a white box means the opposite).

Phase 1 In deeper detail, the SPE allocation and scheduling phase operates according to the schema of figure 3.8B: first, the task with minimum start time is selected (ties are broken looking a the lowest maximum end time). Second, the SPEs are sorted according to the minimum time at which the chosen task could be scheduled on them (let spe_0 be the first in the sequence, where the task can be scheduled at its minimum start time, and so on). Then a choice point is open, with a branch for each SPE.

Along each branch the task is bound to the corresponding resource and a *rank or postpone* decision is taken: we try to rank the task immediately after the last activity on the selected resource, otherwise the task is postponed and not considered until its minimum start time changes due to propagation (see Section 2.3.3). The process is reiterated as long as there are unranked tasks.

Phase 2 In phase 2, memory requirements are allocated to storage devices; this is done implicitly by choosing for each activity (ex_i, rd_r, wr_r) one of the possible duration values. In particular at each step the activity with the highest

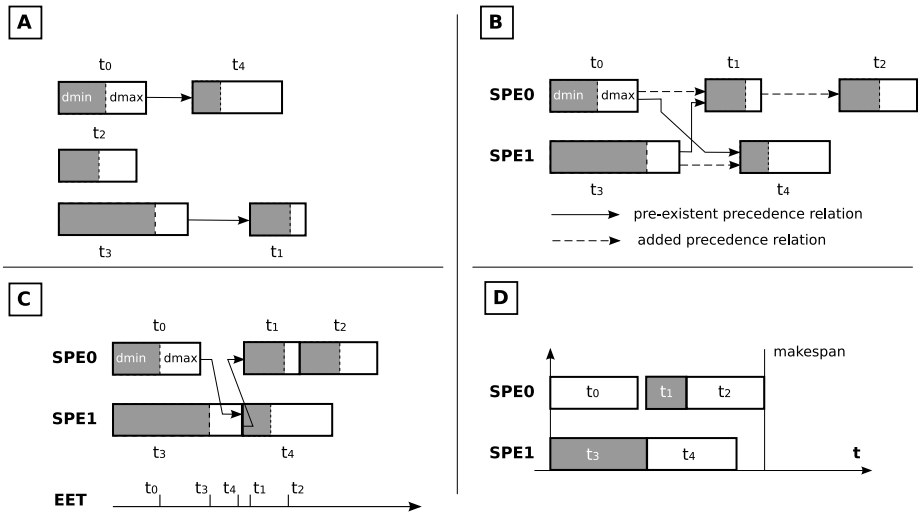


Figure 3.9: A: initial state of activities of a Task Graph; B: a fluid schedule; C: EET ordering; D: a possible fixed time schedule

earliest end time (EET) is selected, which intuitively is most likely to cause a high makespan value. For example in figure 3.9C the activity t_2 is considered first, followed by t_1, t_4, t_3 and t_0. Then a choice point is open. Let the activity be *act* and its duration variable DUR; the choice point is in the form:

$$\texttt{DUR} = min(\texttt{DUR}) \text{ OR } \texttt{DUR} > min(\texttt{DUR}) \tag{3.15}$$

where the two sides of the disjunction correspond to two distinct search branches; the assignment $\texttt{DUR} = min(\texttt{DUR})$ implicitly sets a memory allocation variable (due to propagation). For example in figure 3.9 tasks t_2, t_4, t_0 are assigned their minimum duration (which implies local data allocation), while the remaining activities are set to their maximum.

Phase 3 In phase 3 a start time is assigned to each task; this is done in polynomial time without backtracking, since all resource constraint are already solved and duration variables are assigned.

Since the processing elements are symmetric resources the procedure embeds quite standard symmetry breaking techniques [GS00] to prevent the generation of useless branches in the search tree.

To further prevent trashing due to early bad search choices, we modified the search procedure just outlined by adding randomization (see Section 2.2.2) and restarts. In particular, phase 1 is modified by performing each time a few random permutations in the order SPEs are assigned to tasks: the task to be mapped and scheduled is instead still selected in a deterministic fashion; phase 2 is randomized by inverting each time the order of branching choices in (3.15) with a certain probability (< 0.5). Some restarts are introduced by performing binary search on the makespan variable.

The main drawback of list scheduling-like search is that an early bad choice is likely to lead to thrashing, due to the size of the search space resulting from the

mixture of allocation and scheduling decisions; a more conventional two phase allocation and scheduling approach, with all the allocation variables assigned before any scheduling decision is taken, would be able to recover faster from such a situation. Preliminary tests were performed on a set of realistic instances in order to compare the mixed schedule-and-allocate strategy against a pure two phase one; they showed how the mixed strategy was definitely better whenever the graph has a sufficient amount of precedence relations, as it happens in many cases of practical interest.

3.5 A Hybrid CP-LBD Approach

The experiments performed on a set of instances with different characteristics (see section 3.6) showed how the two solvers proposed in this chapter (MS-LBD and CP) have somehow *complementary strengths*. In particular, the decomposition based solver is effective in dealing with resource allocation choices, being able to significantly narrow the search space by generating strong cuts. On the other hand, the main drawback of the approach are the structural lack of good SPE allocation heuristics and the loose links between variables in different decomposition stages; this results into a poor ability of finding good solutions quickly, as hinted to by figure 3.10, which shows the distribution of the ratio between the iteration when the best solution is found and the total number of iterations (out of a sample of 200 instances). As one can see, in most cases the best solution is found close to the end of the search process.

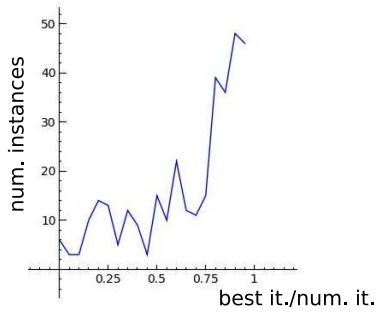


Figure 3.10: Distribution of the ratio between the iteration when the best solution if found and the total number of iterations for the MS-LBD solver

On the contrary, the CP approach (especially if some kind of restart strategy is introduced) is much faster in producing quite good solutions, but has difficulties in handling memory allocation choices, thus making hard for the solver to find the optimal solutions and prove optimality.

This complementarity motivated the introduction of a hybrid solver, that counters the shortcomings of one approach with the strengths of the other. The basic idea is to iterate between the CP solver and the decomposition based one, feeding MS-LBD with the solutions found by CP and injecting in the CP model the cuts generated by the LBD solver; at each step, the solvers are required to find a solution better than the best one found so far. Finally, the CP solver underwent some modifications to be better exploited in the new approach.

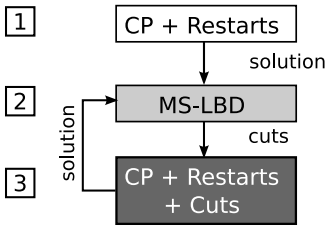


Figure 3.11: Structure of the hybrid approach

The working scheme for the hybrid solver is depicted in figure 3.11, where steps 2 and 3 run with a time limit and form the main loop. In particular, the modified CP solver is always run with a fixed time limit, while the time limit for the MS-LBD approach is dynamically adjusted and stretched whenever the solver does not have enough time to generate at least one cut. Finally, a bootstrapping stage (step 1) is added to start the MS-LBD solver with a solution as good as possible.

Note that the CP and the MS-LBD approaches are complete solvers: in case they terminate within the time limit either they find the optimal solution or they prove infeasibility. In both cases the process can be terminated; if an infeasibility is reported, either the last solution found by any of the two solvers is the optimal one, or the problem is actually infeasible (if no solution was found so far). In case the maximum solution time is hit at any solution step, the partial result obtained (the generated cuts for the MS-LBD solver, the solution found for the CP solver) is used to prime the subsequent solution step. Overall, the hybrid process is convergent, provided at least a cut per iteration is generated: the proof follows trivially from the completeness of the MS-LBD solver.

3.5.1 The modified CP solver

In order to build the hybrid system, we devised and used a specialized version of the CP solver. Namely the approach described in section 3.4 was modified 1) by adopting a more aggressive restart policy and 2) by adding a cut generation step, similarly to the MS-LBD approach. Binary search is no longer employed.

In particular, the CP solver is run with a (quite low) fail limit and restarted whenever this limit is met; at each restart the fail limit is increased, so that the method remains complete. This modification increases the chance to quickly find good solutions, but makes even more difficult to prove optimality.

To overcome this difficulty and to better guide the CP solver away from bad quality or infeasible solutions, cuts are periodically generated after some restarts as follows. First a complete resource allocation is computed (SPEs and memory devices), without taking any scheduling decision. Then scheduling is attempted, thus checking the given allocation for feasibility; in case the allocation is infeasible it provides a nogood:

$$\left[\bigvee_{t_i \in T} \mathtt{TPE}_i \neq \sigma(\mathtt{TPE}_i)\right] \vee \left[\bigvee_{\sigma(\mathtt{M}_i)=1} \mathtt{M}_i = 0\right] \vee \left[\bigvee_{a_h \in A} \mathtt{APE}_r \neq \sigma(\mathtt{APE}_h)\right]$$

where $\sigma(\mathtt{X})$ denotes the value variable X assumes in the allocation; the nogood is then refined by means of the QuickXplain algorithm, described in [Jun04], similarly to what is done in the MS-LBD solver. Such strengthened nogoods narrow the allocation search space, making the restart based solver more effective.

In the hybrid system the CP solver is first used to perform a bootstrap (see figure 3.11, step 1); it is then invoked at each iteration with the main objective of improving the best solution so far (and possibly proving optimality). Executions the CP solver alternate with those of the MS-LBD solver, which has the aim to further reduce the allocation search space via the generated cuts, to prove optimality and possibly to improve the solution. We chose to turn off the cut generation within the CP solver during the bootstrap, as this tends to produce good solutions more quickly and to provide a better "warm start" for the MS-LBD solver; CP cuts are enabled once the hybrid enters the main loop, when the CP solver becomes helpful in proving optimality as well.

3.6 Experimental results

The three approaches were implemented using the state of the art solvers ILOG Cplex 10.1 and Scheduler/Solver 6.3. We ran experiments on 90 real world instances, (modeling a synthetic benchmark running on CELL BE) and 200 random graphs, generated by means of a specific instance generator designed to produce realistic Task Graphs [GLM07].

The first group of 90 instances (let this be group 1) is coming from the actual execution of multi tasking programs on a CELL BE architecture; in particular, in the chosen benchmark application the amount of data to be exchanged is so small that it can be considered negligible: in practice all data communication and execution activities for those graphs have fixed durations.

The 200 random, real-like instances are instead divided into two groups: in the first 100 graphs (group 2) durations of data communications can be considered fixed, but the impact of memory allocation on the length of the execution of each task is not negligible. Finally in the last 100 graphs (group 3) memory allocation choices have a significant impact on the duration of all activities.

Group 1 and 2 are representative of high computational intensive applications in general, like many signal processing kernels. In this scenario the overall task duration is dominated by the data computation section, while the variability induced by different memory allocations is negligible. On the other hand, group 3 is representative of more communication intensive applications. In this case, the overall task duration can be drastically affected by different memory allocations. Several video and image processing algorithms are good examples of applications which fit in this category.

Finally a last set of experiments was performed to test the impact of the cut refinement procedures. All instances feature high parallelism and complex precedence relations. The Cell configuration we used for the tests has 6 available SPEs. All experiments were performed on a Pentium IV 2GHz, 1GB ram.

3.6.1 Results for group 1

Table 3.1 show the results for group 1, containing three sets of 30 tasks graphs with variable number of tasks (15, 25, 30). Each row of the table reports results for 15 instances, listing the number of tasks and the minimum and maximum number of arcs (columns *tasks* and *arcs*). The table reports result for the Multi-Stage (refered to as *MS-LBD*), for the pure CP solver (referred to as *CP*) and for the hybrid solver presented in Section 3.5 (referred to as *HYB*). For each approach we report the average solution time (when the time limit is not exceeded, column *time*), the number of timed out instances ($> TL$) and the distance from the best solution found by any of the solvers (in case the optimal one is not found, column *diff*).

		MS-LBD			**CP**			**HYB**		
tasks	**arcs**	**time**	**> TL**	**diff.**	**time**	**> TL**	**diff.**	**time**	**> TL**	**diff.**
15	9-14	0.42	0	—	0.01	0	—	0.01	0	—
15	14-27	0.57	0	—	0.02	0	—	0.01	0	—
25	30-56	80.88	1	3%	0.10	0	—	0.13	0	—
25	56-66	274.39	2	3%	0.05	0	—	0.08	0	—
30	47-72	354.81	5	4%	1.25	2	13%	34.46	0	—
30	73-83	280.02	7	3%	0.12	0	—	0.42	0	—

Table 3.1: Results for group 1

As one can see, for this first group the CP solver is much faster than the MS-LBD based one. The hybrid has comparable performance, due to the embedded CP search step, and has very good stability, achieved by means of cut generation. The CP solver on the contrary, though extremely fast (usually a little more than the hybrid), exhibits a more erratic behavior (hence the two timed out instances for one of the 30 tasks group).

While both CP and the hybrid are often able to find the optimal solution in a fraction of second, the process is much slower for TD; nevertheless the convergence is very steady: the solver is able to provide very good solutions even when the time limit is exceeded, while for example the quality of solutions provided by CP in case of timeout is poorer. Nevertheless, the CP solver usually succeeds in improving very quickly the solution.

Overall, this first group of experiments shows that the either the proposed CP approach or the hybrid one are most likely the best choice if the impact of memory allocation choices is negligible, either completely or in part. We observed however that the relative number of arcs in the instances seems to have a strong effect on the efficiency of the solver, penalizing the CP and the hybrid solvers more than the MS-LBD one; investigating this correlation is left for future work.

3.6.2 Results for group 2

The 100 instances in group 2 have fixed duration for all data write/read activities, and execution duration depending on where computation data are allocated. Table 3.2 shows the results for this kind of instances, grouped by number of tasks, 10 instances per row. The reported statistics are the same of table 3.1.

		MS-LBD			CP			HYB		
tasks	**arcs**	**time**	**> TL**	**diff.**	**time**	**> TL**	**diff.**	**time**	**> TL**	**diff.**
10-11	6-14	0.21	0	—	0.02	0	—	0.01	0	—
12-13	8-15	1.16	0	—	0.03	0	—	0.02	0	—
14-15	12-19	1	0	—	0.03	0	—	0.03	0	—
16-17	15-22	10.89	0	—	0.11	0	—	0.95	0	—
18-19	17-24	48.92	0	—	0.05	0	—	0.08	0	—
20-21	21-29	116.1	1	0%	27.29	1	14%	36.66	0	—
22-23	21-30	69.16	1	0%	2.41	2	26%	3.69	0	—
24-25	24-35	269.57	3	22%	27.95	1	20%	49.65	0	—
26-27	27-39	88.67	7	12%	62.57	5	27%	133.62	2	36%
28-29	32-43	310	8	3%	15.03	6	16%	255.01	3	4%

Table 3.2: Results for group 2

As it can be seen, when memory allocation becomes significant, the effectiveness of the CP approach gets worse: while the average solution time is still the best among the three approaches, the number of timed out instances is higher. Once again the MS-LBD solver is the slowest one, but is still often able to find good solutions even when the time limit is exceeded (see the *diff* column); on the converse, the CP solver, while still able to find quickly a pretty good solution, has difficulties in improving it on the instances when a timeout is reported: this most likely occurs due to the importance of memory allocation choices to achieve the best makespan on such instances. The best solver as for the stability is the hybrid one, which reports the lowest number of timed out instances: this is a nice example of how a hybrid approach combining different methods can actually have a better performance than each of them.

Overall, the best choices for this type of instances are most likely the CP solver and the hybrid one, depending on what is most important for the practical problem at hand: either having a prompt solution or stability.

3.6.3 Results for group 3

For the instances in group 3, both the duration of the execution phase and that of the data transfers are sensible to memory allocation choices. Results for this group are reported in table 3.3, ordered by number of tasks and grouped by 10 instances per row. The statistics are once again the same as table 3.1.

When buffer allocation choices do matter the pure CP solver becomes definitely impractical. Note that with this kind of instances the MS-LBD approach is competitive, especially w.r.t. the average solution time; interestingly, the quality of solutions provided when the time limit is exceeded gets worse, right on the instance group where the impact of memory allocation on the overall makespan is highest: as in general the MS-LBD approach exhibits a better behavior w.r.t memory allocation, this phenomenon needs to be better investigated in the future. As for robustness, the hybrid approach once again has the best results both as for the number of timed out instances, and as for the quality of the solutions provided in case of timeout; it is worth mentioning that, within the hybrid approach, the CP solver seems to give most of its contribution in the first few iterations, getting less effective as the solution processes goes on. Improving the CP model and search strategy to cope with this issue is left for

future research.

Overall, both MS-LBD and the hybrid solvers report good results when durations of data transfers are strongly affected by memory allocation choices. The CP solver is definitely not the method of choice in this case.

		MS-LBD			CP			HYB		
tasks	**arcs**	**time**	**> TL**	**dif.**	**time**	**> TL**	**diff.**	**time**	**> TL**	**diff.**
10-11	4-12	4.01	0	—	44.60	1	7%	8.03	0	—
12-13	8-15	6.32	0	—	85.04	4	15%	6.38	0	—
14-15	10-16	5.54	0	—	0.04	6	19%	6.79	0	—
16-17	11-18	28.35	0	—	2.05	5	13%	9.22	0	—
18-19	13-20	105.50	0	—	1.95	8	10%	54.12	0	—
20-21	17-23	210.89	1	33%	170.00	9	14%	456.46	0	—
22-23	19-26	388.00	2	23%	41.00	9	28%	669.20	0	—
24-25	20-30	268.57	3	9%	—	10	18%	387.50	0	—
26-27	23-29	375.00	4	20%	—	10	21%	462.94	2	7%
28-29	25-36	528.00	5	20%	0.15	9	21%	825.13	3	10%

Table 3.3: Results for group 3

3.6.4 Refined vs. non refined cuts

As the proposed cut refinement technique requires to repeatedly solve relaxed NP-hard subproblems, we designed a last set of experiments aimed to test whether the approach is worth using for the problem at hand, in particular for the MS-LBD solver. We remind for that solver each no-good is processed by two refining procedure; in particular: 1) a custom binary search based algorithm is repeatedly invoked by 2) an explanation minimization procedure. We hence tried to solve the smallest instances in group 3 by just applying the first refinement step: this results in weaker cuts, but avoids solving a large number of relaxed subproblems.

Results for this last set of experiments are reported in table 3.4, which shows the number of SPE and MEM iterations, the solution time and the number of timed out instances for both cases. As one can see, the strength of generated cuts has a dramatic impact on the solver performance, drastically reducing the number of required SPE and MEM iterations and the solution time.

	Without 2nd ref. step				With 2nd ref. step			
ntasks	**SPE it.**	**MEM it.**	**time**	**> TL**	**SPE it.**	**MEM it.**	**time**	**> TL**
10-11	192	90	497.90	5	12	13	4.01	0
12-13	386	295	1144.21	11	17	15	6.32	0
14-15	410	539	1181.24	12	19	28	5.54	0

Table 3.4: Performance of the MS-LBD solver with and without strong cut refinement

More in detail, the behavior can be explained as follows. As all variables contained in a cut are binary, the strength of a cut can be roughly measured in terms of its size. In particular the relation is exponential: a cut containing all problem variables (i.e. a basic no-good) forbids a single solution; a cut containing all but one variable will likely remove 2 solutions (as all problem

variables are binary) and so on; more formally, a cut containing n variables will approximately clear out $\frac{1}{2^n}$ of the search space. Now, figure 3.12 shows the size distribution of cuts generated by applying both refinement procedures (figure 3.12A) or just the first one (figure 3.12B). The number of variables in the cuts is on the X axis, while the number of times a cut of that size is generated is reported in the Y axis. Ideally, we would like cuts to be as small (i.e. strong) as possible, hence the distribution peaks to be located close to the origin.

As one can see, B-cuts in particular (i.e. those injected into the SPE stage) are massively improved by the second refinement step. This largely explain the performance drop-off and gives some guidelines about how the refinement effort could be tuned: for example one could think of investigating the effect of turning on the strong refinement for B-cuts only.

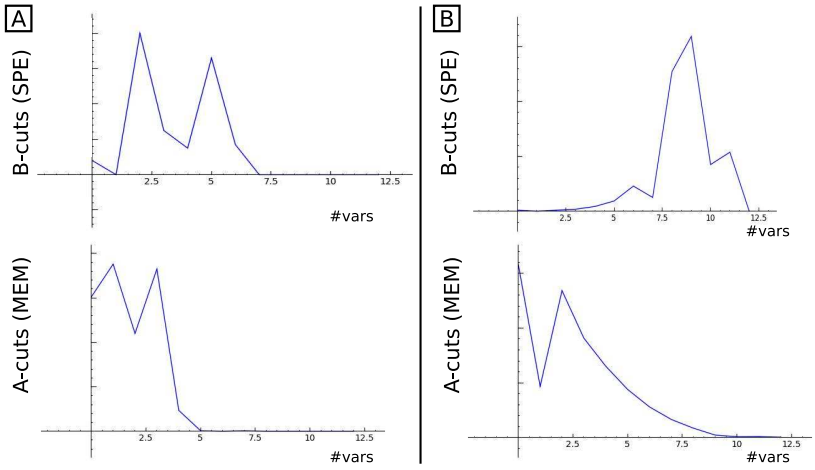


Figure 3.12: A) Size distribution of B-cuts (to SPE stage) and A-cuts (to MEM stage) after the second refinement; B) Size distribution of the same cuts after the first refinement

3.7 Conclusion and future works

In this chapter we have shown how to optimally solve allocation and scheduling of embedded streaming applications modeled as Task Graphs on the Cell BE architecture. We have proposed three different approaches, namely one based on Multi-Stage Logic based Benders' Decomposition, a Constraint Programming approach based on list scheduling ideas, and a second hybrid approach developed to combine the complementary strength of the former two.

Experimental results show how each of the approaches has different peculiarities and performance, depending on the graph features. The choice of the best approach depends therefore on the problem instance at hand and on the specific user needs: we gave qualitative considerations to guide such decision.

The proposed algorithms were devised to be a component of the CellFlow framework, developed at Micrel (University of Bologna), which has the aim to ease the implementation of efficient programs for Cell platforms. Also the

adopted execution model reflects that of CellFlow; as the framework evolves, it requires an evolution of the optimization methods as well: future developments include taking into account DMA transfers, handling multiple repetitions of an application and minimizing throughput.

We are finally planning to apply machine learning techniques to better understand how specific instance features, such as the variability of duration or the relative number of precedence constraints, influence the performance of each solver. The objective is to devise quantitative rules to automatically choose the best approach for a given instance.

Chapter 4

Hybrid methods for A&S of Conditional Task Graphs

4.1 Introduction

Conditional Task Graphs (CTG) are directed acyclic graphs whose nodes represent activities/tasks, linked by arcs representing precedence relations. Some of the activities are *branches* and are labeled with a condition; at run time, only one of the successors of a branch is chosen for execution, depending on the occurrence of a condition outcome labeling the corresponding arc. The truth or the falsity of those condition outcomes is not known a priori: this sets a challenge for any off-line design approach, which should take into account the presence of such elements of uncertainty. A natural answer to this issue is adopting a stochastic model. Each activity has a release date, a deadline and needs a resource to be executed. The problem is to find a resource assignment and a start time for each task such that the solution is feasible whatever the run time scenario is and such that the expected value of a given objective function is optimized. We take into account different objective functions: those depending on the resource allocation of tasks and those depending on the scheduling side of the problem.

CTG are ubiquitous to a number of real life problems. In compilation of computer programs [FFY01], for example, CTGs are used to explicitly take into account the presence of conditional instructions. CTG may be used also in the Business Process Management (BPM) [vdAHW03] and in workflow management [RHEvdA05], as a mean of describing operational business processes with alternative control paths.

Most relevantly in the context of this work, CTGs are used in the field of system design [XW01] to describe applications with if-then-else statements; taking into account branches in the mapping and scheduling problem allows better resource usage, and thus lower costs.

Contribution For solving the allocation and scheduling problem of CTG we need to extend the traditional constraint based techniques with two ingredients. First, to compute the expected value of the objective function, we need an efficient method for reasoning on task probabilities in polynomial time. For example, we have to compute the probability a certain task executes or not, or, more in general, the probability of a given set of scenarios with uniform features

(e.g. the same objective function value). Second, we need to extend traditional constraints to take into account the feasibility in all scenarios.

On this purpose, we define a data structure called Branch/Fork graph - BFG. We show that if the CTG satisfies a property called Control Flow Uniqueness - CFU, the above mentioned probabilities can be computed in polynomial time. CFU is a property that holds in a number of interesting applications, such as for example the compilation of computer programs, embedded system design and in structured business processes.

Outline The chapter is organized as follows: Section 4.2 presents some applications where CTG is a convenient representation of problem entities and their relations; in Section 4.3 we provide some preliminary notions on Constraint-Based Scheduling. Section 4.4 introduces the concept of Conditional Task Graphs, Control Flow Uniqueness, sample space and scenarios and defines the scheduling and allocation problem we consider. In Section 4.5 we define the data structure used for implementing efficient probabilistic reasoning, namely the Branch/Fork Graph and related algorithms. In Section 4.6 we use these algorithms for efficiently computing the expected value of three objective function types, while in Section 4.7 we exploit the BFG for implementing the conditional variant of the timetable global constraint. Section 4.8 discusses related work and Section 4.9 shows experimental results and a comparison with a scenario based approach.

Publications Part of the work at the base of this chapter has been published to international conferences in [LM06, LM07]; an extended version is to appear on *Artificial Intelligence* with title Allocation and Scheduling of Conditional Task Graphs.

4.2 Applications of CTGs

Conditional Task Graphs can be used as a suitable data structure for representing activities and their temporal relations in many real life applications. In these scenarios, CTG allocation and scheduling becomes a central issue.

In compilation of computer programs [FFY01], for example, CTGs are used to explicitly take into account the presence of conditional instructions. For instance, Figure 4.1 shows a simple example of pseudo-code and a natural translation into a CTG; here each node corresponds to an instruction and each branch node to an "if" test; branch arcs are label with the outcome they represent. In this case, probabilities of condition outcomes can be derived from code profiling. Clearly, computer programs may contain loops that are not treated in CTGs, but modern compilers adopt the *loop unrolling* [KA01] technique that can be used here for obtaining cycle free task graphs.

Similarly, in the field of embedded system design [XW01] a common model to describe a parallel application is the task graph. The task graph has a structure similar to a data flow graph, except that the tasks in a task graph represent larger units of functionality. However, a task graph model that has no control dependency information can only capture data dependency in the system specification. Recently, some researchers in the co-synthesis domain have tried to use conditional task graph to capture both data dependencies and control

```
if a = 0 then
  return error
else
  if b < 0 then
    b = b + 1
  end if
end if
return a · b
```

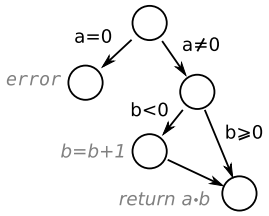


Figure 4.1: Some pseudo-code and a its translation into a CTG

dependencies of the system specification [WAHE03b, KW03]. Once a hardware platform and an application is given, to design a system amounts to allocate platform resources to processes and to compute a schedule; in this context, taking into account branches allows better resource usage, and thus lower costs. However, the presence of probabilities makes the problem extremely complex since the real time and quality of service constraints should be satisfied for any execution scenario. Embedded system design applications will be used in this chapter to experimentally evaluate the performance and quality of our approach.

CTG appear also in Business Process Management (BPM) [vdAHW03] and in workflow management [RHEvdA05] as a mean of describing operational business processes with alternative control paths. Workflows are instances of workflow models, that are representations of real-world business processes [Wes07]. Basically workflow models consist of activities and the ordering amongst them. They can serve different purposes: they can be employed for documentation of business processes or can be used as input to a Workflow Management System that allows their machine-aided execution.

One of the most widely used systems for representing business processes is BPEL [ftAoSISO]. BPEL is a graph-structured language and allows to define a workflow model using nodes and edges. The logic of decisions and branching is expressed through transition conditions and join conditions. Transition conditions and join conditions are both Boolean expressions. As soon as an activity is completed, the transition conditions on their outgoing links are evaluated. The result is set as the *status of the link*, which is true or false. Afterwards, the target of each link is visited. If the status of all incoming links is defined, the join condition of the activity is evaluated. If the join condition evaluates to false, the activity is called *dead* and the status of all its outgoing links is set to false. If the join condition evaluates to true, the activity is executed and the status of each outgoing link is evaluated. CTGs behave exactly in the same fashion and can be used to model BPEL workflow models. In addition CTG provide probabilities on branches. Such numbers, along with task durations and resource consumption and availability can be extracted from process event logs. The CTG allocation and scheduling proposed in this chapter can be used in the context of workflow management as a mean to predict the completion time of the running instances, as done in [vdASS], or for scheduling tasks to obtain the minimal expected completion time.

4.3 Preliminaries on Constraint-Based Scheduling

In this chapter we show how to extend constraint-based scheduling techniques for dealing with probabilistic information and with conditional task graphs. For some preliminaries on Constraint-Based Scheduling, see Section 2.3; here, we just recall that Scheduling problems over a set of activities are classically modeled in CP by introducing for every activity three variables representing the start time (`S`), end time (`E`) and duration (`D`). In this context a solution (or schedule) is an assignment of all `S` and `E` variables. "Start", "end" and "duration" variables must satisfy the constraint $\texttt{E} = \texttt{S} + \texttt{D}$.

Activities require a certain amount of resources for their execution. We consider in this chapter both unary resources and discrete (or cumulative) resources. Unary resources have capacity equal to one and two tasks using the same unary resource cannot overlap in time, while cumulative resources have finite capacity that cannot be exceeded at any point in time.

Scheduling problems often involve precedence relations and alternative resources; precedence relations are modeled by means of constraints between the start and end variables of different activities, while special resource constraints guarantee the capacity of each resource is never exceeded in the schedule; a number of different propagation algorithms for temporal and resource constraints [BP00, Lab03] enable an effective reduction of the search space. Finally, special scheduling oriented search strategies [BLPN01] have been devised to efficiently find consistent schedules or to prove infeasibility.

4.4 Problem description

The problem we consider is the scheduling of Conditional Task Graphs (CTG) in presence of unary and cumulative alternative resources. In the following, we introduce the definitions needed in the rest of the chapter. In Section 4.4.1 we provide some notions about Conditional Task Graphs, Section 4.4.2 concerns Control Flow Uniqueness, a CTG property that enables the definition of polynomial time CTG algorithms, Section 4.4.3 introduces the concept of sample space and scenarios while Section 4.4.4 describes the scheduling and allocation problem considered in the chapter.

4.4.1 Conditional Task Graph

A CTG is a directed acyclic graph, where nodes are partitioned into branch and fork nodes. Branches in the execution flow are labeled with a condition. Arcs rooted at branch nodes are labeled with condition outcomes, representing what should be true in order to traverse that arc at execution time, and their probability. Intuitively, fork nodes originate parallel activities, while branch nodes have mutually exclusive outgoing arcs.

More formally:

Definition 5 (Conditional Task Graph)**.** *A CTG is a directed acyclic graph that consists of a tuple $\langle T, A, C, P \rangle$, where*

- *$T = T_B \cup T_F$ is a set of nodes; $t_i \in T_B$ is called a branch node, while $t_i \in T_F$ is a fork node. T_B and T_F partition set T, i.e., $T_B \cap T_F = \emptyset$. Also, if $T_B = \emptyset$ the graph is a deterministic task graph.*

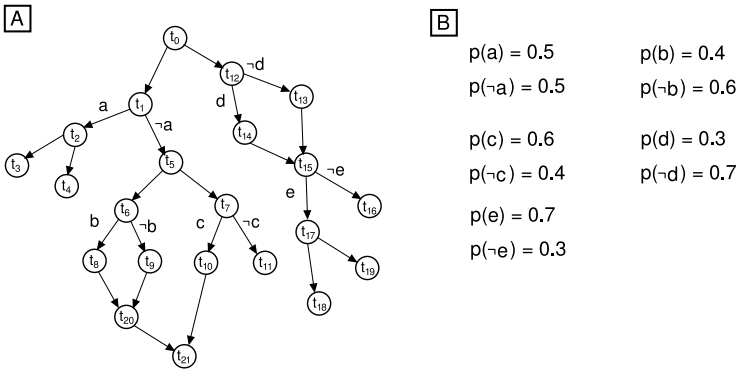


Figure 4.2: A: Example of CTG; B: Probabilities of condition outcomes

- *A is a set of arcs as ordered pairs $a_h = (t_i, t_j)$.*
- *C is a set of pairs $\langle t_i, c_i \rangle$ for each branch node $t_i \in T_B$. c_i is the condition labeling the node.*
- *P is a set of triples $\langle a_h, Out, Prob \rangle$ each one labeling an arc $a_h = (t_i, t_j)$ rooted in a branch node t_i. $Out = Out_{ij}$ is a possible outcome of condition c_i labeling node t_i, and $Prob = p_{ij}$ is the probability that Out_{ij} is true ($p_{ij} \in [0, 1]$).*

The CTG always contains a single root node (with no incoming arcs) that is connected (either directly or indirectly) to each other node in the graph. For each branch node $t_i \in T_B$ with condition c_i every outgoing arc (t_i, t_j) is labeled with one distinct outcome Out_{ij} such that $\sum_{(t_i,t_j)} p_{ij} = 1$.

Intuitively, at run time, only a subgraph of the CTG will *execute*, depending on the branch node condition outcomes. Each time a branch node is executed, its condition is evaluated and only one of its outgoing arcs is evaluated to true. In Figure 4.2A if condition a is true at run time, then arc (t_1, t_2) status is true and node t_2 executes, while arc (t_1, t_5) status is false and node t_5 does not execute. Without loss of generality, all examples throughout this chapter target graphs where every condition, say a, has exactly two outcomes, $a = true$ or $a = false$. However, we can model multiple alternative outcomes, say $a = 1$ or $a = 2$ or $a = 3$ provided that they are mutually exclusive (i.e., only one of them is true at run time).

In Figure 4.2A t_0 is the root node and it is a fork node that always executes at run time. Arcs (t_0, t_1) and (t_0, t_{12}) rooted in an executing fork node are always evaluated to true. Node t_1 is a branch node, labeled with condition a. With an abuse of notation we have omitted the condition in the node and we have labeled arc (t_1, t_2) with the outcome a meaning $a = true$ and (t_1, t_5) with $\neg a$ meaning $a = false$. The probability of $a = true$ is 0.5 and the probability of $a = false$ is also 0.5.

Let $A^+(t_i)$ be the set of outgoing arcs of node t_i, that is $A^+(t_i) = \{a_h \in A \mid a_h = (t_i, t_j)\}$; similarly let $A^-(t_i)$ be the set of ingoing arcs of node t_i, i.e., $A^-(t_i) = \{a_h \in A \mid a_h = (t_j, t_i)\}$. Then t_i is a said to be a *root node* if

$|A^-(t_i)| = 0$ (t_i has no ingoing arc), t_i is a *tail node* if $|A^+(t_i)| = 0$ (t_i has no outgoing arc).

Without loss of generality, we restrict our attention to CTGs such that every node t_i with two or more ingoing arcs ($|A^+(t_i)| > 1$) is either an *and-node* or an *or-node*. The concept of and/or-nodes, that of executing node and the arc status can be formalized in a recursive fashion:

Definition 6. (Run Time Execution of Nodes, Arc Status, And/or Node)

- *The root node always executes.*
- *The status of arc (t_i, t_j) rooted in a fork node $t_i \in T_F$ is true if node t_i executes.*
- *The status of arc (t_i, t_j) rooted in a branch node $t_i \in T_B$ is true if node t_i executes and the outcome Out_{ij} labeling the arc is true.*
- *A node t_i with $|A^-(t_i)| > 1$ is an or-node if either none or only one of the ingoing arcs status can be true at run-time.*
- *An or-node t_i executes if any arc in $A^+(t_i)$ has a status equal to true.*
- *A node t_i with $|A^-(t_i)| \geq 1$ is an and-node if it is possible that all the ingoing arcs status are true at run time.*
- *An and-node executes if all arcs in $A^-(t_i)$ have a status equal to true.*

Note that the definition is recursive: deciding whether a node t_i with $|A^+(t_i)|$ is an and/or-node depends on whether its predecessors can execute, and deciding whether a node can execute requires to know whether the predecessors are and/or-nodes. The system is however consistent as both concepts only depend on information concerning the *predecessors* of the considered node t_i; as the root node by definition always executes and the CTG contains no cycle, both the concept of and/or-node and that of executing node/status of an arc are well defined.

Observe that ingoing arcs in an or-node are always mutually exclusive; mixed and/or-nodes are not allowed but can be modeled by combining pure (possibly fake) and-nodes and or-nodes. Note also that nodes with a single ingoing arc are classified as and-nodes. Again, for the sake of simplicity in the chapter we have used examples with only two ingoing arcs in and/or-nodes, but the presented results are valid in general and apply for any number of ingoing arcs.

In Figure 4.2A t_{15} is an or-node since at run time either the status of (t_{14}, t_{15}) or the one of (t_{13}, t_{15}) is true (depending on the outcome of condition d); t_{21} is an and-node since, if condition a has outcome *false*, arc (t_{20}, t_{21}) is true and arc (t_{10}, t_{21}) status is true if the outcome $c = true$ holds. Therefore, it is possible that both incoming arcs are true at run time. t_{15} executes if any of the ingoing arcs status is true, while t_{21} executes only if both the ingoing arc status evaluate to true.

4.4.1.1 Activation event of a node

For modeling purposes, it is useful to express the combination of outcomes determining the execution of a node as a compact expression. As outcomes are logical entities (either they are *true* or *false* at run time) it is convenient to formulate such combination of outcomes as a logical expression, referred here to as *activation event*.

The activation event of a node t_i is denoted as $\varepsilon(t_i)$ and can be obtained in a recursive fashion, similarly to definition of executing node and and/or-nodes. In practice:

$$\varepsilon(t_i) = \begin{cases} \texttt{true} & \text{if } |A^-(t_i)| = 0 \text{ (t_i is the root node)} \\ \bigvee_{a_h=(t_j,t_i)\in A^-(t_i)} \varepsilon(a_h) & \text{if } t_i \text{ is an or-node} \\ \bigwedge_{a_h=(t_j,t_i)\in A^-(t_i)} \varepsilon(a_h) & \text{if } t_i \text{ is an and-node or if } |A^-(t_i)| = 1 \end{cases}$$

and $\varepsilon(a_h)$ is the activation event of an arc a_h and is defined as follows:

$$\varepsilon(a_h = (t_i, t_j)) = \begin{cases} \varepsilon(t_i) & \text{if } t_i \text{ is a fork} \\ \varepsilon(t_i) \wedge Out_{ij} & \text{if } t_i \text{ is a branch} \end{cases}$$

For example, the activation event of task t_2 in Figure 4.2A is $\varepsilon(t_2) = a$, while the activation event of t_{21} is $\varepsilon(t_{21}) = ((\neg a \wedge b) \vee (\neg a \wedge \neg b)) \wedge (\neg a \wedge c) = (\neg a \wedge b \wedge c) \vee (\neg a \wedge \neg b \wedge c) = \neg a \wedge c \wedge (b \vee \neg b) = \neg a \wedge c$.

In general we need to express activation events in Disjunctive Normal Form (DNF), that is a disjunction of one or more conjunctions of one or more literals.

4.4.2 Control Flow Uniqueness

Even if many of the definitions and algorithms we present in this chapter work in the general case, we are interested in specific CTG satisfying a property called *Control Flow Uniqueness* (CFU)[1]. Intuitively, CFU is satisfied if no node t_i in the graph requires for its execution the occurrence of two outcomes found on separated paths from the root to t_i. More formally:

Definition 7 (Control Flow Uniqueness)**.** *A CTG satisfies the CFU if for each and-node t_i, there is a single arc $a \in A^-(t_i)$ such that, for all other incoming arcs $a' \in A^-(t_i)$:*

$$\text{status of arc } a \text{ is true} \Rightarrow \text{status of arc } a' \text{ is true}$$

where the symbol "$\Rightarrow$" denotes the logical implication. Intuitively a single ingoing arc $a \in A^-(t_i)$ is *logically* responsible of the execution of the and-node t_i; if the status of such arc becomes true at some point of time, the status of all other ingoing arcs will become (or have become) true as well. Note the actual run time execution of t_i only occurs once all ingoing arcs have become true. As a consequence there is also only one path from the root to the and-node that is *logically* responsible for the execution of that node. More formally:

Corollary: If a CTG satisfies CFU, then for each task t_i each conjunction of condition outcomes in its activation event $\varepsilon(t_i)$ (in DNF) can be derived by collecting condition outcomes following a single path from the root node to t_i.

For example in Figure 4.3A, task t_8 is an and-node; its activation event is $\varepsilon(t_8) = (a \wedge b) \vee (\neg a \wedge b) = b$, thus CFU holds. Conversely, in Figure 4.3B both $\neg a$ and b are strictly required for the execution of t_7 and they do not appear in sequence along any path from the root to t_7; hence CFU is not satisfied.

[1] In the rest of the chapter, we will explicitly underline which algorithms/properties need the CFU.

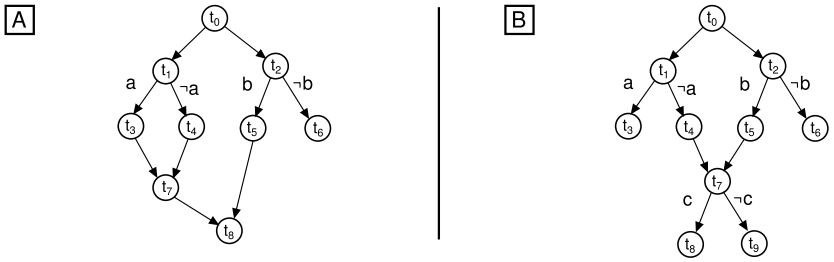


Figure 4.3: A: a CTG which satisfies CFU — B: a CTG which does not satisfy CFU

In many practical cases CFU is not a restrictive assumption; for example, when the graph results from the parsing of a computer program written in a high level language (such as C++, Java, C#) CFU is quite naturally enforced by the scope rules of the language, or can be easily made valid by proper modeling. For example, consider again the pseudo-code in Figure 4.1. One can easily check that 1) CFU is satisfied and 2) there exist no simple translation of the pseudo code violating CFU as each conditional instruction (if) has a collector node (end if).

Moreover, in some application domains (e.g., business process management, embedded system design), a common assumption is to consider so called *structured* graphs, i.e., graphs with a single collector node for each conditional branch. In this case, the CFU is trivially satisfied. Note how a structured graph cannot model early exits (e.g. in case of error), as the one reported in Figure 4.1.

4.4.3 Sample Space and Scenarios

On top of a CTG we define the *sample space* S.

Definition 8 (Sample Space)**.** *The sample space of a CTG is the set of events occurring during all possible executions of a CTG, each event being a set of condition outcomes.*

For example, the sample space defined on top of the CTG in Figure 4.2A can be computed by enumerating all possible graph executions and contains 20 events. Again using an abuse of notation we refer to the outcome $a = true$ with a and to the outcome $a = false$ with $\neg a$. Also, for sake of clarity we have removed the logical conjunctions among conditions: the term $a \wedge b \wedge e$ has been simplified in abe. Therefore, the sample space associated to the CTG in Figure 4.2A is the following.

$$\begin{aligned}\mathcal{S} \;=\; & \{ade, ad\neg e, a\neg de, a\neg d\neg e, \neg abcde, \neg abc\neg de, \neg abcd\neg e,\\ & \neg abc\neg d\neg e, \neg a\neg bcde, \neg a\neg bc\neg de, \neg a\neg bcd\neg e, \neg a\neg bc\neg d\neg e,\\ & \neg ab\neg cde, \neg ab\neg c\neg de, \neg ab\neg cd\neg e, \neg ab\neg c\neg d\neg e, \neg a\neg b\neg cde,\\ & \neg a\neg b\neg c\neg de, \neg a\neg b\neg cd\neg e, \neg a\neg b\neg c\neg d\neg e\}\end{aligned}$$

We need now to associate a probability to each element of the sample space.

$$\forall s \in S \;\; p(s) = \prod_{Out_{ij} \in s} p_{ij}$$

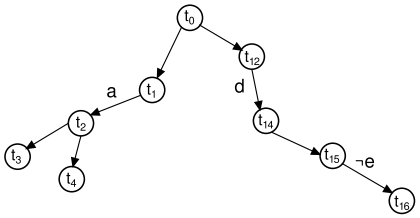


Figure 4.4: The deterministic task graph associated with the run time scenario $a = true$, $d = true$ and $e = false$, for the CTG of Figure 4.2

.

For instance, with reference to Figure 4.2A, the probability of event *ade* is $0.5 * 0.3 * 0.7 = 0.105$.

Each event in the sample space of the CTG is associated to a scenario. A scenario corresponds to a deterministic task graph containing the set of nodes and arcs that are active in the scenario. We have to define how to build such a task graph. This task graph is defined recursively.

Definition 9 (Task Graph for a Scenario)**.** *Given a CTG=$\langle T, A, C, P\rangle$, and an event $s \in S$ the deterministic task graph TG(s) associated with s is defined as follows:*

- *The CTG root node always belongs to the TG(s)*
- *A CTG arc (t_i, t_j) belongs to TG(s) if either*
  - *t_i is a fork node and t_i belongs to TG(s) or*
  - *t_i is a branch node, $Out_{ij} \in s$ and t_i belongs to TG(s).*
- *A CTG node t_i belongs to TG(s) if it is an and-node and all arcs $a_h \in A^-(t_i)$ are in TG(s) or if it is an or-node and any arc $a_h \in A^-(t_i)$ is in TG(s)*

TG(s) is called scenario associated with the event s.

With an abuse of notation, in the following we refer to the event s also as scenario. The deterministic task graph derived from the CTG in Figure 4.2A associated to the run time scenario $a = true$, $d = true$ and $e = false$ (or equivalently $ad\neg e$) is depicted in Figure 4.4.

Often we are interested in identifying a set of scenarios, such as for instance all scenarios where a given task executes. We have to start by identifying the events associated to scenarios where task t_i executes. This set is defined as $S_i = \{s \in S | t_i \in TG(s)\}$. The probability that a node t_i executes (let this be $p(t_i)$) can then be computed easily: $p(t_i) = \sum_{s \in S_i} p(s)$. For example, let us consider task t_2 in Figure 4.2A; then $S(t_2) = \{ade, ad\neg e, a\neg de,\ a\neg d\neg e\}$ and $p(t_2) = 0.5 \cdot 0.3 \cdot 0.7 + 0.5 \cdot 0.3 \cdot 0.3 + 0.5 \cdot 0.7 \cdot 0.7 + 0.5 \cdot 0.7 \cdot 0.3 = 0.5$. Alternatively, the probability $p(t_i)$ can be computed starting from the activation event; for example, $\varepsilon(t_2) = a$, hence $p(t_2) = p(a) = 0.5$, or $\varepsilon(t_8) = \neg a \wedge b$, hence $p(t_8) = p(\neg a) \cdot p(b) = 0.5 \cdot 0.4 = 0.2$.

For modeling purposes, we also define for each task an activation function $f_{t_i}(s)$; this is a stochastic function $f_{t_i} : S \rightarrow \{0, 1\}$ such that

$$f_{t_i}(s) = \begin{cases} 1 & \text{if } t_i \in TG(s) \\ 0 & \text{otherwise} \end{cases}$$

Finally, we need to define the concept of mutually exclusive tasks:

Definition 10 (Mutually Exclusive Tasks)**.** *Two tasks t_i and t_j are said to be mutually exclusive (i.e. "mutex (t_i, t_j)") iff there is no scenario $TG(s)$ where both tasks execute, i.e., where $t_i \in TG(s)$ and $t_j \in TG(s)$ or equivalently, $f_{t_i}(s) = f_{t_j}(s) = 1$.*

4.4.4 A&S on CTGs: Problem definition and model

The allocation and scheduling problem we face is defined on a conditional task graph whose nodes are interpreted as activities (also referred to as tasks) and arcs are precedence relations between pairs of activities. The CTG is annotated with a number of activity features, such as duration, due and release dates, alternative resource sets and resource consumption. We have to schedule tasks and assign them resources from the alternative resource set such that all temporal and resource constraints in any run time scenario are satisfied and the expected value of a given objective function is optimized. More formally:

Definition 11 (CTG A&S Problem)**.** *An instance of the CTG allocation and scheduling problem is a tuple $\langle CTG, Obj, Dur, Rel, Due, ResSet, ResCons \rangle$.*

In the $CTG = \langle T, A, C, P \rangle$ T represents the set of non preemptive tasks to be allocated and scheduled, A represents the set of precedence constraints between pairs of tasks, C is the set of conditions labeling the nodes and P is the set of outcomes and probabilities labeling the arcs. Obj is the objective function. Dur, Rel, Due, $ResSet$ and $ResCons$ are functions mapping each task in T to the respective duration, release date, due date, alternative resource set and resource consumption. Given a task $t_i \in T$, its duration is referred to as Dur_i, its release date as Rel_i, its due date as Due_i, its alternative resource set to as $ResSet_i$ and its resource consumption $ResCons_i$; with the exception of $ResSet$ all mentioned functions have values in $\mathbb{N}^+$.

For sake of simplicity, we assume each task t_i needs a single resource taken from its $ResSet_i$ for its execution; however, the results presented in this chapter can be easily extended to tasks requiring more than one resource. More in detail, suppose each task requires up to m of *types* of resource; provided separate *ResSet* and *ResCons* functions, all the presented results apply to each type in a straightforward fashion.

4.4.4.1 Modeling tasks and temporal constraints

As far as the model is concerned, each node in the CTG corresponds to a task (also called activity). Similarly to constraint-based scheduling, a task t_i is associated to a time interval $[\mathtt{S}_i, \mathtt{E}_i)$ where $\mathtt{S}_i$ is a decision variable denoting the starting time of the task and $\mathtt{E}_i$ is a variable linked to $\mathtt{S}_i$ as follows: $\mathtt{E}_i = \mathtt{S}_i + \mathtt{D}_i$. Depending on the problem, the duration may be known in advance or may be a decision variable. In this chapter we consider fixed, known in advance, durations.

The start time variable of a task t_i has a domain of possible values ranging from the earliest start time $EST(t_i)$ to the latest start time $LST(t_i)$. Similarly, the end time variable has a domain ranging from earliest to the latest end times, referred to as $EET(t_i)$ and $LET(t_i)$. Initially $\mathtt{S}(t_i)$ and $\mathtt{E}_i$ range on the whole schedule horizon (from the time point 0 to the end of horizon $[0..eoh]$).

Each arc (t_i, t_j) in the CTG corresponds to a precedence constraint on decision variables and has the form $\mathtt{S}(t_i) + \mathtt{D}_i \leq \mathtt{S}(t_j)$. Due dates and release dates translate to constraints $\mathtt{S}(t_i) + \mathtt{D}_i \leq Due_i$ and $\mathtt{S}(t_i) \geq Rel_i$.

4.4.4.2 Modeling alternative resources

Beside start time of activities, an additional decision variable $res(t_i)$ represents the resource assigned to the activity t_i. The domain of possible values of $res(t_i)$ is $ResSet_i$.

Resources in the problem can be either discrete or unary. Discrete resources (also referred to as cumulative resources) have a known maximal capacity. A certain amount of resource $ResCons_i$ is consumed by activity t_i assigned to a discrete resource $res(t_i)$ at the start time of activity t_i and the same quantity is released at its end time. A unary resource has a unit capacity. It imposes that all activities requiring the same unary resource are totally ordered. Given a resource r_k its capacity is referred to as C_k.

4.4.4.3 Classical objective function types

Depending on the problem, the objective function may depend on the temporal allocation of the activities, i.e., decisions on variables $\mathtt{S}$ (or equivalently variables $\mathtt{E}$ if the duration is fixed), or on the resource assignments, i.e., decisions on variables res.

In constraint-based scheduling, a widely studied objective function is the makespan, i.e., the length of the whole schedule. It is the maximum value of the $\mathtt{E}_i$ variables.

$$Obj_1 = \max_{t_i \in T} \mathtt{E}_i \tag{4.1}$$

Another example of objective function is the sum of costs of resource assignments to single tasks. As an example, consider the case where running a task on a given resource consumes a certain amount of energy or power.

$$Obj_2 = \sum_{t_i \in T} cost(t_i, res(t_i)) \tag{4.2}$$

In the hypothesis to have a cost matrix where each element c_{ij} is the cost of assigning resource j to task t_i, $res(t_i) = j \leftrightarrow cost(t_i, res(t_i)) = c_{ij}$.

A third example that we will consider in this chapter still depends on resource assignment, but on pairs of assignments.

$$Obj_3 = \sum_{a_h = (t_i, t_j) \in A} cost(t_i, res(t_i), t_j, res(t_j)) \tag{4.3}$$

For instance, suppose that arcs represent communication activities between two tasks. If t_i and t_j are assigned to the same resource, their communication cost

is zero, while if they are assigned to different resources, the communication cost increases. Suppose we have a vector where each element c_k is the cost of arc $arc_k = (t_i, t_j)$ if t_i and t_j are assigned to different resources.

$$cost(t_i, res(t_i), t_j, res(t_j)) = \begin{cases} c_k \text{ if } res(t_i) \neq res(t_j) \\ 0 \text{ otherwise} \end{cases}$$

Other objective functions could be considered as well. For example there could be a cost for having at least one task using a certain resource (e.g. to turn the resource "on"). In this case the cost is associated to the execution of *sets* of tasks; the objective function can be considered a generalization of Obj_3 and is dealt with by means of the same techniques.

Clearly, having probabilities and conditional branches, we have to take into account all possible run time scenarios and optimize the expected value of the objective function. Therefore, given the objective function Obj and a scenario s, we refer to the objective function computed in the scenario s to as $Obj^{(s)}$. For example, in $Obj_1^{(s)}$, the maximum of end variables is restricted only to those tasks that are active in the scenario s (those belonging to TG(s)).

$$Obj_1^{(s)} = \max_{t_i \in TG(s)} \mathtt{E}_i = \max_{t_i \in T} f_{t_i}(s)\mathtt{E}_i$$

Similarly $Obj_2^{(s)} = \sum_{t_i \in TG(s)} cost(t_i) = \sum_{t_i \in T} f_{t_i}(s) cost(t_i, res(t_i))$

and

$$Obj_3^{(s)} = \sum_{a_h=(t_i,t_j) \in TG(s)} cost(res(t_i), res(t_j)) =$$
$$= \sum_{a_h=(t_i,t_j) \in A} f_{t_i}(s) f_{t_j}(s) cost(t_i, res(t_i), t_j, res(t_j))$$

Finally, by recalling the definition of "expected value" in probability theory, we state that:

Definition 12 (Expected Objective)**.** *The expected value of a given objective function Obj is a weighted sum of the $Obj^{(s)}$ weighted by scenario probabilities*

$$E(Obj) = \sum_{s \in S} p(s) Obj^{(s)}$$

4.4.4.4 Solution of the CTG Scheduling Problem

The solution of the CTG scheduling problem can be given either in terms of a scheduling and allocation table [WAHE03a] where each task is assigned to a different resource and a different start time, depending on the scenario, or as a unique allocation schedule where each task is assigned a single resource and a single start time independently on the run time scenario. The first solution is much more precise and able to better optimize the expected value of the objective function. Unfortunately, the size of such a table grows exponentially as the number of scenarios increases, making the problem of computing an optimal scheduling table P-SPACE complete (it is analogous to finding an optimal policy in stochastic constraint programming [Wal02]). We therefore chose to provide

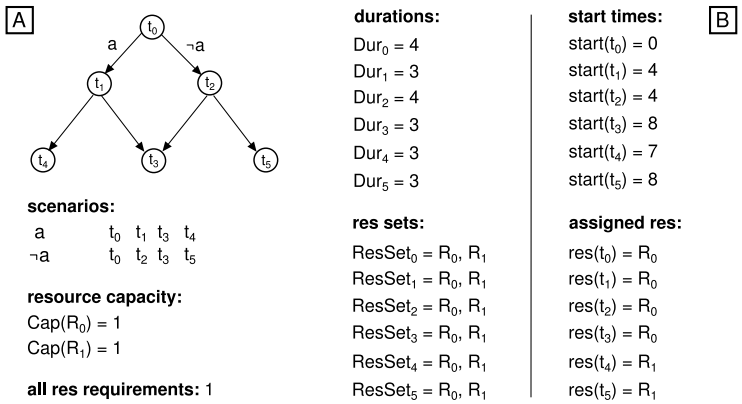


Figure 4.5: A: a CTG scheduling problem; B: a possible solution

a more compact solution, where each task is assigned a unique resource and a unique start time feasible in every possible run time scenario. this keeps the problem NP-hard. This choice goes in the line of the notion of strong controllability defined in [VF99] for temporal constraint networks with uncertainty; in particular, a network is said to be strongly controllable if there exists a single control sequence satisfying the temporal constraints for every scenario. In addition, for some classes of problems such as compilation of computer programs, this is the only kind of solution which can be actually implemented and executed [SK03]. More formally, we provide the following definition.

Definition 13 (Solution to a CTG A&S Problem)**.** *The solution of the allocation and scheduling problem* $\langle\langle T, A, C, P\rangle, Obj, Dur, Rel, Due, ResSet, ResCons\rangle$ *is an assignment to each task* $t_i \in T$ *of a start time* $\mathtt{S}_i \in [0..eoh]$ *and of a resource* $res(t_i) \in ResSet_i$ *such that*

1. $\forall t_i \in T \quad \mathtt{S}_i \geq Rel_i$
2. $\forall t_i \in T \quad \mathtt{S}_i + Dur_i \leq Due_i$
3. $\forall (t_i, t_j) \in A \quad \mathtt{S}_i + Dur_i \leq \mathtt{S}_j$
4. $\forall t = 0 \ldots eoh \; \forall s \in S \quad \forall r_k \in R \in \bigcup_{t_i \in TG(s)} ResSet_i :$

$$\sum_{\substack{t_i \in TG(s) : \\ res(t_i) = r_k \\ \mathtt{S}_i \leq t < \mathtt{E}_i}} ResCons_i \leq C_k$$

Constraints (1) and (2) ensure each task is executed within its release date and due date. Constraints (3) enforce precedence constraints, constraints (4) enforce resource capacity restrictions in all scenarios and at every time instant t on the time line. A solution is *optimal* if $E(Obj)$ is minimized (resp. maximized).

For example, in Figure 4.5A we show a small CTG scheduling problem and in Figure 4.5B the corresponding solution. Note that all tasks have a unique start time and a unique resource assignment independent from the scenario, but feasible in all scenarios. For instance, tasks t_1 and t_2 are mutually exclusive, as they cannot appear in the same scenario. Therefore, although they use the same unary resource, their execution can overlap in time.

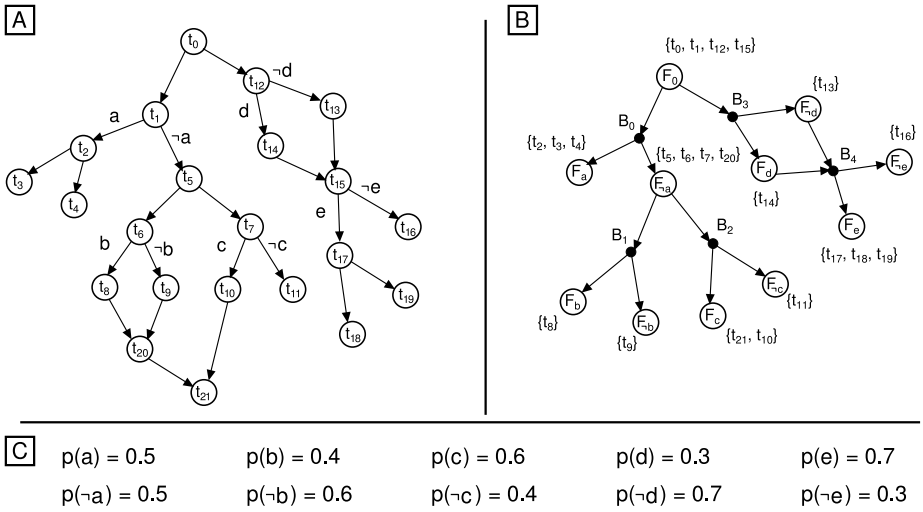


Figure 4.6: A: The CTG from Figure 4.2; B: The associated BFG; C: probabilities of condition outcomes

4.5 Probabilistic Reasoning

The model presented in Section 4.4 cannot be solved via traditional constraint-based scheduling techniques. In fact, there are two aspects that require probabilistic reasoning and should be handled efficiently: the resource constraints to be enforced in all scenarios and the computation of the expected value of the objective function (a weighted sum on all scenarios). In both cases, in principle, we should be able to enumerate all possible scenarios, whose number is exponential. Thus, we need a more efficient way to cope with this expression.

One contribution of this chapter is the definition of a data structure, called Branch/Fork Graph (BFG), that compactly represents all scenarios, and one parametric polynomial time algorithm working on the BFG that enables efficient probabilistic reasoning. For instance, we instantiate the parametric algorithm for the computation of the probability of a given set of scenarios, such as the probability of all scenarios where a given set of tasks execute (resp. do not execute).

4.5.1 Branch/Fork Graph

A Branch/Fork Graph (BFG) intuitively represents the skeleton of all possible control flows and compactly encodes all scenarios of the corresponding CTG; for example Figure 4.6B shows the BFG associated to the CTG from Figure 4.2A (reported again in Figure 4.6A for simplicity).

A BFG is an acyclic directed graph. Nodes are either branch nodes (B-nodes, dots in Figure 4.6B) or fork nodes (F nodes, circles in Figure 4.6B). There is a branch node in the BFG for each branch node in the CTG. F-nodes instead represent sets of events and group CTG nodes executing at all such events. For example in Figure 4.6B F_a groups together nodes t_2, t_3 and t_4 as they all execute in all scenarios where $a = true$. More formally:

Definition 14 (Branch/Fork Graph)**.** *A Branch/Fork graph is a directed, acyclic graph associated to a CTG* $= \langle T = T_B \cup T_F, A, C, P \rangle$ *with two types of nodes, referred to as B-nodes and F-nodes.*

BFG nodes *satisfy the following conditions:*

1. *The graph has a B-node* B_i *for every branch node* $t \in T_B$ *in the associated CTG.*
2. *Let* S *be the sample space of the CTG; then the BFG has an F-node* F_i *for every subset of events* $\sigma \in 2^S$*, unless:*
   (a) $\nexists t_i \in T$ *such that* $\forall s \in \sigma : \ t_i \in TG(s)$.
   (b) $\exists c_k \in C$ *such that (I) more than one outcome of* c_k *appear in scenarios in* σ *and (II) not all the outcomes of* c_k *appear in* σ.
   (c) $\exists \sigma' \in 2^S$ *such that (a) and (b) are not satisfied (i.e. hence, an F-node would be built) and* $\sigma \subset \sigma'$.

The CTG branch node corresponding to a B-node B_i *is denoted as* $t(B_i)$*. The F-nodes are said to "represent" the set of events* σ *they correspond to, denoted as* $\sigma(F_i)$*. The set of CTG nodes* t_i *such that* $\forall s \in \sigma(F_i)$, $t_i \in TG(s)$ *is said to be "mapped on"* F_i *and is denoted as* $t(F_i)$[2]

BFG arcs *satisfy the following conditions:*

1. *An F-node* F_i *is connected to a B-node* B_j *by an arc* (F_i, B_j) *if* $t(B_j) \in t(F_i)$
2. *A B-node* B_i *is connected by means of an arc labeled with the outcome* $Out_{t(B_i),k}$ *to an F-node* F_j *such that* $t_k \in t(F_j)$
3. *An F-node* F_i *is connected to an F-node* F_j *such that no path from* F_i *to* F_j *already exists and:*
   (a) $\sigma(F_j) \subset \sigma(F_i)$
   (b) *There exists no F-node* F_k *such that* $\sigma(F_j) \subset \sigma(F_k) \subset \sigma(F_i)$

Condition on BFG nodes (1) tells the BFG has one B-node for each branch in the associated CTG. Following condition (2), each F-node models a subsets of events $\sigma \in 2^S$; there is however no need to model a subset σ if any of the three conditions (2abc) holds. In particular:

2a) there is no need to model sets of events σ such that no task in the graph would be mapped to the resulting F-node; such sets of events are of no interest, as the ultimate purpose of the BFG is to support reasoning about task executions and their probability.

2b) there is no need to model a set of events σ, if two or more outcomes of a condition c_k appear in σ and still there is some outcome of c_k not in σ. In fact if two or more (but not all) outcomes of c_k are in σ, then σ still *depends* on c_k and one could model this by using several F-nodes, each one referring to a single outcome. If however all outcomes of c_k are in σ, then σ is *independent* on c_k.

2c) provided neither condition (2a) nor (2b) holds, there is still no need to build an F-node if there exist a larger set of events σ' such that neither condition (2a) nor (2b) holds as well. In practice, due to condition (2c) F-nodes always model maximal sets of events.

[2]Note that $t(B_i)$ is a node, while $t(F_i)$ is a set.

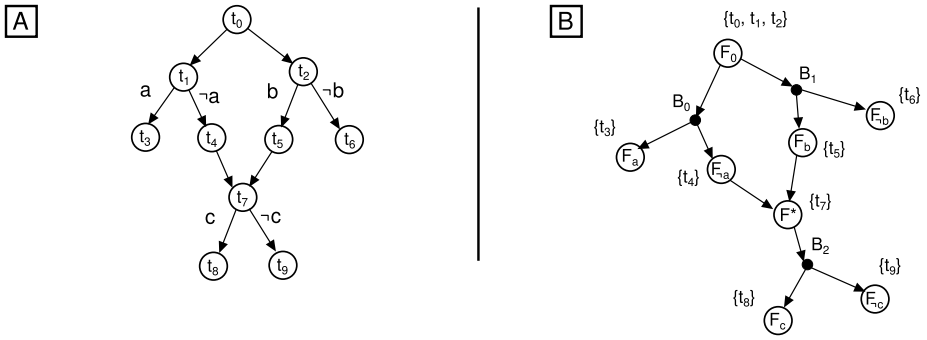


Figure 4.7: A: The non CFU CTG from Figure 4.3.B; B: The associated BFG

For instance, according to the definition, the BFG corresponding to the CTG in Figure 4.6A contains a branch node for each branch: B_0 corresponds to t_1 (i.e. $t(B_0) = t_1$), B_1 to t_6, B_2 to t_7, B_3 to t_{12}, B_4 to t_{15}. As for F-nodes, F_0 represents the whole sample space and nodes t_0, t_1, t_{12}, t_{15} are mapped to it (i.e. $t(F_0) = \{t_0, t_1, t_{12}, t_{15}\}$), as they execute in all scenarios. F-node F_b corresponds to the set of events $\{\neg abcde, \neg abc\neg de, \neg abcd\neg e, \neg abc\neg d\neg e, \neg ab\neg cde, \neg ab\neg c\neg de, \neg ab\neg cd\neg e, \neg ab\neg c\neg d\neg e\}$, that is the set of events where outcomes $\neg a$ and b are both true; t_8 is the only task in $t(F_b)$ as it executes in all such scenarios (condition (2b)) and does not execute in any superset of scenarios (condition (2c)).

Concerning the BFG connectivity, condition (1) intuitively states that every B-node has an ingoing arc from all F-nodes where the corresponding CTG branch is mapped; in the BFG of Figure 4.6B condition (1) yields all arcs from F-nodes to B-nodes. Condition (2) defines instead the connectivity from B-nodes to F-nodes: it tells that every B-node has an outgoing arc for each outcome of the corresponding CTG branch; the destination of such BFG arc is the F-node where the destination of the arc with that outcome (task t_k) is mapped to. The BFG arc is labeled with the corresponding CTG outcome. In Figure 4.6B condition (2) yields all arcs from B-nodes to F-nodes.

Finally condition (3) (that never happens in CTG satisfying the CFU) defines connectivity between F-nodes and other F-nodes linked by no path resulting from conditions (1) and (2). In particular, arcs (F_i, F_j) are built where F_j is the destination F-node, and F_i is the "minimal" (see condition (3b)) F-node such that $\sigma(F_j) \subset \sigma(F_i)$ (see condition (2b)). Observe that parents of B-nodes are always F-nodes, parents of F-node can be both F-nodes and B-nodes; children of B-nodes are always F-nodes, children of F-nodes can be both F-nodes and B-nodes. As an example consider Figure 4.7 where we have links between F-nodes in the BFG corresponding to a CTG that does not satisfy CFU.

Some properties follow from the BFG definition. First of all, *given a CTG, its associated BFG is uniquely defined.* The result comes from the fact that node condition (1) univocally defines the set of B-nodes, node condition (2c) univocally selects the set of scenarios to which every F-node corresponds to and the graph connectivity is univocally defined once F-nodes and B-nodes are given.

The family of mappings of CTG nodes to F-nodes in general does not par-

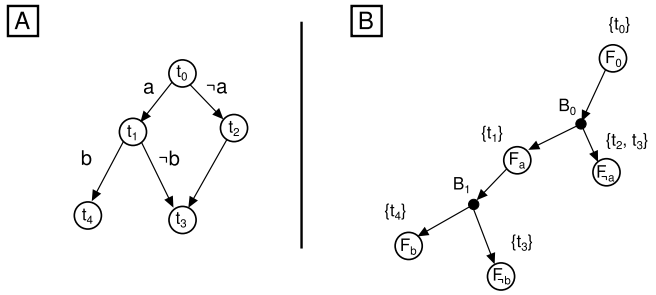


Figure 4.8: Task t_3 is mapped on two F-nodes ($F_{\neg a}, F_{\neg b}$)

tition nodes in the CTG; Figure 4.8 shows an example graph where a node (namely t_3) is mapped to more than one F-node in the BFG ($F_{\neg a}, F_{\neg b}$). The following theorem holds:

Theorem 1. *If a CTG node t_i is mapped to more than one F-node, such F-nodes are mutually exclusive, and represent pairwise disjoint sets of scenarios.*

Proof. Suppose a t_i is both mapped on F-nodes F_j and F_h; this can be true only if no set of events σ' exists such that (2a), (2b) (2c) are satisfied. From condition (2b), $t_i \in TG(s) \forall s \in \sigma(F_j) \cup \sigma(F_h)$, hence $\sigma' = \sigma(F_j) \cup \sigma(F_h)$ satisfies (2b), and of course (2c). Hence $\sigma' = \sigma(F_j) \cup \sigma(F_h)$ must violate (2a), or t_i would not be mapped to F_j and F_h. Therefore, there must be a condition c_k (let O be the set of outcomes) such that events in $\sigma(F_j) \cup \sigma(F_h)$ do not contain the whole set of outcomes of c_k and $\sigma(F_j)$, $\sigma(F_h)$ contains exactly one outcome of c_k. As different outcomes of the same condition generate mutually exclusive events, $\sigma(F_j)$ and $\sigma(F_h)$ are mutually exclusive. □

Note also that in general F-nodes and B-nodes can have more than one parent (despite this is not the case for F-nodes in Figure 4.6 and 4.8), as well as more than one child. In particular:

Theorem 2. *Parents of B-nodes are always mutually exclusive; parents of F-nodes are never mutually exclusive.*

Proof. Let B_i be a B-node; due to connectivity condition (1), its parents are the F-nodes where the corresponding CTG branch $t(B_i)$ is mapped; due to Theorem 1 those F-nodes are mutually exclusive.

Now, let F_i be an F-node, with F-node parents F' and F'', then $\sigma(F_i) \subset \sigma(F')$ and $\sigma(F_i) \subset \sigma(F'')$, due to connectivity condition (3). Note that the strict inclusion holds, hence $\sigma(F') \nsubseteq \sigma(F'')$, $\sigma(F'') \nsubseteq \sigma(F'')$ and $\sigma(F') \cap \sigma(F'') \neq \emptyset$. Therefore the two parents are non mutually exclusive, as they share some event. The reasoning still holds when one parent (or both) is a B-node B_j, by substituting $\sigma(F')$ with $\{s \in \sigma(F') \mid t(B_j) \in t(F')\}$. □

4.5.2 BFG and Control Flow Uniqueness

Control flow uniqueness translates into additional properties for the BFG:

Theorem 3. *If CFU holds, every F-node has a single parent and it is always a B-node.*

Proof. (1. Every F-node has a single parent) Suppose a node F_i has two parents F', F'' (see the proof of Theorem 2 for how to adapt the reasoning to B-nodes). Let t_j be a CTG and-node $\in t(F_i)$, and consider two incoming arcs $a' = (t', t_j)$ and $a'' = (t'', t_j)$ such that $a' \Rightarrow a''$ (for CFU to hold); either the parents t' and t'' are in $t(F_i)$ as well, or they are mapped to some *ancestors* of F_i; in this case they execute in the events represented by any descendant of such ancestors (as a consequence of connectivity conditions), hence they also execute on some parents of F_i. It is therefore always possible to identify a t_j such that:

(a) $t_j \in t(F_i)$
(b) parents t' and t'' respectively execute in all events in $\sigma(F')$ and $\sigma(F'')$.

Now, since $a' \Rightarrow a''$, in terms of set of events this implies $\sigma(F'') \subset \sigma(F')$. However, due to Theorem 2, we know that and $\sigma(F'') \nsubseteq \sigma(F')$, which leads to a contradiction. Hence, if CFU holds, every F-node has a single parent.

(2. The parent is a B-node) Suppose there exists an F-node F_i with a single, F-node, parent F'. As F' is the only parent of F_i and there is no intermediate B-node, every node $t_j \in t(F_i)$ executes in $\sigma(F')$ as well. At the same time, due to connectivity condition (3), $\sigma(F_i) \subset \sigma(F')$; hence such a node F_i, if existent, would fail to satisfy node condition (2c). Therefore, the single parent of every F-node is a B-node. □

From Theorem 3 we deduce that if CFU holds the BFG is a bi-chromatic *alternate* graph. Moreover, since every branch node with m outgoing arcs originates exactly m F-nodes, the BFG has exactly $n_o + 1$ F-nodes, where n_o is the number of condition outcomes; for this reason, when CFU is satisfied, one can denote F-nodes (other than the root) by the outcome they refer to; for example an F-node referring to outcome $a = true$ will be denoted as F_a, if referring to $b = false$ as $F_{\neg b}$ and so on.

CFU is also a necessary condition for the structural property listed above to hold; therefore we can check CFU by trying to build a BFG with a single parent for each F-node: if we cannot make it, then the original graph does not satisfy the condition. The BFG construction procedure in case CFU is satisfied is outlined in the following section.

4.5.3 BFG construction procedure if CFU holds

The BFG construction procedure has exponential time complexity in case the CTG satisfies CFU since the number of possible conjunctions in the activation events is exponential; in practice we can devise a polynomial time algorithm if Control Flow Uniqueness holds. In this case we know the BFG contains an F-node for each condition outcome in the original CTG, plus an F root node; therefore we can design an algorithm to build a BFG with an F-node for each condition outcome, and check at each step whether CFU actually holds; if a violation is encountered we return an error.

In the following, we suppose that each CTG node t_i is labeled with the set of condition outcomes in all paths from the root node to t_i (or "upstream

conditions"); this can be easily done in polynomial time by means of a forward visit of the graph.

A polynomial time complexity BFG building procedure for graphs satisfying CFU is shown in Algorithm 2. The algorithm performs a forward visit of the Conditional Task Graph starting from the root node; as the visit proceeds the BFG is built and the CTG nodes are mapped to F-nodes. The acyclicity of the CTG ensures that whenever a CTG node t_i is visited, all F-nodes needed to map it have already been built.

Figure 4.9 shows an example of the procedure, where the input is a subgraph of the CTG from Figure 4.2A. Initially (Figure 4.9B) the L set only contains the root task t_5 and $V = \emptyset$. The CTG node t_5 is visited (line 5) and mapped to the pre-built F-node F_0. The `mapping_set` function at line 6 returns the set of F-nodes in the current BFG on which t_5 has to be mapped; details on how to compute this set will be given later. In the next step (Figure 4.9C) t_6 is processed and mapped on F_0; however, being t_6 a branch, a new B-node is built (line 9) and an F-node for each condition outcome (F_b, $F_{\neg b}$). In Figure 4.9D t_7 is visited and, similarly to t_6, a new B-node and two new F-nodes are built. Next, t_8 is visited and mapped on F_b (Figure 4.9E); similarly t_9 is mapped on $F_{\neg b}$, t_{10} on F_c and t_{11} on $F_{\neg c}$ (those steps are not shown in the figure). Finally, in Figure 4.9F and 4.9G, nodes t_{20} and t_{21} are visited; since the first is triggered by both the outcomes b and $\neg b$ it is mapped directly on F_0; t_{21} is instead an and node, whose main predecessor is t_{10}, therefore the `mapping_set` function maps it on the same F-node as t_{10}.

Algorithm 2: Building a BFG

```
1: input:  a Conditional Task Graph with all nodes labeled with the set of their
   upstream conditions
2: build the root F-node F_0
3: let L be the set of nodes to visit and V the one of visited nodes. Initially L
   contains the CTG root and V = ∅
4: while L ≠ ∅ do
5:    pick the first node t_i ∈ L, remove t_i from L
6:    let F(t_i) = mapping_set(t_i) be the set of F-nodes t_i has to be mapped on
7:    map t_i on all F-nodes in F(t_i)
8:    if t_i is a branch then
9:       build a B-node B_i
10:      build an F-node F_Out for each condition outcome Out of the branch
11:      connect each F-node in F(t_i) to B_i
12:   end if
13:   add t_i to V
14:   for all child node t_j of t_i do
15:      if all parent nodes of t_j are in V, add t_j to L
16:   end for
17: end while
```

The set $F(t_i)$ for each CTG node is computed by means of a two-phase procedure. In first place an extended set of F-nodes is derived by a forward visit of the BFG built so far. The visit starts from the root F-node. At each step: (A) if an F-node is visited, then all its children are also visited; (B) if a B-node is visited, then each child nde is visited only if the corresponding condition outcome appears in the label of t_i (which reports the "upstream outcomes"). The

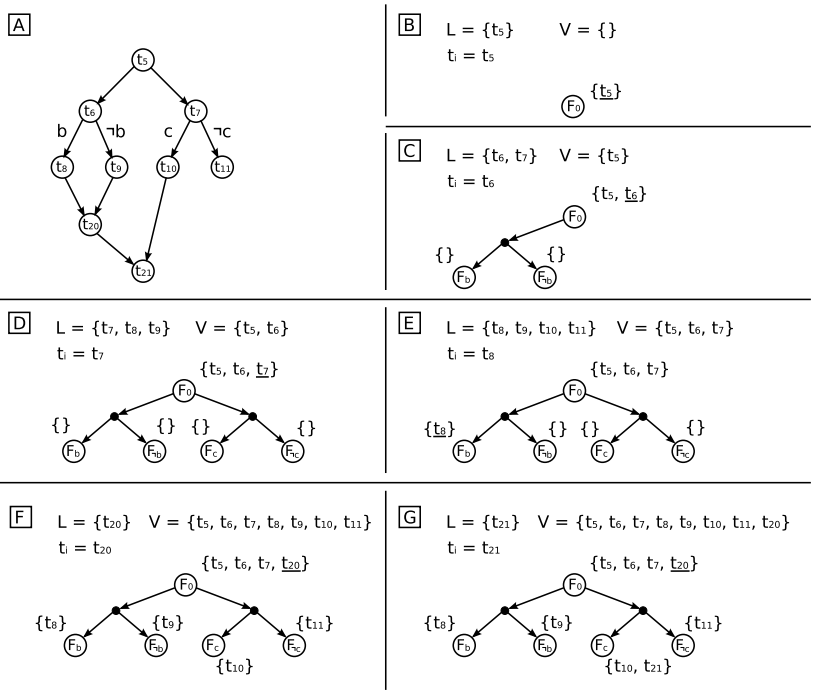


Figure 4.9: BFG building procedure

CTG node t_i is initially mapped to all leaves reached by the visit; for example, with reference to Figure 4.9G, CTG node t_{21} is first mapped to $F_b, F_{\neg b}, F_c$.

In the second phase this extended set of F-nodes is simplified by recursively applying two simplification rules; in order to make the description clearer we temporarily allow CTG branch nodes to be mapped on B-nodes: B-node mappings, however, will be discarded at the end of the simplification process.

rule 1: if a B-node B_i is the *only* parent of F-nodes $F_0, F_1, \ldots$ and a task t_j is mapped on all of them, then add B_i to $F(t_j)$. For example, with reference to Figure 4.9G, where initially $F(t_{21}) = \{F_b, F_{\neg b}, F_c\}$, after an application of rule 1 we have $F(t_{21}) = \{B_{left}, F_b, F_{\neg b}, F_c\}$

rule 2: if a task t_j is mapped on a B-node B_i with parent $F_0, F_1, \ldots$, then B_i and all its descendants can be removed from $F(t_j)$. If at the end of this process no descendant of F_0, F_1, etc is in $F(t_j)$, then map t_j on F_0, F_1, etc. For example, after the application of rule 2, $F(t_{21})$ becomes $\{F_c\}$.

Once all simplifications are done, all remaining F-nodes in $F(t_i)$ must be mutually exclusive, as said in Section 4.5.1 and shown in Figure 4.8: if this fails to occur it means the BFG has not enough F-nodes for the mapping, which in turn means the original graph does not meet CFU. In this case an error is reported.

4.5.4 BFG and scenarios

The most interesting feature of a BFG is that it can be used to select and encode groups of scenarios in which arbitrarily chosen nodes execute. A specific algorithm can then be applied to such scenarios, in order to compute the corresponding probability, or any other feature of interest.

Groups of scenarios are encoded in the BFG as sets of s-trees:

Definition 15 (S-Tree)**.** *An s-tree is any subgraph of the BFG satisfying the following properties:*

1. *The subgraph includes the root node*
2. *If the subgraph includes an F node, it includes also all its children*
3. *If the subgraph includes an F-node, it includes also all its parents*
4. *If the subgraph includes a B-node, it includes also one and only one of its children.*

Note that the s-tree associated to a scenario s is the BFG associated to the deterministic task graph $TG(s)$.

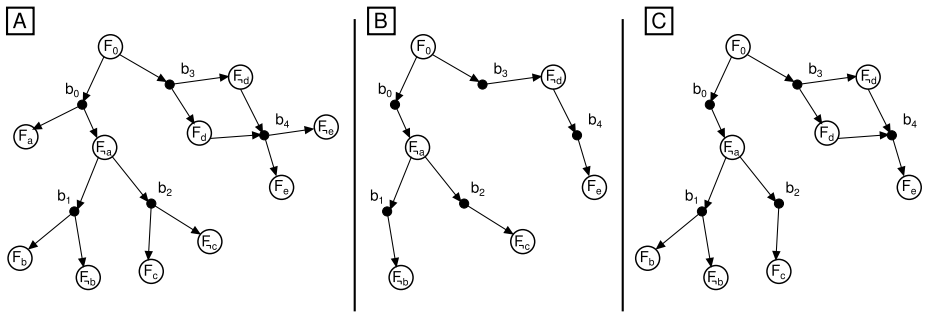


Figure 4.10: (A) BFG for the graph in Figure 4.6 - (B) the s-tree associated to the scenario $\neg a \neg b \neg c \neg d e$ - (C) a subgraph (set of s-trees) associated to scenarios where $\neg ace$ holds

Despite its name, an s-tree is not necessarily a tree: this is always the case only if CFU holds (which is not required by Definition 15) (see Figure 4.10A/B, where CFU holds and F-nodes are labeled with the condition outcome they refer to).

By relaxing condition (4) in Definition 15 and allowing the inclusion of more than one condition per B-node, we get a subgraph representing a set of s-trees; a single s-tree (and hence a scenario) can be derived by choosing from the subgraph a single outcome per branch condition. For example from the subgraph in Figure 4.10C one can extract the set of s-trees corresponding to $\neg abcde$, $\neg abc \neg de$, $\neg a \neg bcde$, $\neg a \neg bc \neg de$. This encoding method is sufficient to represent sets of scenarios of practical interest (e.g. those required by the algorithm and constraints discussed in the chapter). The importance of s-trees mainly lies in the fact that they are required for the algorithm presented in the forthcoming Section 4.5.6.

4.5.5 Querying the BFG

We now restrict our attention to CTG satisfying Control Flow Uniqueness; namely we want to provide a way to select a set of s-trees representing a set of scenarios which include or exclude a specified group of nodes; once such a subgraph is available, the execution probability can be extracted by a proper algorithm. We consider selection rules specified by means of either conjunctions or disjunctions of positive and negative terms[3]. Each basic term of the query can be either t_i (with meaning "task t_i executes") or $\neg t_i$ (with meaning "task t_i does not execute"). Some examples of valid queries are:

$$q_0 = t_i \wedge t_j \qquad q_1 = t_i \wedge \neg t_j \wedge \neg t_k \qquad q_2 = t_i \vee t_j$$

A query returns the BFG subgraph representing the events where the specified tasks execute/do not execute, or *null* in case no such event exists. The idea of the query processing procedure is that, since the complete BFG represents all possible scenarios, we can select a subset of them by removing F-nodes which do not satisfy the boolean query. Thus, in order to be processed, queries are first negated:

$$\neg q_0 = \neg t_i \vee \neg t_j \qquad \neg q_1 = \neg t_i \vee t_j \vee t_k \qquad \neg q_2 = \neg t_i \wedge \neg t_j$$

Each element in the negated disjunction now has to be mapped to a set of F-nodes to be removed from the BFG. This can be efficiently done by precomputing for each BFG node an *inclusion label* and an *exclusion label*:

1. Inclusion labels A CTG task t_i is in the *inclusion label* $i(F_j)$ of an F-node F_j either if it is directly mapped on it, $t_i \in t(F_j)$, or if t_i is in the inclusion label of any of its parents.

A CTG task t_i is in the *inclusion label* $i(B_j)$ of a B-node B_j if t_i is in the inclusion label of all of its parents.

In practice, $t_i \in i(F_j)$ (resp. $i(B_j)$) if it *does* execute in all scenarios corresponding to every s-tree containing F_j (resp. B_j).

2. Exclusion labels A CTG task t_i is in the *exclusion label* $e(F_j)$ of an F-node F_j either if parents of F_j are F-nodes[4] and t_i is in the exclusion label of any parent, or if the parent of F_j is a B-node and it exists a brother node F_k such that t_i is mapped on a descendant (either direct or not) of F_k and t_i is not mapped on a descendant (either direct or not) of F_j.

A CTG task t_i is in the *exclusion label* $e(B_j)$ of a B-node B_j if t_i is in the exclusion label of all of its parents.

In practice, $t_i \in e(F_j)$ (resp. $e(B_j)$) if it *cannot* execute in the scenario correspondent to any s-tree containing F_j (resp. B_j).

For example in Figure 4.6B (reproduced in Figure 4.11A for sake of clarity), the inclusion label of node F_0 is $i(F_0) = \{t_0, t_1, t_{12}, t_{15}\}$ and $i(B_3)$ is equal to

[3]Mixed queries are also allowed by converting them to groups of conjunctive queries representing disjoint sets of scenarios, but paying an exponential complexity blow-up, depending on the size and the structure of the query. Pure conjunctive and disjunctive queries are however enough for managing cases of practical interest as shown in the rest of the chapter.

[4]This property holds for general CTG, while if CFU is satisfied parents of F-nodes are always B-nodes.

$i(F_0)$; then $i(F_d) = i(F_0) \cup \{t_{14}\}$ and $i(F_{\neg d}) = i(F_0) \cup \{t_{13}\}$; $i(B_4)$ is again equal to $i(F_0)$, since neither t_{13} nor t_{14} are mapped on both parents of B_4. As for the exclusion labels: $e(F_0) = \emptyset$ and $e(B_0) = \emptyset$; $e(F_{\neg a}) = \{t_2, t_3, t_4\}$, since those tasks are mapped on the brother node F_a and they are not mapped on any descendant of $F_{\neg a}$.

Once inclusion and exclusion labels are computed, each (conjunctive) term of the query (e.g. $t_i \wedge \neg t_j \wedge \ldots$) is mapped to a set of F/B-nodes satisfying $t_i \in i(F_j)$ (or $i(B_j)$) for every positive element t_i in the term, and $t_i \in e(F_j)$ (or $e(B_j)$) for each negative element $\neg t_i$ in the term. For example:

$$
\begin{aligned}
t_i &\rightarrow \{F_j \mid t_i \in i(F_j)\} \cup \{B_j \mid t_i \in i(B_j)\} \\
\neg t_i &\rightarrow \{F_j \mid t_i \in e(F_j)\} \cup \{B_j \mid t_i \in e(B_j)\} \\
t_i \wedge t_k &\rightarrow \{F_j \mid t_i, t_k \in i(F_j)\} \cup \{B_j \mid t_i, t_k \in i(B_j)\} \\
t_i \wedge \neg t_k &\rightarrow \{F_j \mid t_i \in i(F_j),\ t_k \in e(F_j)\} \cup \{B_j \mid t_i \in i(B_j),\ t_k \in e(B_j)\}
\end{aligned}
$$

Note that an (originally) conjunctive query is mapped to a set of terms, each consisting of a single positive or negative task literal; the query is processed by removing from the complete BFG the F and B-nodes corresponding to each term. Conversely, an (originally) disjunctive query yields a single term consisting of a conjunction of positive of negative task; the query is processed by removing from the BFG the F and B-nodes corresponding to the term. For example, on the graph of Figure 4.6B (reproduced in Figure 4.11A for sake of clarity), the query $q = t_{21} \wedge \neg t_3 \wedge \neg t_{16} = \neg(\neg t_{21} \vee t_3 \vee t_{16})$ is processed by removing from the BFG $F_{\neg c}$, F_a and $F_{\neg e}$, since $t_{21} \in e(F_{\neg c})$, $t_3 \in i(F_a)$ and $t_{16} \in i(F_{\neg e})$. The resulting subgraph is the one shown in Figure 4.10C (reproduced in Figure 4.11B).

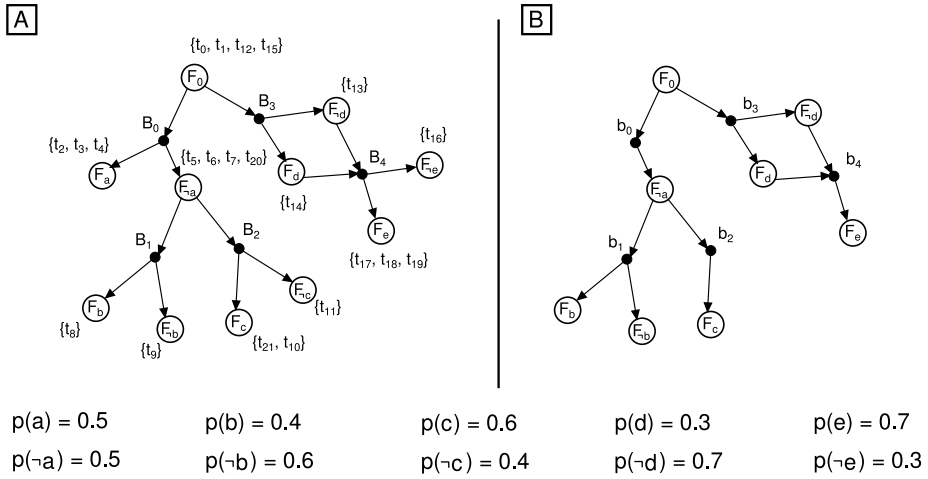


Figure 4.11: (A) the BFG from Figure 4.6B (B) the subgraph from Figure 4.10C

Disconnected nodes are removed at the end of the process. During query processing, one has to check whether at some step any B-node loses all children; in such case the output is *null*, as the returned BFG subgraph would contain a B-node with no allowed outcome and this is impossible. Similarly, the result is

null if all BFG nodes are removed. A query is always processed in linear time. Finally, the following theorem holds:

Theorem 4. *If a query returns a BFG subgraph, this represents a set of s-trees.*

Proof. Assume the query result is not *null* and remember we consider CFU to be satisfied; then condition (1) in Definition 15 is trivially satisfied, as a non-empty query always includes the root node. Conditions (2) and (3) are satisfied as, in a graph satisfying CFU, children and parents of F-nodes are always B-nodes, and B-nodes are never removed when processing a query. Finally, condition (4) is satisfied as query processing may remove some children of a B-node, but not all of them (or *null* would be returned). □

As the result of a query is always a set of s-trees, it can be used as input for the backward visit algorithm.

4.5.6 Visiting the BFG

Many algorithms along the chapter are based on a backward visit of the BFG. During these visits each algorithm collects some attributes stored in F and B nodes. We therefore propose a meta algorithm, using a set of parameters which have to be defined case by case. All backward visit based algorithms assume CFU is satisfied and require as input a subgraph representing a set of s-trees (hence a BFG as a particular case).

In particular, Algorithm 3 shows the generic structure of a backward visit of a given BFG. The visit starts from the leaves and proceeds to the root; every predecessor node is visited when all its successors are visited (lines 12-13). The meta algorithm is parametric in the five-tuple $\langle A, init_F, init_B, update_F, update_B \rangle$. In particular A is a set of possible attribute values characterizing each F and B-node, and $A(n)$ denotes the values of the attributes for node n; the function $init_F : \{F_i\} \rightarrow A$ associates to an F-node an element in A, that is values for each of its attributes, and the function $update_F : \{F_i\} \times \{B_j\} \rightarrow A$ associates to an F-node and a B-node an element in A. The functions $init_B : \{B_i\} \rightarrow A$ and $update_B : \{B_i\} \times \{F_j\} \rightarrow A$ are defined similarly to $init_F$ and $update_F$ for B-nodes. In the algorithm, $init_F$ and $init_B$ are used to assign an initial value of attributes in A to each F and B-node (line 2); the function $update_F$ is used to update the attribute values of the *parent* of an F-node (which is a B-node — line 6) and $init_B$ is used to update the attributes of the *parent* of a B-node (which is an F-node — line 8). In the following, we will use Algorithm 1 with different parameter settings for different purposes.

4.5.7 Computing subgraph probabilities

In the following we show how to compute the probability for a given BFG, or part of it (sets of s-trees derived from querying the BFG).

The probability of a subgraph can be computed via a backward visit which is an instantiation of the meta Algorithm 3. In particular, a single attribute p, representing a probability, is stored in F and B-nodes, and thus $A = [0, 1]$. The result of the algorithm is the probability value of the root node. The *init* and

Algorithm 3: Backward visit(A, $init_F$, $init_B$, $update_F$, $update_B$)

1: let L be the set of nodes to visit and V the one of visited nodes. Initially L contains all subgraph leaves and $V = \emptyset$
2: for each F and B-node, store the values of attributes in A. Initially set $A(n) = init_F(n)$ for all F nodes, $A(n) = init_B(n)$ for all B-nodes
3: **while** $L \neq \emptyset$ **do**
4: pick a node $n \in L$
5: **if** n is an F-node with parent n_p **then**
6: $A(n_p) = update_F(n, n_p)$
7: **else if** n is a B-node **then**
8: **for every parent** n_p**:** $A(n_p) = update_B(n, n_p)$
9: **end if**
10: $V = V \cup \{n\}$
11: $L = L \setminus \{n\}$
12: **for** every parent n_p **do**
13: **if** all children of n_p are in V **then** $L = L \cup \{n_p\}$
14: **end for**
15: **end while**

update functions are as follows:

$$\begin{aligned} init_F(F_i) &= \text{the probability of the outcome labeling the arc} \\ &\qquad \text{from the single B-node parent of } F_i \text{ and } F_i \text{ itself} \\ init_B(B_i) &= 0 \\ update_F(F_i, B_j) &= p(B_j) + p(F_i) \\ update_B(B_i, F_j) &= p(F_j) \cdot p(B_i) \end{aligned}$$

As an example, consider the subgraph of Figure 4.10C (also reported in Figure 4.11B, together with the probabilities). The computation starts from the leaves; for example at the beginning $p(F_b) = 0.4$, $p(F_{\neg b}) = 0.6$, $p(F_c) = 0.6$ (set by $init_F$). Then, probabilities of B-nodes are the weighted sum of those of their children (see $update_F$); for example $p(b_1) = p(F_b) + p(F_{\neg b}) = 0.4 + 0.6 = 1$ and $p(b_2) = p(F_c) = 0.6$. Probabilities of F-nodes are instead the product of those of their children (see $update_B$), and so $p(F_{\neg a}) = p(b_1)p(b_2) = 0.6$. The visit proceeds backwards until $p(F_0)$ is computed, which is also the probability of the subgraph.

4.6 Objective Function

One of the purposes of the probabilistic reasoning presented so far is to derive the expected value of a given objective function efficiently. We consider in this section three examples of objective functions that are commonly used in constraint based scheduling, described in Section 4.4.4.3: the minimization of costs of single task-resource assignments, the minimization of the assignment cost of pairs of tasks, and the makespan. We refer to the first two examples as *objective functions depending on the resource allocation* while we refer to the third case as *objective function depending on the task schedule*.

This first and the second case are easier since we can transform the expected value of the objective function in a deterministic objective function provided that we are able to compute the probability a single task executes and the probabilities that a pair of tasks executes respectively. The third example is much more complicated since there is not a declarative description of the objective function that can be computed in polynomial time. Therefore, we provide an operational definition of such expected value by defining an expected makespan constraint, and the corresponding filtering algorithm.

4.6.1 Objective function depending on the resource allocation

We first consider an objective function depending on single tasks assignments and on the run time scenario; for example, suppose there is a fixed cost for the assignment of each task t_i to a resource $res(t_i)$, as it is the case for objective (4.2) in Section 4.4.4.3. The general form of the objective function on a given scenario s is.

$$Obj^{(s)} = \sum_{t_i \in TG(s)} cost(t_i, res(t_i)) = \sum_{t_i \in T} f_{t_i}(s) cost(t_i, res(t_i))$$

We remind that $f_{t_i}(s) = 0$ if $t_i \notin TG(s)$. According to Definition 8, the expected value of the objective function is

$$E(Obj) = \sum_{s \in S} p(s) Obj^{(s)} = \sum_{s \in S} p(s) \sum_{t_i \in T} f_{t_i}(s) cost(t_i, res(t_i))$$

We remind that $S_i = \{s \mid t_i \in TG(s)\}$ is the set of all scenarios where task i executes. Thus,

$$E(Obj) = \sum_{t_i \in T} \left[cost(t_i, res(t_i)) \sum_{s \in S_i} p(s) \right]$$

Now every stochastic dependence is removed and the expected value is reduced to a deterministic expression. Note that $\sum_{s \in S_i} p(s)$ is simply the probability of execution of node/task i. This probability can be efficiently computed by running Algorithm 3 instantiated as explained in Section 4.5.7, on the BFG sub-graph resulting from the query $q = t_i$.

As a second example, we suppose the objective function is related to arcs and to the run time scenario; again, we assume there is a fixed cost for the activation of an arc, as it is the case for the objective (4.3) in Section 4.4.4.3. The general form of the objective function is.

$$\begin{aligned} Obj^{(s)} &= \sum_{a_h = (t_i, t_j) \in TG(s)} cost(res(t_i), res(t_j)) = \\ &= \sum_{a_h = (t_i, t_j) \in A} f_{t_i}(s) f_{t_j}(s) cost(t_i, res(t_i), t_j, res(t_j)) \end{aligned}$$

The expected value of the objective function is

$$E(Obj) = \sum_{s \in S} p(s) \sum_{a_h = (t_i, t_j) \in A} f_{t_i}(s) f_{t_j}(s) cost(t_i, res(t_i), t_j, res(t_j))$$

note that $cost(t_i, res(t_i), t_j, res(t_j))$ is a cost that we can derive from a cost matrix c

$$E(Obj) = \sum_{a_h=(t_i,t_j)\in A} \left[cost(t_i, res(t_i), t_j res(t_j)) \sum_{s\in S_i\cap S_j} p(s) \right]$$

Now every stochastic dependence is removed and the expected value is reduced to a deterministic expression. Note that $\sum_{s\in S_i\cap S_j} p(s)$ is the probability that both tasks i and j execute. Again this probability can be efficiently computed using Algorithm 3 on the BFG sub-graph resulting from query $q = t_i \wedge t_j$.

4.6.2 Objective function depending on the task schedule

For a deterministic task graph, the makespan is simply the end time of the last task; it can be expressed as: $makespan = \max\{\mathtt{E}_i \mid t_i \in T\}$. If the task graph is conditional the last task depends on the occurring scenario. Remember we are interested in finding a single assignment of start times, valid for each execution scenario; in this case each scenario s identifies a deterministic Task Graph ($TG(s)$) and its makespan is $\max\{\mathtt{E}_i \mid t_i \in TG(s)\}$. Thus, the most natural declarative expression for the expected makespan would be:

$$E(makespan) = \sum_{s\in S} p(s) \max\{\mathtt{E}_i \mid t_i \in TG(s)\} \tag{4.4}$$

where $p(s)$ is the probability of the scenario s. Note that the expression can be simplified by considering only tail tasks (i.e. tasks such that $|A^+(t_i)| = 0$). For example, consider the CTG depicted in Figure 4.12A, the scenarios are $\{a\}$, $\{\neg a, b\}$, $\{\neg a, \neg b\}$, and the expected makespan can be expressed as:

$$\begin{aligned} E(makespan) \quad = \quad & p(a) \max\{\mathtt{E}_2, \mathtt{E}_6\} + \\ & p(\neg a \wedge b) \max\{\mathtt{E}_4, \mathtt{E}_6\} + \\ & p(\neg a \wedge \neg b) \max\{\mathtt{E}_5, \mathtt{E}_6\} \end{aligned}$$

Unluckily the number of scenarios is exponential in the number of branches, which limits the direct use of expression (4.4) to small, simple instances. Therefore, we defined an expected makespan global constraint

$$\texttt{exp\_mkspan\_cst}([\mathtt{E}_1, \ldots, \mathtt{E}_n], \texttt{emkspan})$$

whose aim is to compute legal bounds on the expected makespan variable `emkspan` and on the end times of all tasks ($\mathtt{E}_i$) in a procedural fashion. We devised a filtering algorithm described in Section 4.6.2.1 whose aim is to prune the expected makespan variable on the basis of the task end variables, and vice-versa, see Section 4.6.2.2.

4.6.2.1 Filtering the expected makespan variable

The filtering algorithm is based on a simple idea: the computation of the expected makespan is tractable when the order of tasks, and consequently of end variables, is known. Consider the schedule in Figure 4.12B, where all tasks use a unary resource $URes_0$: since t_5 is the last task, the makespan of all scenarios

containing t_5 is $\mathtt{E}_5$. Similarly, since t_4 is the last but one task, $\mathtt{E}_4$ is the makespan value of all scenarios containing t_4 and not containing t_5, and so on.

The computation can be done even if start times have not yet been assigned, as long as the end-order of tasks is known; in general, let $t_0, t_1, \ldots, t_{n_t-1}$ be the sequence of CTG tasks ordered by increasing end time, then:

$$E(makespan) = \sum_{i=0}^{n_t-1} p(t_i \wedge \neg t_{i+1} \wedge \ldots \wedge \neg t_{n_t-1})\ \mathtt{E}_i \tag{4.5}$$

The expected makespan can therefore be computed as a weighted sum of end times, where the weight of task t_i is given by the probability that 1) t_i executes 2) none of the task ending later than t_i executes. The sum contains n_t terms, where n_t is the number of tasks; again this number can be decreased by considering tail tasks only.

Hence, *once the end order of tasks is fixed*, we can compute the expected makespan in polynomial time, we just need to be able to efficiently compute the probability weights in expression (4.5): if CFU holds, this can be done as explained in Section 4.5 by running Algorithm 3 (in its probability computation version) on the BFG subgraph resulting from query $q = t_i \wedge \neg t_{i+1} \wedge \ldots \wedge \neg t_{n_t-1}$.

In general, however, during search the order of tasks is not fixed, but it is always possible to identify possibly infeasible task schedules whose makespan, computed with expression (4.5), can be used as a bound for the expected makespan variable. We refer to these schedules as S_{min} and S_{max}, see Figure 4.13. In particular, S_{min} is a schedule where all tasks are assumed to end at the minimum possible time and are therefore sorted by increasing $\min(\mathtt{E}_i)$; conversely, in S_{max} tasks are assumed to end at the maximum possible time, hence they are ordered according to $\max(\mathtt{E}_i)$. Obviously both situations will likely be infeasible, but have to be taken into account. Moreover, the following theorem holds:

Theorem 5. *The expected makespan assumes the maximum possible value in the S_{max} schedule, the minimum possible value in the S_{min} schedule.*

Proof. Let us take into account S_{max}. Let $t_0, \ldots, t_{n-1}$ be the respective task order; the corresponding expected makespan value due to expression 4.5 is a weighted sum of (maximum) end times:

$$emkspan(S_{max}) = w_0 \cdot max(\mathtt{E}_0) + \cdots + w_{n-1} \cdot max(\mathtt{E}_{n-1})$$

Note that $\sum_i w_i = 1$ as weights are probability; also note weights are univocally defined by the task order. If S_{max} were not the expected maximum makespan

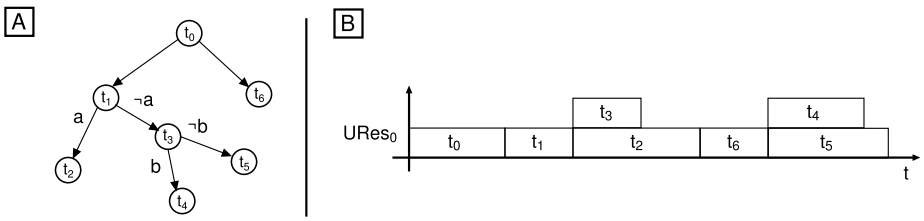


Figure 4.12: Temporal task grouping

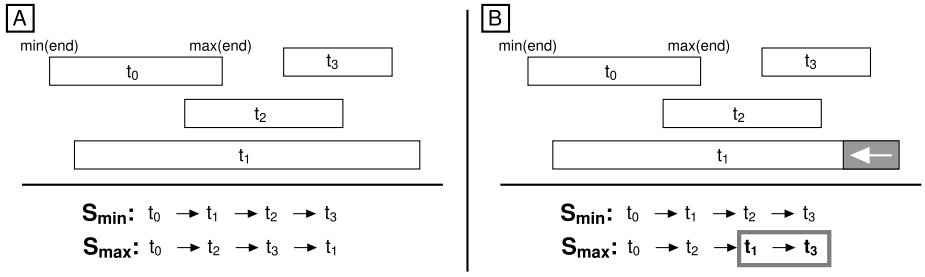


Figure 4.13: A: Example of S_{min} and S_{max} sequences. B: An update of $\mathtt{E}_1$ causes a swap in S_{max}

.

schedule, it should be possible to increase the expected makespan value by reducing the end time of some tasks. Now, let us gradually decrease $max(\mathtt{E}_i)$ while maintaining $max(\mathtt{E}_i) \geq max(\mathtt{E}_{i-1})$: as long as w_i does not change the expected makespan value necessarily decreases. When $max(\mathtt{E}_i)$ gets lower than $max(\mathtt{E}_{i-1})$, weights w_i and w_{i-1} change as follows:

$$w_i = p(t_i \wedge \neg t_{i+1} \wedge \ldots \wedge \neg t_{n-1}) \;\rightarrow\; p(t_i \wedge \neg t_{i-1} \wedge \neg t_{i+1} \wedge \ldots \wedge \neg t_{n-1})$$
$$w_{i-1} = p(t_{i-1} \wedge \neg t_i \wedge \neg t_{i+1} \wedge \ldots \wedge \neg t_{n-1}) \;\rightarrow\; p(t_{i-1} \wedge \neg t_{i+1} \wedge \ldots \wedge \neg t_{n-1})$$

hence w_{i-1} gets higher, w_i gets lower. As the sum $\sum_i w_i$ is constant and equal to 1 both before and after the swap, w_{i-1} grows exactly by the amount by which w_i shrinks; in other terms, some of the weight of t_i is transferred to t_{i-1} or, equivalently, t_{i-1} "steals" some weight from t_i. From now on, if we keep on decreasing $max(\mathtt{E}_i)$ the expected makespan will still decrease, just at a slower pace due to the lower value w_i, until w_i will become 0. Hence by reducing the end time of a single time variable the expected makespan can only get worse. Moving more tasks complicates the situation, but the reasoning still holds. An analogous proof can be done for the expected makespan for the S_{min} schedule. □

We can therefore prune the expected makespan variable by enforcing:

$$emkspan(S_{min}) \leq emkspan \leq emkspan(S_{max}) \tag{4.6}$$

In order to improve computational efficiency, we can use F-nodes of the BFG instead of tasks in the computation of $emkspan(S_{min})$ and $emkspan(S_{max})$, exploiting the mapping between tasks (CTG nodes) and F-nodes; details are given later on in Section 4.6.2.3.

Pruning the makespan variable requires to compute the makespan of the two schedules S_{min}, S_{max}; this is done by performing a BFG query (complexity $O(n_t)$) and a probability computation (complexity $O(n_t)$) for each task ($O(n_t)$ iterations). The basic worst case complexity is therefore $O(n_t^2)$, which can be reduced to $O(n_t \log(n_t))$ by exploiting caching and dynamic updates during search. As an intuition, all probability weights in the BFG can be computed at the root of the search tree and cached. Then, each time a variable $\mathtt{E}_i$ changes, possibly some nodes change their positions in S_{min}, S_{max} (see Figure 4.13B, where max($\mathtt{E}_1$) changes and becomes lower than max($\mathtt{E}_3$), thus the two nodes

are swapped); in such a situation, the probabilities of all the re-positioned nodes have to be updated. Each update is done in $O(\log(n_t))$ by modifying the probability weights on the BFG; as no more than n_t nodes can move between a search node and any of its children, the overall complexity is $O(n_t \log(n_t))$.

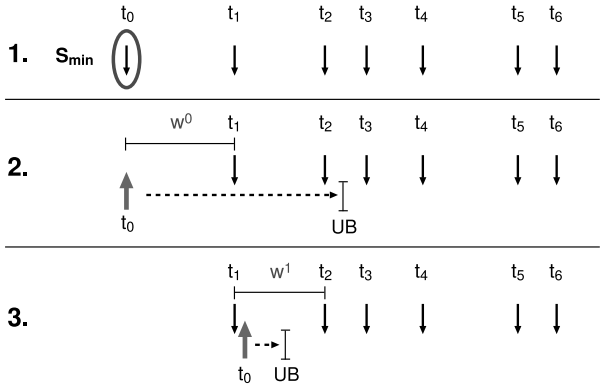


Figure 4.14: Upper bound on end variables

4.6.2.2 Filtering end time variables

When dealing with a makespan minimization problem, it is crucial for the efficiency of the search process to exploit the makespan variable domain updates (e.g. when a new bound is discovered) to filter the end variables domains.

Bounds for $\mathtt{E}_i$ can be computed again with expression (4.5); for example to compute the upper bound for $\mathtt{E}_i$) we have to subtract from the maximum expected makespan value ($\max(mkspan)$) the minimum contribution of all tasks except t_i:

$$\mathtt{E}_i \leq \frac{\max(emkspan) - \sum_{j \neq i} p(t_j \wedge \neg t_{j+1} \wedge \ldots) \min(\mathtt{E}_j)}{p(t_i \wedge \neg t_{i+1} \wedge \ldots)} \tag{4.7}$$

where $t_0, \ldots, t_{i-1}, t_i, t_{i+1}, \ldots$ is the sequence where the contribution of $t_j, j \neq i$ is minimized. Unfortunately, this sequence is affected by the value of $\mathtt{E}_i$. In principle, we should compute a bound for all possible assignments of $\mathtt{E}_i$, while keeping the contribution of other nodes minimized.

Note that the sequence where the contribution of *all* tasks is minimized is that of the S_{min} schedule; we can compute a set of bounds for $\mathtt{E}_i$ by "sweeping" its position in the sequence, and repeatedly applying formula (4.7). An example is shown in Figure 4.14, where a bound is computed for t_0 (step 1 in Figure 4.14). We start by computing a bound based on the current position of t_0 in the sequence (step 2 in Figure 4.14); if such a bound is less than $\min(\mathtt{E}_1)$, then $\max(\mathtt{E}_0)$ is pruned, otherwise we swap t_0 and t_1 in the sequence and update the probabilities (the original probability w^0 becomes w^1) according to expression (4.5). The process continues by comparing t_0 with t_2 and so on until $\max(\mathtt{E}_0)$ is pruned or the end of S_{min} is reached. Lower bounds for $\min(\mathtt{E}_i)$ can be computed similarly, by reasoning on S_{max}.

A detailed description of the filtering procedure is given in Algorithm 4. The tasks are processed as they appear in S_{min} (line 2); for each t_j the algorithm starts to scan the next intervals (line 6). For each interval we compute a bound (lines 7 to 11) based on the maximum makespan value ($\max(mkspan)$), the current task probability/weight (wgt) and the contribution of all other tasks to the makespan lower bound ($rest$).

If the end of the list is reached or the bound is within the interval (line 12) we prune the end variable of the current task (line 13) and the next task is processed. If the bound exceeds the current interval, we move to the next one. In the transition the current task possibly gains weight by "stealing" it from the activity just crossed (lines 15 to 18); wgt and $rest$ are updated accordingly.

Algorithm 4: End variables pruning (upper bound)

```
 1: let S_min = t_0, t_1, ..., t_{k-1}
 2: for j = 0 to k − 1 do
 3:    compute result of query q = t_j ∧ ¬t_{j+1} ∧ ... ∧ t_{k-1} and probability p(q)
 4:    wgt = p(q)
 5:    rest = mkspan(S_min) − min(E_j)wgt
 6:    for h = j to k − 1 do
 7:       if wgt > 0 then
 8:          UB = (max(mkspan) − rest) / wgt
 9:       else
10:          UB = ∞
11:       end if
12:       if h = (k − 1) or UB ≤ min(E_{h+1}) then
13:          set UB as upper bound for t_j
14:       else
15:          remove element ¬t_{h+1} from query q and update p(q)
16:          newwgt = p(q)
17:          rest = rest − (newwgt − wgt) min(E_{h+1})
18:          wgt = newwgt
19:       end if
20:    end for
21: end for
```

The algorithm takes into account all tasks (complexity $O(n_t)$) and for each of them it analyzes the subsequent intervals (complexity $O(n_t)$); weights are updated at each transition with complexity $O(\log(n_t))$ taking care of the fact that a task can be mapped to more than one F-node (note that directly working on F nodes avoids this issue). The overall complexity is $O(n_t^2 \log(n_t))$; by manipulating F-nodes instead of tasks it can be reduced down to $O(n_t + n_o^2 \log(n_o))$, where n_o is the number of condition outcomes in the CTG.

4.6.2.3 Improving the constraint efficiency

In order to improve the computational efficiency of all filtering algorithms used in the expected makespan constraint (see Section 4.6.2.1), we can use F-nodes instead of tasks in the computation of $emkspan(S_{min})$ and $emkspan(S_{min})$. Remember that there is a mapping between tasks (CTG nodes) and F nodes. Each F-node can therefore be assigned a minimum and a maximum end value

computed as follows:

$$\text{maxend}(F_j) = \max\{\max(\mathtt{E}_i) \mid t_i \in t(F_j)\}$$
$$\text{minend}(F_j) = \max\{\min(\mathtt{E}_i) \mid t_i \in t(F_j)\}$$

The rationale behind the formulas is that tasks mapped to an F-node F_i all execute in events in $\sigma(F_i)$; therefore the end time of the set of tasks will always be dominated by the one ending as last.

The two schedules S_{min}, S_{max} can store F-nodes (sorted by minend and maxend) instead of activities and their size can be reduced to at most $n_o + 1$ (where n_o is the number of condition outcomes, often $n_o \ll n_t$): this is in fact the number of F-nodes in a BFG if CFU holds (see Section 4.4.2).

Each time the end variable of a task t_i mapped to F_j changes, values maxend(F_j) and minend(F_j) are updated and possibly some nodes are swapped in S_{min}, S_{max} (similarly to what Figure 4.13B shows for tasks). These updates can be done with complexity $O(\max(n_t, n_o))$, where n_t is the number of tasks. The makespan bound calculation of constraints (4.6) can be done by substituting tasks with F-nodes in expression (4.5), as shown in expression 4.8:

$$E(makespan) = \sum_i p(F_i \wedge \neg F_{i+1} \wedge \ldots \wedge \neg F_{n_o-1})\ end(F_i) \tag{4.8}$$

where $end(F_i) \in [\text{minend}(F_i), \text{maxend}(F_i)]$ and probabilities $p(F_i \wedge \neg F_{i+1} \wedge \ldots \wedge \neg F_{n_o-1})$ can be computed by querying the BFG with $q = F_i \wedge \neg F_{i+1} \wedge \ldots \wedge \neg F_{n_o-1}$. BFG queries involving F-nodes can be processed similarly to usual queries; basically, whilst each task is mapped to one or more F-nodes, an F node is always mapped to a *single* F-node (i.e. itself); thus, (a) the inclusion and exclusion labels can be computed as usual and (b) every update of the weight or the time window of an F-node is performed in strictly logarithmic time. The same algorithms devised for tasks can be used to prune the makespan and the end variables, but the overall complexity goes down to $O(\max(n_t, n_o^2 \log(n_o)))$.

4.7 Conditional Constraints

To tackle scheduling problems of conditional task graphs we introduced the so called *conditional constraints*, which extend traditional constraints to take into account the feasibility in all scenarios.

Let C be a constraint defined on a set of variables X, let S be the set of scenarios of a given CTG, let $X(s) \subseteq X$ be the set of variables related to tasks appearing in the scenario $s \in S$. The conditional constraint corresponding to C must enforce:

$$\forall s \in S \quad C|_{X(s)}$$

where $C|_{X(s)}$ denotes the restriction of constraint C to variables in $X(s)$.

A very simple example is the disjunctive conditional constraint [Kuc03] that models temporal relations between tasks t_i and t_j that need the same unary resource for execution. The disjunctive constraint enforces:

$$mutex(t_i, t_j) \vee (\mathtt{E}(t_i) \leq \mathtt{S}_j) \vee (\mathtt{E}_j \leq \mathtt{S}_i)$$

where *mutex(t_i,t_j)* holds if tasks t_i and t_j are mutually exclusive (see Definition 6) so that they can access the same resource without competition.

As another example, let us consider the cumulative constraint modeling limited capacity resources. The traditional resource constraint enforces for each time instant t:

$$\forall \text{ time } t, \forall R \in \bigcup_{t_i} ResSet_i \quad \sum_{\substack{t_i : \\ \mathtt{S}_i \leq t < \mathtt{E}_i \\ res(t_i) = r_k}} ResCons_i \leq C_k$$

while its conditional version enforces:

$$\forall \text{ time } t, \forall s \in S,\ \forall R \in \bigcup_{t_i \in TG(s)} ResSet_i \quad \sum_{\substack{t_i \in TG(s) : \\ \mathtt{S}_i \leq t < \mathtt{E}_i \\ res(t_i) = r_k}} ResCons_i \leq C_k$$

where the same constraint as above must hold *for every scenario*; this indeed amounts to a relaxation of the deterministic case.
As a consequence, resource requirements of mutually exclusive tasks are not summed, since they never appear in the same scenario. In principle, a conditional constraint can be implemented by checking the correspondent non conditional constraint for each scenario; however, the number of scenarios in a CTG grows exponentially with the number of branch nodes and a case by case check is not affordable in practice. Therefore, implementing conditional constraints requires an efficient tool to reason on CTG scenarios; this is provided by the BFG framework, described in Section 4.5.1.

We have defined and implemented the conditional version of the timetable constraint [ILO94] for cumulative resources described in the following section; other conditional constraints can be implemented by using the BFG framework and taking inspiration from existing filtering algorithms.

4.7.1 Timetable constraint

A family of filtering algorithms for cumulative resource constraints is based on timetables, data structures storing the worst case usage profile of a resource over time [ILO94]. While timetables for traditional resources are relatively simple and very efficient, computing the worst usage profile in presence of alternative scenarios and mutually exclusive activities is not trivial, since it varies in a non linear way; furthermore, every activity has its own resource view.

Suppose for instance we have the CTG in Figure 4.15A; tasks $t_0, \ldots, t_4$ and t_6 have already been scheduled: their start time and durations are reported in Figure 4.15B; all tasks require a single cumulative resource of capacity 3, and the requirements are reported next to each node in the graph. Tasks t_5 and t_7 have not been scheduled yet; t_5 is present only in scenario $\neg a$, where the resource usage profile is the first one reported in Figure 4.15B; on the other hand, t_7 is present only in scenario a, b, where the usage profile is the latter in Figure 4.15B. Therefore the resource view at a given time depends on the activity we are considering. In case an activity is present in more than one scenario, the worst case at each time instant has to be considered.

We introduce a new global timetable constraint for cumulative resources and conditional tasks in the non-preemptive case. The global constraint keeps a list of all known starting and ending points of activities (in particular their latest

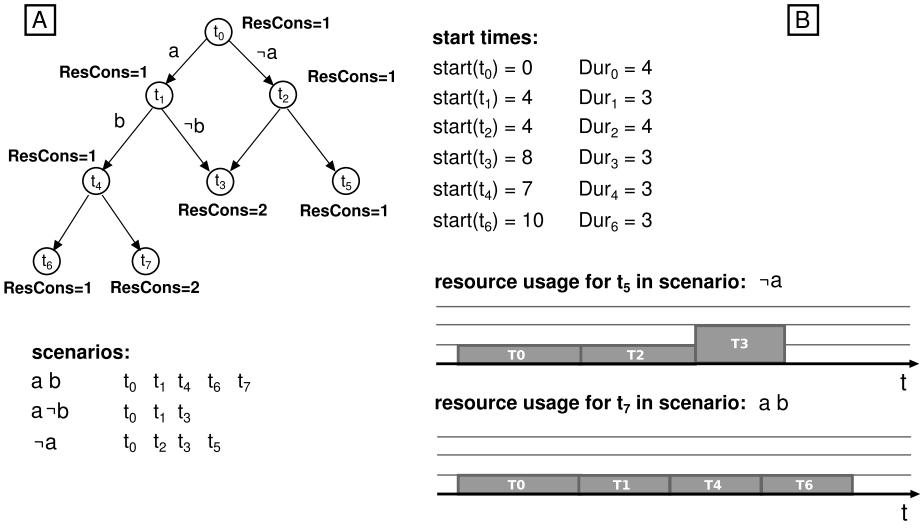


Figure 4.15: Capacity of a cumulative resource on a CTG

start times and earliest end times); given an activity t_i, if $LST(t_i) \leq EET(t_i)$ then the activity has an *obligatory part* from $LST(t_i)$ to $EET(t_i)$ contributing to the resource profile.

The filtering algorithm is described in Algorithm 5. All along the algorithm t_i is the target activity, the variable "*time*" represents the time point currently under exam and "$finish$" is finish line value (when it is reached the filtering is over); finally "$firstPStart$" represents the first time point where t_i can start and"*good*" is a flag whose value is false if the resource capacity is exceeded at the last examined time point.

Algorithm 5 keeps on scanning meaningful end points of all obligatory parts in the interval $[est(t_i), finish)$ until (line 4):

1. The resource capacity is exceeded in the current time point ($good = false$) and the current time point has gone beyond the latest start time of t_i (in this case the constraint fails)

2. The resource capacity is not exceeded in the current time point ($good = true$) and the finish line has been reached ($time \geq finish$)

Next, the resource usage is checked at the current time point (line 5); in case the capacity is exceeded this is recorded ($good = false$ at line 7) and the algorithm moves to the next *end* point of an obligatory part ($EET(t_j)$) in the hope the resource will be freed by that time. In case the capacity is not exceeded: (A) the current time point becomes suitable for the activity to start (line 10) and (B) the finish line is updated (line 11) to the current time value plus the duration of the activity; then the algorithm keeps on checking the starting time of obligatory parts (see line 14). If the finish line is reached without reporting a resource over-usage, then the start time of t_i can be updated (line 18).

Algorithm 5 treats the computation of the resource usage as a black box: the $resUsage(t_i, time)$ denotes the worst case usage at time $time$, as seen by task t_i; the worst case usage of a cumulative resource as seen by the current activity

Algorithm 5: Filtering algorithm for the conditional timetable constraint

```
1: let time = EST(t_i), finish = EET(t_i)
2: let firstPStart = time
3: let good = true
4: while ¬[(good = false ∧ time > LST(t_i)) ∨ (good = true ∧ time >= finish)] do
5:    if ResCons_i + resUsage(t_i, time) > resCapacity then
6:       let time = next EET(t_j)
7:       let good = false
8:    else
9:       if good = false then
10:         let firstPStart = time
11:         let finish = max(finish, time + Dur_i)
12:         let good = true
13:      end if
14:      let time = next LST(t_j)
15:   end if
16: end while
17: if good = true then
18:    let EST(t_i) = firstPStart
19: else
20:    fail
21: end if
```

can be computed efficiently via a backward visit, as described in Algorithm 3, on a BFG whose F-nodes are labeled with a weight value as follows.

To compute the worst case usage of a resource at time t we first have to "load" the requirement of each task t_i executing at time t (such that $LST(t_i) \leq t \leq EET(t_i)$) on each F-node F_j such that t_i belongs to the node inclusion label ($t_i \in i(F_j)$). For the computation of the maximum weight of a scenario each F and B-node has a single attribute w representing a weight value (in particular $A = [0, \infty)$). The *init* and *update* functions are defined as follows:

$$init_F(F_i) = \sum_{\substack{LST(t_j) \leq t \leq EET(t_j), \\ t_j \in i(F_i)}} ResCons_i$$

$$init_B(B_i) = 0$$

$$update_F(F_i, B_j) = \max(w(B_j), w(F_i))$$

$$update_B(B_i, F_j) = w(F_j) + w(B_i)$$

At the end of the process the weight of the root node is the worst case resource usage.

Basically Algorithm 3, instantiated as described, performs a backward visit of the BFG, summing up the weight of each child for every F-node (see $update_B$) and choosing for each B-node the maximum weight among those of its children (see $update_F$). As each outcome has to be processed once, the complexity is $O(n_o)$; loading a CTG task on an F-node has complexity $O(n_o)$. The timetable filtering algorithm Algorithm 5 in the worst case loads a CTG node and computes a weight for each task, hence n_t times; therefore the complexity of the filtering algorithm for the time window of a single task is $O(n_t(n_o + n_o)) = O(n_t n_o)$. This value can be reduced by caching the results and updating the data structures when a time window (say of task t_i) is modified; this is done by updating data on F-nodes and propagating the change backward along the

BFG; due to its tree like structure this is done in $O(\log(n_o))$ and the overall complexity is reduced to $O(n_t \log(n_o))$

4.8 Related work

This chapter is a substantially revised and extended version of two previous papers: in [LM06] we propose a similar framework for dealing with objective functions depending on task allocation in the field of embedded system design, while in [LM07] we face the makespan minimization problem. In the present chapter, we recall some ideas of these previous papers, but in addition we describe conditional constraints, we formalize the overall stochastic framework and we perform extensive evaluation.

The area where CTG allocation and scheduling has received more attention is most probably the one of embedded system design. In this context, Conditional Task Graphs represent a functional abstraction of embedded applications that should be optimally mapped onto multi core architectures (Multi Processor Systems on Chip - MPSoCs). The optimal allocation and schedule guarantees high performances for the entire life time of the system. The problem has been faced mainly with incomplete approaches: in particular, [PEP00] is one of the earliest works were CTGs are referred to as Conditional Process Graphs; there the focus is on minimizing the worst case completion time and a solution is provided by means of a branch outcome dependent "schedule table"; a list scheduling based heuristic is provided and inter tasks communications are taken into account as well. In [WAHE03a] 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 size of such a table can indeed be reasonable in real world applications. Another incomplete approach is described in [XW01] that proposes a heuristic algorithm for task allocation and scheduling based on the computation of mutual exclusion between tasks. Finally, [SK03] describes an incomplete algorithm for minimizing the energy consumption based on task ordering and task stretching.

To our knowledge, beside our previous work on CTG, the only complete approach to the CTG allocation and scheduling problem is proposed in [Kuc03] and is based on Constraint Programming. The solving algorithm used only scales up to small task graphs ($\sim$ 10 activities) and cumulative resources are not taken into account. Only a simple unary resource constraint is implemented in the chapter. Also, the expected value of the objective function is not taken into account.

Another complete CP based approach is described in [KW03] and targets low level instruction scheduling with Hierarchical Conditional Dependency Graphs (HCDG); conditional dependencies are modeled in HCDGs by introducing special nodes (guards), to condition the execution of each operation; complexity blowup is avoided by providing a single schedule where operations with mutually exclusive guards are allowed to execute at the same time step even if they access the same resource. We basically adopted the same approach to avoid scheduling each scenario independently. Mutual exclusions relations are listed in HCDGSs for each pair of tasks and are computed off line by checking compatibility of guard expressions, whereas in CTGs they are deduced from the

graph structure; note the in-search computation described in the chapter is just used to support speculative execution. Pairwise listing of exclusion relations is a more general approach, but lacks some nice properties which are necessary to efficiently handle non-unary capacity resources; in particular computing worst case usage of such a resource is a NP-complete problem if only pairwise mutual exclusions are known; in fact, in [KW03] only unary resources are taken into account.

An interesting framework where CTG allocation and scheduling can fit is the one presented in [BVLB07]. The framework is general taking into account also other forms of stochastic variables (say for example task durations) and can integrate three general families of techniques to cope with uncertainty: proactive techniques that use information about uncertainty to generate and solve a problem model; revision techniques that change decisions when it is relevant during execution and progressive techniques that solve the problem piece by piece on a gliding time horizon. Our work is less general and more focused on the efficient solution of a specific aspect of the framework, namely conditional branches and alternative activities.

Conditional Task Graph may arise in the context of conditional planning [PS92]. Basically, in conditional planning we have to check that each execution path (what we call scenario) is consistent with temporal constraints. For this purpose extensions to the traditional temporal constraint based reasoning have been proposed in [TVP03] and [TPH00]. However, these approaches do not take into account the presence of resources, which is conversely crucial in constraint based scheduling

Other graph structures similar to CTG have been considered in [KJF07, BF00]. These graphs contain the so called *optional activities* but their choice during execution is decided by the scheduler and is not based on the condition outcome. Basically, constraint based scheduling techniques should be extended to cope with these graphs, but no probability reasoning is required. For graphs with optional activities, an efficient unary resource constraint filtering algorithm is proposed in [VBC05].

Close in spirit to optional activities are Temporal Networks with Alternatives (TNA), introduced in [BC07]. TNA augment Simple Temporal Networks with alternative subgraphs, consisting of a "principal node" and several arcs to the same number of "branching nodes", from which the user has to choose one for run-time execution. Like in optional activities, the user is responsible for choosing a node; unlike in optional activities, exactly one branching node has to be selected. TNA do not allow early exits (as our approach does) and do not require any condition such as CFU. The follow-up work [BCS07] proposes a heuristic algorithm to identify equivalent nodes in a TNA, similarly to what we do with the BFG. Note however that F-nodes do not necessarily represent equivalence classes (think of the fact that a CTG node can be mapped to more than one F-node), but rather elementary groups of scenarios where a subset of tasks execute.

Speaking more generally, stochastic problems have been widely investigated both in the Operations Research community and in the Artificial Intelligence community.

Operations Research has extensively studied stochastic optimization. The main approaches can be grouped under three categories: sampling [AS02] consisting of approximating the expected value with its average value over a given

sample; the *l-shaped* method [LL93] which faces stochastic problems with recourse, i.e. featuring a second stage of decision variables, which can be fixed once the stochastic variables become known. The method is based on Benders Decomposition [Ben62]; the master problem is a deterministic problem for computing the first phase decision variables. The subproblem is a stochastic problem that assigns the second stage decision variables minimizing the average value of the objective function. A third method is based on branch and bound extended for dealing with stochastic variables, [NPR98].

In the field of stochastic optimization an important role is played by stochastic scheduling. This field is motivated by problems arising in systems where scarce resources must be allocated over time to activities with random features. The aim of stochastic scheduling problems is to come up with a policy that, say, prioritize over time activities awaiting service. Mainly three methodologies have been developed:

- Models for scheduling a batch of stochastic jobs, where the tasks to be scheduled are known but their processing time is random, with a known distribution, (see the seminal papers [Smi56, Rot66]).
- Multi armed bandit models [GJ74] that are concerned with the problem of optimally allocating effort over time to a collection of projects which change state in a random fashion.
- Queuing scheduling control models [CS71] that are concerned with the design of optimal service discipline, where the set of activities to be executed is not known in advance but arrives in a random fashion with a known distribution.

Temporal uncertainty has also been considered in the Artificial Intelligence community by extending Temporal Constraint Networks (TCSP) to allow contingent constraints [VF99] linking activities whose effective duration cannot be decided by the system but is provided by the external world. In these cases, the notion of consistency must be redefined in terms of controllability; intuitively, a network is controllable if it is consistent in any situation (i.e. any assignment of the whole set of contingent intervals) that may arise in the external world. Three levels of controllability must be distinguished, namely the strong, the weak and the dynamic one. We ensure in this work strong controllability since we enforce consistency in all scenarios.

The Constraint Programming community has recently faced stochastic problems: in [Wal02] 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 [TMW06] where an algorithm based on the concept of scenarios is proposed. In particular, the paper shows how to reduce the number of scenarios and still provide a reasonable approximation of the value of optimal solution. We will compare our approach to the one reported in this paper both in terms of efficiency and solution quality.

4.9 Experimental results

Our approach, referred to as conditional solver, has been implemented using the state of the art ILOG Cplex 11.0, Solver 6.3 and Scheduler 6.3. We tested the

approach on a number of instances representing several variants of a real world hardware design problem, where a multi task application (described by means of a CTG) has to be scheduled on a multiprocessor hardware platform. The problem features complex precedence relations (representing data communications), unary resources (the processors) and a single cumulative resource (modelling a shared communication channel whose capacity is its total bandwidth).

Instances for all problem variants are "realistic", meaning that they are randomly generated on the base of real world instances [GLM07]. We designed groups of experiments for two variants of the problem (described respectively in Sections 4.9.1 and 4.9.2), to evaluate the conditional timetable constraint and objective functions presented in this chapter and the performance of the BFG framework. Also, we compare our approach with a scenario based solver [TMW06] that explicitly considers all scenarios or a subset of them.

4.9.1 Bus traffic minimization problem

In the first problem variant hardware resources like processing elements and memory devices have to be allocated to tasks in order to minimize the expected bus traffic. Once all resources are assigned, tasks have to be scheduled and a specified global deadline must be met. The objective depends only on the allocation choices and counts two contributions: one depending on single task-resource assignments, one depending on pairs of task-resource assignments. Basically, the objective function captures both cases described in Section 4.6.1.

We faced the problem by means of Logic Based Benders' Decomposition [HO03] as explained in [LM06], where the master problem is the resource allocation and the subproblem is the computation of a feasible schedule.

We implemented a conditional solver based on BFG and a scenario based one [TMW06]. In the first case the stochastic objective function in the master problem is reduced to a deterministic expression where scenario probabilities are computed as described in Section 4.6.1; in the scheduling subproblem unary resources (the processors) are modeled with conditional disjunctive constraints, while the communication channel is considered a cumulative resource and is modeled with a conditional timetable constraint.

In the scenario based solver the objective function is a sum of an exponential number of linear terms, one for each scenario. Processors are modeled again with conditional disjunctive constraints, while for the communication channel a collection of discrete resources (one per scenario) is used. A simple scenario reduction technique is implemented, so that the solver can be configured to take into account only a portion of the most likely scenarios.

We generated 200 instances for this problem, ranging from 10 to 29 tasks, 8 to 33 arcs, which amounts to 26 to 95 activities in the scheduling subproblem (tasks and arcs are split into several activities). All instances satisfy Control Flow Uniqueness, and the number of scenario ranges from 1 to 72. The CTG generation process works by first building a deterministic Task Graph, and then randomly selecting some fork nodes to be turned into branches (provided CFU remains satisfied); outcome probabilities are chosen randomly according to a uniform distribution. The number of processors in the platform goes from 2 to 5. We ran experiments on a Pentium IV 2GHz with 512MB of RAM with a time limit of 900 seconds.

The first set of experiments, reported in Table 4.1, has the goal to test the

					time				
tasks	**arcs**	**acts**	**scens**	**proc**	**min**	**med**	**max**	**> TL**	**inf**
10-12	8-12	26-36	1-6	2-2	0.02	0.07	25.41	0	3
12-14	10-16	32-46	3-9	2-2	0.04	0.12	610.96	1	1
14-15	12-17	38-49	2-9	2-3	0.03	0.22	33.70	0	7
15-18	13-22	41-62	2-18	3-3	0.11	0.43	2.56	1	7
18-19	16-22	50-63	4-30	3-3	0.31	1.78	87.52	2	2
20-21	16-25	52-71	3-24	4-4	0.29	1.68	741.49	2	4
21-23	19-28	59-79	6-24	4-4	1.27	1.27	641.74	3	11
24-25	20-29	64-83	4-36	4-5	0.73	3.25	479.16	2	5
25-28	22-30	69-88	5-72	5-5	0.19	2.37	382.36	4	7
28-29	23-33	74-95	8-48	5-5	2.49	11.49	78.46	4	4

Table 4.1: Performance tests for the expected bus traffic minimization problem

performance of the conditional solver. Each row refers to a group of 20 instances and reports the minimum and maximum number of tasks, arcs, scheduling activities scenarios and processors (columns: tasks, arcs, acts, scens and proc); minimum (column: min), median (med) and maximum (max) computation time for the instances solved to optimality, included the time to perform pre-processing and build the BFG. Then the number of instances not solved within the time limit follows (> TL) and the number of infeasible instances (inf). As one can see the median computation time is pretty low and grows with the size of the instance, while its maximum has a more erratic behavior, influenced by the presence of uncommonly difficult instances. The number of timed-out instances intuitively grows with the size of the graph.

		S100			**S80**				**S50**			
scens	**proc**	$\frac{\mathbf{T}}{\mathbf{T_{cond}}}$	**> TL**	**inf**	$\frac{\mathbf{T}}{\mathbf{T_{cond}}}$	**> TL**	**inf**	$\frac{\mathbf{Z}}{\mathbf{Z_{cond}}}$	$\frac{\mathbf{T}}{\mathbf{T_{cond}}}$	**> TL**	**inf**	$\frac{\mathbf{Z}}{\mathbf{Z_{cond}}}$
1-3	2-3	1.30	1	3	1.31	1	3	1.00	0.85	1	3	0.86
3-4	2-4	0.98	2	5	1.22	2	5	1.00	0.57	1	1	0.74
4-6	2-5	1.33	2	4	0.85	1	4	0.97	1.04	0	4	0.69
6-6	2-5	1.06	0	5	1.31	0	5	0.96	0.83	0	4	0.78
6-8	2-4	0.96	1	6	1.15	1	6	0.93	1.20	0	6	0.69
8-9	2-5	0.89	6	5	1.00	6	5	0.88	0.48	5	4	0.62
9-12	2-5	1.24	3	6	1.11	4	6	0.96	0.79	3	6	0.79
12-12	3-5	1.38	3	8	0.96	4	8	0.91	0.97	4	8	0.73
12-24	3-5	0.98	2	6	0.96	3	6	0.87	0.59	3	5	0.79
24-72	3-5	1.21	3	2	1.04	3	2	0.81	0.89	3	2	0.64

Table 4.2: Performance tests for the expected bus traffic minimization problem

Then we compared the conditional solver we realized with a scenario based one for the same problem: the results for this second group of tests are shown in Table 4.2. Each row reports results for a group of 20 instances, for which it shows the minimum and maximum number of scenarios (column scens), the minimum and maximum number of processors (procs), and some data about the scenario based solver when 100% (S_{100}) 80% (S_{80}) and 50% (S_{50}) of the most likely scenarios are considered. In particular we report for each scenario

based solver the average solution time ratio with the conditional solver ($\frac{T}{T_{cond}}$, on the instances solved by both approaches), the number of timed out instances (> TL, not considered in the average time computation) and the number of infeasible instances (inf). For the S50 and S80 solvers also the average solution quality ratio is shown ($\frac{Z}{Z_{cond}}$). Note that in Table 4.2, instances are sorted by number of scenarios, rather than by size; as a consequence, the first rows do not necessarily refer to the smallest nor the easiest scheduling problems.

On average, the conditional solver improves the scenario based one by a 13% factor; this is not an impressive improvement. Also the improvement does not occur in all cases; the reason is that the computation time for this problem is dominated by that of finding an optimal resource allocation, and with regard to this subproblem the conditional approach only offers a more efficient way to build the *same* objective function expression. We expect to have much better results as the importance of the scheduling subproblem grows.

Note how the use of scenario reduction techniques speeds up the solution process for S_{80} and S_{50}, but introduces inaccuracies in the objective function value, which is lower than it would be (see column $\frac{Z}{Z_{cond}}$). Also, some infeasible instances are missed (the value of the "inf" column for S_{50} is lower than S_{100}).

4.9.2 Makespan minimization problem

In the second problem variant we consider the minimization of the expected makespan, that is the expected application completion time. This is indeed much more complex than the previous case, since this objective function depends on the scheduling related variables. We therefore chose to limit ourselves to computing an optimal schedule for a given resource allocation.

As we did for the previous problem variant, we implemented both a conditional and a scenario based solver. In the conditional solver the makespan computation is handled by the global constraint described in Section 4.6.2, whereas in the scenario based solver the makespan is the sum of the completion time of each possible scenario, weighted by the scenario probability (see expression 4.4). Processors and bus constraints are modeled as described in Section 4.9.1.

For this problem we generated 800 instances, ranging from 37 to 109 tasks, 2 to 5 "heads" (tasks with no predecessor), 3 to 11 "tails" (tasks with no successor), 1 to 135 scenarios. The number of processors (unary resources) ranges from 3 to 6. Again all instances satisfy the control flow uniqueness. We ran experiments with a time limit of 300 seconds; all tests were executed on a AMD Turion 64, 1.86 GHz.

We performed a first group of tests to evaluate the efficiency of the expected makespan conditional constraint and the quality of the solutions provided(in particular the amount of gain which can be achieved by minimizing the expected makespan compared to worst case based approaches); a second group of experiment was then performed to compare the performances of the conditional solver with the scenario-based one. Table 4.3 shows the results for the first group of tests; here we evaluate the performance of the solver using the conditional timetable constraint (referred to as C) and compare the quality of the computed schedules versus an identical model where the deterministic makespan is minimized (referred to as W). In this last case, no expected makespan constraint is used; the objective function is thus deterministic and amounts to minimizing the worst case makespan (hence the objective for the deterministic model will

be necessarily worse than the conditional one). The models for C and W are identical with every other regard (they both use conditional resource constraints and assign a fixed start time to every task). Each row identifies a group of 50 instances. For each group we report the minimum and maximum number of activities (acts), of scenarios (scens) and of unary resources (proc), the average solution time (T(C)), the average number of fails (F(C)) and the number of instances which could not be solved within the time limit (> TL) by the conditional solver.

In column C/W we report the makespan value ratio which shows an average improvement of 12% over the deterministic objective. The gain is around 16% if we consider only the instances where the makespan is actually improved (column stc C/W). The computing time of the two approaches is surprisingly roughly equivalent for all instances.

acts	**scens**	**proc**	**T(C)**	**F(C)**	**> TL**	$\frac{\mathbf{C}}{\mathbf{W}}$	**stc** $\frac{\mathbf{C}}{\mathbf{W}}$
37-45	1-2	3-4	1.54	3115	0	0.83	0.80
45-50	1-3	3-5	2.67	4943	0	0.88	0.84
50-54	1-3	3-5	9.00	17505	0	0.88	0.85
54-57	2-4	4-5	25.68	52949	1	0.88	0.85
57-60	1-6	4-5	29.78	77302	1	0.94	0.90
60-65	1-6	4-6	24.03	28514	0	0.85	0.80
65-69	2-8	4-6	32.12	47123	2	0.90	0.84
69-76	3-12	4-6	96.45	101800	14	0.86	0.82
76-81	1-20	5-6	144.67	134235	21	0.90	0.86
81-86	3-24	5-6	143.31	130561	17	0.84	0.75
86-93	2-36	5-6	165.74	119930	25	0.93	0.87
93-109	4-135	5-6	185.56	127321	28	0.93	0.87

Table 4.3: Performance tests

scens	**T(C)**	**> TL**	$\frac{\mathbf{T(S_{100})}}{\mathbf{T(C)}}$	**> TL**	$\frac{\mathbf{T(S_{80})}}{\mathbf{T(C)}}$	**> TL**	$\frac{\mathbf{S_{80}}}{\mathbf{C}}$	$\frac{\mathbf{T(S_{50})}}{\mathbf{T(C)}}$	**> TL**	$\frac{\mathbf{S_{50}}}{\mathbf{C}}$
1-2	41.00	5	22.60	5	22.66	5	1.00	0.58	3	0.77
2-3	66.02	8	19.85	10	19.93	10	1.00	1.69	7	0.80
3-4	43.80	5	35.05	8	35.12	8	1.00	9.19	5	0.79
4-5	49.94	6	73.75	9	73.63	9	1.00	57.03	8	0.80
5-6	66.39	9	48.74	12	16.64	12	0.98	0.77	8	0.82
6-6	51.26	5	6.52	8	6.11	8	0.96	41.99	8	0.80
6-8	38.85	5	82.21	11	71.09	9	0.98	84.41	3	0.80
8-9	57.78	9	66.32	10	63.70	10	0.98	26.76	9	0.85
9-12	52.96	5	89.52	13	86.97	13	0.98	40.43	6	0.85
12-14	117.93	17	45.60	22	43.02	22	0.97	37.35	18	0.84
14-20	95.74	11	32.62	22	31.85	21	0.99	28.76	15	0.90
20-135	178.88	24	66.19	37	65.56	37	1.00	22.09	35	0.912

Table 4.4: Comparison with scenario based solver

Table 4.4 compares the conditional model with a scenario based solver; we remind that in this second case the cumulative resource is implemented with one constraint per scenario and the expected makespan is expressed with the

declarative formula (4.4). In both models unary resources (processors) are implemented with conditional constraints.

Again, rows of Table 4.4 report average results for groups of 50 instances; instances are grouped and sorted by increasing number of scenarios; hence, once again the results on the first row do not necessarily refer to the easiest/smallest instances. The table reports the solution time of the conditional solver (T(C)) and the performance ratios w.r.t the scenario based solver with 100% (S_{100}), 80% (S_{80}) and 50% (S_{50}) of the most likely scenarios. The four columns "> TL" show the number of instances not solved within the time limit for each approach. Finally, columns S_{50}/C and S_{80}/C show the accuracy of the solution provided by S_{50} and S_{80} solvers.

As it can be seen the conditional model now outperforms the scenario based one by an average factor of 49.08. For this problem, in fact, the conditional approach provides a completely different and more efficient representation of the objective function, rather then just a more efficient procedure to build the same expression (as it was the case for the traffic minimization).

By reducing the number of considered scenarios the performance gap decreases; nevertheless, the conditional solver remains always better than S_{80}; it is outperformed by S_{50} when the number of scenarios is low, but the solution provided has an average 17% inaccuracy. Moreover, neither S_{50} nor S_{80} guarantee feasibility in all cases, since some scenarios are not considered at all in the solution.

4.10 Conclusion

CTG allocation and scheduling is a problem arising in many application areas that deserves a specific methodology for its efficient solution. We propose to use a data structure, called Branch/Fork graph, enabling efficient probabilistic reasoning. BFG and related algorithms can be used for extending traditional constraint to the conditional case and for the computation of the expected value of a given objective function.

The experimental results show that the conditional solver is effective in practice, and that it outperforms a scenario based solver for the same problem. The performance gap becomes significant when the makespan objective function is considered.

Current research is devoted to taking into account other problems where the stochastic variables are task durations or resource availability. Also, the application of CTG allocation and scheduling to the time prediction for business process management is a subject of our current research activity.

Chapter 5

A&S with Uncertain Durations

5.1 Introduction

This Chapter addresses Allocation and Scheduling problems when tasks cannot be assumed to have fixed durations; similarly to the previous chapter, the main motivation for the approach come from the need of predictable off-line scheduling techniques in the design processes of modern embedded systems.

Multicore platforms have become widespread in embedded computing. This trend is propelled by ever-growing computational demands of applications and by the increasing number of transistors-per-unit-area, under continuously tightening energy budgets dictated by technology and cost constraints [Mul08, JFL+07, YPB+08]. Effective multicore computing is however not only a technology issue, as we need to be able to make efficient usage of the large amount of computational resources that can be integrated on a chip. This has a cross-cutting impact on architecture design, resource allocation strategies, programming models.

Scalable performance is only one facet of the problem: predictability is another big challenge for embedded multicore platforms. Many embedded applications run under real-time constraints, i.e. deadlines that have to be met for any possible execution. This is the case of many safe-critical applications, such as in the automotive and the aircraft industries.

Predictability and computational efficiency are often conflicting objectives, as many performance enhancement techniques (such as caches, branch prediction...) aim at boosting expected execution time, without considering potentially adverse consequences on worst-case execution. Hence applications with strong predictability requirements often tend to under-utilize hardware resources [TW04] (e.g. forbidding or restricting the use of cache memories, limiting resource sharing, etc.). At the same time, many economical and practical reasons push for the integration of more and more application functionalities on a few powerful computing devices; thus, achieving both predictability and high average case performance becomes an inescapable requirement.

Chapter Objective The objective of this chapter is *predictable and efficient non-preemptive scheduling of multi-task applications with inter-task dependencies*. We focus on non-preemptive scheduling because it is widely utilized to schedule tasks on clustered domain-specific data-processors under the supervision of a general-purpose CPU [PAB+06, Pag07]. Non-preemptive schedul-

ing is known to be subject to *anomalies*[1] when tasks have dependencies and variable execution times [RWT+06]. Traditional anomaly-avoidance techniques [Liu00] rely on either synchronizing task switching on timer interrupts (which unfortunately implies preemption), or on delaying inter-task communication (or equivalently stretching execution time).

In our formulation, the target platform is described as a set of resources, namely execution clusters (i.e one or more tightly coupled processors with shared memory) and communication channels. An application is a set of dependent tasks that have to be periodically executed with a max-period constraint (i.e. deadline equal to period), modeled as a Task Graph. *A task execution is characterized by known min-max intervals and inter-task dependencies are associated with a known amount of data communication.* Min-max duration values can be extracted off-line by statical timing analysis, directly tackled in WP2.

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

The enumeration of possible conflict sets is a key component of the proposed algorithm. We deal with this fundamental step by means of two different techniques: the first one is based on Constraint Programming; the second one leverages Operations Research techniques to find conflicts in a fraction of time compared to the first method; however, the found conflict tends to be less critical. While the first method has exponential worst case complexity, the second one is plynomial.

Outline The rest of the chapter is organized as follows: Section 5.3.1 and Section 5.3.2 present some related work; additionally, Section 5.3.2 introduces some basic Precedence Constraint Posting Concepts. Section 5.4 gives the problem definition, while Section 5.5 describes the proposed heuristic resource allocation method. Section 5.6 describes the core of the work, i.e. the predictable scheduling algorithm; two alternative approaches to perform a key algorithmic step (conflict set enumeration) are given in Sections 5.6.2.1 and 5.6.2.2. Finally, Section 5.7 reports experimental results and Section 5.8 provides conclusions and suggests some possible future work.

Publications The work at the based of this chapter has been partly published to internation conferences in [LM09, LMB09]. The submission of an extended version to an international journal is planned for the near future.

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

5.2 Overview of the approach

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

Once the task-to-resource allocation step is done, one has to compute a schedule, guarantee to meet all the hard deadline constraint for every combination of task durations. From a computational point of view, this is an instances of the Resource Constrained Project Scheduling Problem (RCPSP) with variable durations. RCPSP aims to schedule a set of activities subject to precedence constraints and the limited availability of resources.

Additionally, we allow minimum and maximum time lags to be specified on each precedence constraint in the scheduling step; namely, the time distances between the end of a task and the beginning of its successor is forced to be in a constrained interval with user-specified min and max. This is the case, for example, of Dynamic Voltage Scaling (DVS) platforms, where tasks running on the same processor at different frequencies may require a minimum time distance constraint to allow frequency switching. Note that in the considered case study time lags are only partially used.

We adopt a Precedence Constraint Posting approach (see [PCOS07] and Section 2.3.3): the solution we provide is an augmented Task Graph; this is the original project graph plus a *fixed* set of new precedence constraints, such that all possible resource conflicts are cleared and a consistent assignment of start times can be computed by the on-line scheduler for whatever combination of activity durations. The main advantage of a PCP approach is to retain flexibility: each task starts at run-time as soon as all its predecessors are over; in case the predecessors end earlier, the task seamless starts earlier. This is not the case for many common approaches to deal with duration uncertainty (e.g. stretching task durations in case of early end, or time triggered scheduling).

5.3 Related work

5.3.1 Scheduling with variable durations

Non-preemptive real-time scheduling on multiprocessors has been studied extensively in the past. We focus here on partitioned scheduling. We can cluster previously proposed approaches in two broad classes, namely: on-line and off-line.

On-line Scheduling with Uncertain Durations 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 allocation and scheduling algorithms must be extremely fast, typically constant time or low-degree polynomial time. Given that multiprocessor allocation and scheduling on bounded resources is NP-Hard [GJ+79], obviously, on-line approaches cannot guarantee optimality, but they focus on safe acceptance tests,

i.e. a schedulable task set may be rejected but a non-schedulable task set will never be accepted.

An excellent and comprehensive description of the on-line approach is given in [Liu00], and can be summarized as follows. When an application (i.e. a task graph), enters the system, it is first partitioned on processor clusters using a fast heuristic assignment (e.g. greedy first-fit bin packing). Then schedulability is assessed on a processor-by-processor basis. First, local task deadlines are assigned, based on a deadline assignment heuristic (e.g. ultimate deadline), then priorities are assigned to the tasks, and finally a polynomial-time schedulability test is performed to verify that deadlines can be met. It is important to stress that a given task graph can fail the schedulability test even if its execution would actually meet the deadline. This is not only because the test is conservative, but also because several heuristic, potentially sub-optimal decisions have been taken before the test which could lead to infeasibility even if a feasible schedule does exist. One way to improve the accuracy of the schedulability test is to modify synchronization between dependent tasks by inserting idle times (using techniques such as the release-guard protocol [SL96]) that facilitate worst-case delay analysis. Recent work has focused on techniques for improving allocation and reducing the likelihood of failing schedulability tests even without inserting idle times [FB06, Bar06].

Off-line approaches with Uncertain Durations Off-line approaches, like the one proposed in this chapter, assume the knowledge of the task graph and its characteristics before execution starts. Hence, they can afford to spend significant computational effort in analyzing and optimizing allocation and scheduling (e.g. by applying advance Timing Analysis techniques). In this field, we distinguish two main sub-categories. One sub-category assumes a fixed, deterministic, execution time for all tasks. Even though the allocation and scheduling problems remain NP-Hard, efficient complete (exact) algorithms are known (see for instance [RGB+08] and references therein) that work well in practice, and many incomplete (heuristic) approaches have been proposed as well for very large problems that exceed the capabilities of complete solvers [ZTL+03]. These approaches can also be used in the case of variable execution times, but we need to force determinism: at run-time, task execution can be stretched artificially (e.g. using timeouts or idle loops) to always complete as in the worst case. This eliminates scheduling anomalies, but implies significant platform underutilization and unexploitable idleness [Liu00]. Alternatively, powerful formal analysis techniques can be used to test the absence of scheduling anomalies [BHM08]. However, if an anomaly is detected, its removal is left to the software designer. This can be a daunting task.

The second sub-category is robust scheduling. The main idea here is to build an a priori schedule with some degree of robustness. This can be achieved by modeling time variability (for example by means of min-max intervals), and inserting redundant resources [Gho96], redundant activities [GMM95] or more generally slack time to absorb disruptions; it is also common to add time redundancy by stretching task durations [CF99]. A different approach is based on building a backup schedule for the most likely disruptions [DBS94]. Robust scheduling has been extensively investigated in the process scheduling community [LI08]. In this chapter we leverage a technique called Precedence Con-

straint Posting (PCP) [LG95, Lab05, PCOS07] devised for Resource Constraint Project Scheduling Problem (RCPSP) to obtain a robust schedule by preventing resource conflicts with added precedence constraints. Compared to existing PCP approaches, we directly exploit known time bounds on task durations and guarantee feasibility for every possible scenario; existing approaches assume fixed durations and focus on adding some degree of flexibility to the schedule, without giving any guarantee.

5.3.2 PCP Background

We adopt a Precedence Constraint Posting approach (PCP, see [PCOS07] and Section 2.3.3); in PCP, possible resource conflicts are resolved off-line by adding a fixed set of precedence constraints between the involved activities. The resulting augmented graph defines a set of possible schedules, rather than a schedule in particular. The actual schedule will depend on the run-time duration of each task and is produced by the on-line scheduler.

Central to any PCP approach is the definition of Minimal Conflict Set (MCS), making its first appearance in [IR83a] (1983), where a branching scheme based on resolution of the so called "minimal forbidden sets" is first proposed. A MCS is a set of tasks t_i collectively overusing one of the resources (e.g. a processor cluster, communication channel...) and such that the removal of a single task from the set wipes out the conflict as well; additionally, the tasks must have the possibility to overlap in time. Following [Lab05], we formally define a MCS for a resource r_k as a set of tasks such that:

1. $\sum_{t_i \in MCS} req(t_i, r_k) > C_k$
2. $\forall t_i \in MCS \ : \ \sum_{t_j \in MCS \setminus \{t_i\}} req(t_j, r_k) \leq C_k$
3. $\forall t_i, t_j \in MCS$ with $i < j \ : \ t_i \preceq t_j \wedge t_j \preceq t_i$ is consistent with current state of the schedule, where $t_i \preceq t_j$ denotes that t_i can run before t_j.

where r_k denotes a resource specific resource, C_k is the respective capacity, $req(t_i, r_k)$ is the amount of resource r_k required by task t_i. Condition 1 requires the set to be a conflict, condition 2 enforces minimality and condition 3 requires activities to be possibly overlapping. A MCS can be *resolved* by posting a single precedence constraint between any pair of activities in the set; complete search can thus be performed by using MCS as choice points and posting on each branch a precedence constraint (also referred to as *resolver*). This is the case of many PCP based works: for example [Lab05] makes use of complete search to detect MCS and proposes a heuristic to rank possible resolvers. Other branch and bound approaches based on posting precedence constraints to resolve MCS are reported in [RH98], where minimum and maximum time lags are also considered; variable activity durations are not taken into account in any of these works.

MCS: Enumeration and Execution Policies One of the key difficulties with complete PCP approaches is that detecting MCS by complete enumeration can be time consuming, as their number is in general exponential in the size of the Task Graph. A possible way to overcome this issue is to only consider MCS which can occur given for a specified "execution policy": this is the case for example of the already cited work [RH98] where an earliest start policy (start

every activity as soon as possible) is implicitly assumed; note that with this approach the number of MCS to be considered remains exponential, despite it is effectively reduced.

Having an earliest start policy [MRW84, MRW85] is also a basic assumption for many stochastic RCPSP approaches; here variable task durations are explicitly considered and usually modeled as stochastic variables with known distribution. Most works in this area assume the objective function is to minimize the makespan and focus on computing restrictions of earliest start policies to resolve resource conflicts on-line. This is the case of preselective policies, introduced in [IR83a, IR83b], and a priori selecting for each minimal conflict set an activity to be delayed in case the conflict occurs. A major drawback with this approach is the need to enumerate all possible MCS [Sto01, Sto00]. To overcome this limitation, [MS00] introduces so called linear preselective policies: in this case a fixed priority is assigned to each activity such that when a conflict is about to occur at run time, the lowest priority activity is always delayed. To the best of the authors knowledge, no stochastic RCPSP approach has considered minimum and maximum time lags so far. Another way to fix the issue of efficient conflict detection is to drop completeness and resort to heuristic methods; see for example the method described in [PCOS07], which also incorporates time reasoning by representing the target project as a simple temporal network; many feature of this efficient and expressive model are leveraged by our approach. Finally, for excellent overviews about methods for the RCPSP problem and dealing with uncertainty in scheduling see [BDM$^+$99, BD02, HL05b].

5.4 Problem Definition

5.4.1 The Target Platform

A common template for embedded MPSoCs features an advanced general-purpose CPU with memory hierarchy and OS support and an array of programmable processors, also called *accelerators*, for computationally intensive tasks. accelerators have limited or no support for preemption and feature private memory subsystems [PAB$^+$06, Pag07, BEA$^+$08]. Explicit communication among accelerators is efficiently supported (e.g. with specialized instructions or DMA support), but uniform shared memory abstraction is either not supported or it implies significant overheads [2].

We model the heterogeneous MPSOC as shown in Figure 5.1A (a). All shaded blocks are resources that will be allocated and scheduled. For the sake of clarity, all empty blocks are not modeled in the problem formulation. accelerators ($CL1, \ldots, CLN$) are modeled as clusters of one or more execution engines. A cluster with C execution engines models an accelerator with multiple execution pipelines (e.g. a clustered VLIW) or a multi-threaded processor with a fixed number of hardware-supported threads. Note that the case of $C = 1$ (single-instruction-stream accelerator) is handled without loss of generality. From a broader perspective, one may say a cluster is modeled as a resource with finite capacity C; each task requires exactly 1 unit of such a resource. The master CPU is left implicit as its function is mostly related to orchestration.

[2] it can be emulated in software by propagating via copy operations all private memory modifications onto external shared memory [PAB$^+$06]

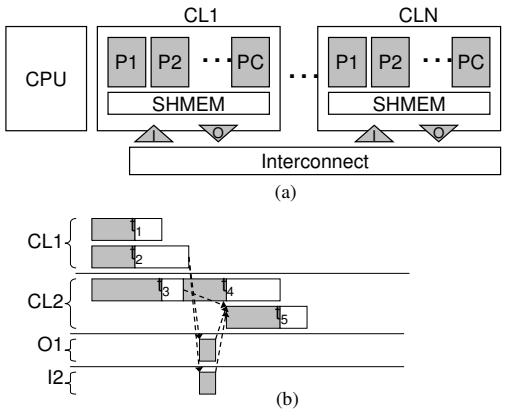


Figure 5.1: The target platform model

When two dependent tasks are allocated onto the same cluster, their communication is implicitly handled via shared variables on local memory. On the contrary, communication between tasks allocated on two different clusters is explicitly modeled as an extra task which uses a known fraction $b_{i,j}$ of the total bandwidth[3] 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. More in general one may say that input and output ports are resources with finite capacity equal to the bandwidth; a data communication requires b_{ij} units of the full bandwidth.

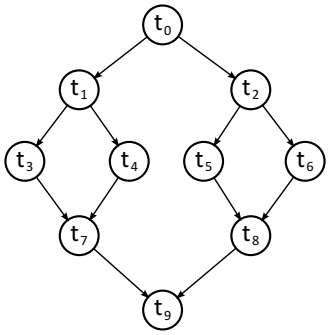


Figure 5.2: A structured Task Graph

5.4.2 The Input Application

We assume the input application for the mapping and scheduling algorithm is described as a Task Graph $\langle T, A \rangle$; T is a set of tasks t_i, representing processes

[3]Note that a previous study [BBGM05a] showed how modeling the communication overhead as a fraction of the bandwidth has bounded impact on task durations as long as less than 60% of the available bandwidth is used.

or any type of sequential application subunit; A is a set of directed edges $a_h = (t_i, t_j)$ representing precedence relations due to data communications.

The Gantt chart in Figure 5.1(b) shows a simple example of a scheduled Task Graph. We assume Tasks t_i do not have a fixed duration, but they are described by duration intervals $[d_i, D_i]$. In the figure, the shortest duration corresponds to the shaded part of the rectangles, while the longest duration is represented by the empty part following the shaded part. Tasks t_1, t_2 are scheduled on cluster $CL1$, while task t_3, t_4, t_5 are allocated and scheduled on cluster $CL2$. Both clusters have two execution engines. Figure 5.2 show a simple structured Task Grah, representing a software radio application. The graph will be used in all the examples throughout the section; for sake of simplicity, we assume all tasks in Figure 5.2 are homogeneous and in particular have the same minimum and maximum duration d_i and D_i, respectively with value 1 and 2.

Dependencies are represented by arrows. Intra-cluster communications are assumed to occur instantaneously, while inter-cluster communication are carried out by a DMA engine and require some time, bounded by a minimum value d_{ij} and a maximum value D_{ij}. Note that inter-cluster communication is explicitly represented by additional tasks executing at the same time on the input ($I2$) and output ($O1$) ports of the clusters involved in the communication. Such additional task are not present in the original application graph, but are built as a by product of the resource allocation step (see Section 5.5). In the sample graph in Figure 5.2 we assume all arcs are homogeneous, and in particular have the same minimum and maximum duration d_{ij} and D_{ij} (with value 1 and 2) and bandwidth requirement $b_{ij} = 1$.

Time window constraints exist on the start and end time of each task; specifically, we assume every task has to start after a specified *release time* (rs_i) and to end before a specified *deadline* (dl_i). Without loss of generality, we assume there is a single source task (t_0) with no ingoing arcs and a single sink task (t_{n-1}) with no outgoing arcs. Finally, precedence relations may have associated minimum and maximum time lags; namely, a precedence relation (t_i, t_j) may be annotated by values d_{ij}, D_{ij} such that the time distance between the end of t_i and the start of t_j at run-time cannot be lower than d_{ij} nor higher than D_{ij}.

5.4.3 Problem Statement

The problem consists in finding a *allocation* and a *schedule* for the input application on the target platform. In deeper detail, an allocation consists of a mapping of each task to a cluster; we denote as $pe(t_i)$ the cluster (Processing Element – PE) a task t_i is assigned to.

A schedule is a collection of additional precedence relations defining an augmented graph. The additional precedence relations are sufficient to prevent any resource overusage (either cluster or communication port) at run-time, provided they are respected by the on-line scheduler.

From a technical point of view, at run-time an allocation & schedule (sometimes referred to as partitioned schedule) is described simply by a set of tasks allocated on each processor and the triggering guards of each task, which are identifiers of all predecessor tasks in the augmented graph. The run-time semantic is straightforward: whenever a task terminates execution, a new task is started among those with completely released triggering guards (any tie-breaker rule can be used, e.g. EDF). If no task has fully released guards, the processor

remains idle until a guard change triggered by a communication. Note that a single Gantt chart like the one in Figure 5.1 corresponds to a huge set of actual executions, one for each combination of different durations for all the tasks.

5.5 Resource Allocation Step

Before scheduling, a heuristic allocation of the available resources to tasks is computed; in particular, tasks are mapped to clusters by means of a state of the art multilevel graph partitioning algorithm [KK99].

The Graph Partioning Algorithm Tasks in the graph are split in a number of sets equal to the number of clusters, trying:

1. to balance the computational load and
2. to minimize inter-cluster communication

In detail, let $wgt(t_i)$ bet a weight value associated task t_i, and let a second type of weight $wgt(a_h)$ be associated to each arc $a_h \in A$; finally, let P_0, P_1 the returned partition (assuming the number of clusters is 2). The objective of the algorithm is minimizing the *edge-cut*, given by:

$$edgecut(P_0, P_1) = \sum_{\substack{a_h=(t_i,t_j) \\ t_i \in P_0, t_j \in P_1}} wgt(a_h)$$

the partition P_0, P_1 must satisfy a balancing constraint; namely, given an input parameter $\alpha \in (0, 0.5)$, the algorithm approximatively ensures:

$$(1-\alpha) \cdot \sum_{t_j \in P_j} \leq \sum_{t_i \in P_i} \leq (1+\alpha) \cdot \sum_{t_j \in P_j}$$

The graph partitioning approach can be applied to hardware resource allocation with nice results, by assigning to each task as weight the average computation time (i.e. $.5 \cdot (D_i - d_i)$) and to each arc (t_i, t_j)) the bandwidth requirement b_{ij} corresponding to the communication. Consequently, the balancing constraints naturally matches the need to partition the computation workload over different clusters, while minimizing the edge cut (i.e. the inter cluster communication in the case of hardware mapping) contains the usage of communication resources.

In detail, we require the difference between the maximum and minimum workload to be within a 6% bound ($\alpha = 0.06$). The method is designed for homogeneous computational resources, but generalizes to heterogeneous ones with proper modifications. Figure 5.3A shows a likely outcome of the partitioning of the Task Graph from Figure 5.2 on a two cluster platform; the presented cuts splits the tasks T in two equal weight sets (namely $\{t_0, t_1, t_3, t_4, t_7\}$ and $\{t_2, t_5, t_6, t_8, t_9\}$) and minimizes the number of arcs connecting the two components of the partition.

Output of the Algorithm and Graph Transformation The output of the multilevel partitioning algorithm is a collection of task sets, one per each cluster. Based on this information, the original task graph is transformed by

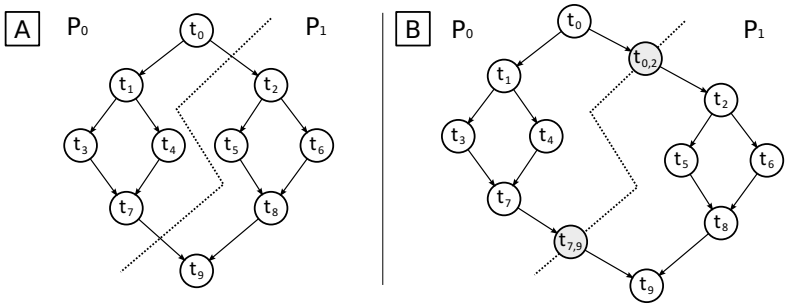


Figure 5.3: A) Outcome of the partitioning – B) Transformed Graph

adding an extra tasks t', t for each inter-cluster communication (t_i, t_j); task t' represents the data transfer and uses b_{ij} units of the *Output Port* of cluster $pe(t_i)$ and of the the *Input Port* of cluster $pe(t_j)$; analogously to the other tasks in the graph, t' has variable duration corresponding to that of the inter-cluster communication, i.e. in the interval d_{ij}, D_{ij}. The transformed graph is the actual input for the subsequent scheduling step. Figure 5.3B shows the transformed graph corresponding to the partition in Figure 5.3A; note that two extra activities ($t_{0,2}$ and $t_{7,9}$) have been added to model the inter cluster communications.

5.6 Scheduling Step

We propose a PCP based approach to compute a schedule (i.e. an augmented Task Graph). Once the allocation step is over and a transformed graph is available, the input application is described by a collection of tasks, each requiring one or more unit of some finite capacity resources. Arcs in the graph simply represent precedence relations with no associated execution, as intra-cluster communications are considered instantaneous and inter-cluster communications are modeled by extra tasks.

The problem of scheduling a set of tasks with arbitrary precedence relations on a set of finite capacity resources is known in the literature as Resource Constrained Project Scheduling Problem (RCPSP – see Section 2.4). Here in particular we tackle the RCPSP with minimum and maximum time lags on the precedence relations and variable, bounded, task durations.

The scheduling method we propose performs complete search branching on MCS. The MCS to be used to open a choice point and branch can be selected in one of two ways: (I) by solving a Constraint Programming model or (II) use of an efficient, polynomial time, MCS detection procedure based on the solution of a minimum flow problem (related, but substantially different from the method outlined in [Mus02]). In general, the method used for MCS selection represents a first major contribution w.r.t. similar approaches like [Lab05].

Similarly to [PCOS07], our approach incorporates an expressive and efficient time model, taking care of consistency checking of time windows constraints and enabling the detection of possibly overlapping activities. Our second main contribution is the extension of such a time model in order to provide efficient time reasoning with variable durations and enable constant time consis-

tency/overlapping check.

5.6.1 The time model

The time model we devised relates to Simple Temporal Networks with Uncertainty (STNU, [VF99]). The model itself consists of a set $\mathcal{T}$ of temporal *events* or time points τ_i with associated time windows, connected by directional binary constraints so as to form a directed graph. Such constraints are in either of the two forms:

1. $\tau_i \xrightarrow{[\delta_{ij},\Delta_{ij}]} \tau_j$ (*free* constraint, with STNU terminology), meaning that a value d in the interval $[\delta_{ij}, \Delta_{ij}]$ must exist such that $\tau_j = \tau_i + d$; equivalently we can state that $\delta_{ij} \leq \tau_j - \tau_i \leq \Delta_{ij}$.

2. $\tau_i \xrightarrow{[\delta_{ij}:\Delta_{ij}]} \tau_j$ (*contingent* constraint, with STNU terminology), meaning that τ_i and τ_j must have enough flexibility to let $\tau_j = \tau_i + d$ hold for each value $d \in [\delta_{ij}, \Delta_{ij}]$.

Both δ_{ij} and Δ_{ij} are assumed to be non-negative; therefore, unlike in STN, a constraint cannot be reverted. The above elements are sufficient to represent a variety of time constraints; in particular, an instance of the problem at hand, can be modeled by:

1. introducing two events σ_i, η_i for the start and the end of each activity, respectively with time windows $[st_i, \infty]$ and $[0, dl_i]$;

2. adding a contingent precedence constraint $\sigma_i \xrightarrow{[d_i:D_i]} \eta_i$ for each activity;

3. adding a free precedence constraint $\eta_i \xrightarrow{[0,\infty]} \sigma_j$ for each arc in the project graph.

The precedence constraints in the time model define a directed graph $\langle \mathcal{T}, \mathcal{C} \rangle$, referred to as *time graph*. Here $\mathcal{T}$ is the set of time points and $\mathcal{C}$ the set of temporal constraints. In the following, we will write $\tau_i \preceq \tau_j$ if a chain of precedence constraints connects τ_i to τ_j (we write $\tau_i \prec \tau_j$ if the precedence relation is strict). Testing $\tau_i \preceq \tau_j$ can be done in costant time by keeping the transitive closure of the time graph (and this is with cubic complexity along a branch on the search tree).

Figure 5.4 shows the time graph for the TG in Figure 5.2; each event σ_i, η_i is represented as a black circle. The solid edges are contingent constraint, with the respective lags explicitly reported ($[1:2]$ for all the nodes). Dotted edges represent free constraints; the time lags in this case are not reported and always equal to $[0, \infty]$.

Bound Consistency for the Temporal Network We rely on CP and constraint propagation for dynamic controllability checking, rather than using a specialized algorithm like in [MMV01, MM05]. In particular, we are interested in maintaining for each time point τ_i four different bounds, namely: the start (s_p) and the end (e_p) of the time region where the associated event can occur (*possible region*); and the start (s_o) and the end (e_o) of a time region (*obligatory region*) such that, if τ_i is forced to occur out of there, dynamic controllability is

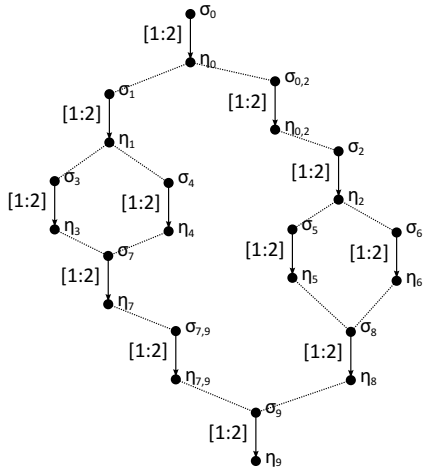


Figure 5.4: Time model for the graph in Figure 5.2

compromised. An alternative view is that the obligatory region keeps track of the amount of flexibility left to a time point.

Figure 5.5 shows as example of such bounds: s_p and e_p delimit the region where each τ_i *can* occur at run: for example τ_1 can first occur at time 10, if τ_0 occurs at 0 and has run time 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, note that τ_2 cannot occur after time 60, or there would be a value $d \in [10, 20]$ with no support in the time window of τ_3; conversely, τ_1 can occur as late as time 50, since there is *at least* a value $d \in [10, 20]$ with a support in the time window of τ_2. Consider now bounds on the obligatory region: note that if (for instance) 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 span between τ_0 and τ_1. Similarly, τ_2 cannot be forced to occur later than time 60 or there would be a value $d \in [10, 20]$ such that the precedence constraint between τ_2 and τ_3 cannot be satisfied.

In general, bounds on the possible region (s_p, e_p) are needed to enable efficient dynamic controllability checking (if the time window of every time point is not empty) and constant time overlapping checking; bounds on the obligatory region (s_o, e_o) are a novel contribution and let one check in constant

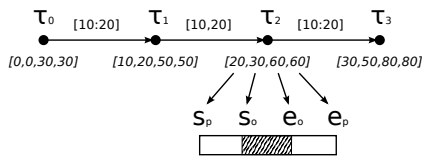


Figure 5.5: Time bounds for a simple time point network

time whether a new precedence constraint can be added without compromising dynamic controllability. With this trick it is possible for example to remove inconsistent ordering relations between tasks that cannot be trivially inferred starting from the possible region.

CP-FD Implementation of the Temporal Model We maintain the described bounds by introducing two CP integer variables $\mathtt{T^m}, \mathtt{T^M}$ for each time point, such that $\min(\mathtt{T^m})$ marks the start of the possible region and $\max(\mathtt{T^m})$ tells how far this start can be pushed forward; similarly, $\max(\mathtt{T^M})$ marks the end of the possible region and $\min(\mathtt{T^M})$ tells how far this end can be pulled back; consequently, we map s_p to $\min(\mathtt{T^m})$, e_o to $\max(\mathtt{T^m})$, s_o to $\min(\mathtt{T^M})$, e_p to $\max(\mathtt{T^M})$; then we post for each τ_i: $\mathtt{T^m_i} \leq \mathtt{T^M_i}$. This ensures $s_p \leq s_o$, $e_o \leq e_p$ and triggers a failure whenever s_p is pushed beyond e_o (the time point is forced to occur after the obligatory region) or e_p is pulled before s_o (the time point is forced to occur before the obligatory region). Note that if dynamic controllability has to be kept, the case $s_p > e_o$ never occurs. For each constraint $\tau_i \xrightarrow{[\delta,\Delta]} \tau_j$, we perform the following filtering:

$$\delta \leq \max(\mathtt{T^m_j}) - \max(\mathtt{T^m_i}) \leq \Delta \qquad \delta \leq \max(\mathtt{T^M_j})\, \max(\mathtt{T^M_i}) \leq \Delta$$
$$\delta \leq \min(\mathtt{T^m_j}) - \min(\mathtt{T^m_i}) \leq \Delta \qquad \delta \leq \min(\mathtt{T^M_j})\, \min(\mathtt{T^M_i}) \leq \Delta$$

which can be done by posting $\delta \leq \mathtt{T^m_j} - \mathtt{T^m_i} \leq \Delta$ and $\delta \leq \mathtt{T^M_j} - \mathtt{T^M_i} \leq \Delta$. The rationale behind the filtering rules can be explained by looking at Figure 5.6A. For example $s_p(\tau_j)$ cannot be less than δ time units away from $s_p(\tau_i)$, or no time distance value d exists in $[\delta, \Delta]$ such that $\tau_j = \tau_i + d$; so $\min(\mathtt{T^m_j})$ can be updated to $\min(\mathtt{T^m_i}) + \delta$, in case it is less than that value: this is depicted in the figure with an arrow going from $s_p(\tau_i)$ to $s_p(\tau_j)$. By reasoning in the same fashion one can derive all other filtering rules. Similarly, for each contingent constraint $\tau_i \xrightarrow{[\delta:\Delta]} \tau_j$ we perform the following filtering:

$$\max(\mathtt{T^m_i}) + \Delta = \max(\mathtt{T^m_j}) \qquad \max(\mathtt{T^M_i}) + \Delta = \max(\mathtt{T^M_j})$$
$$\min(\mathtt{T^m_i}) + \delta = \min(\mathtt{T^m_j}) \qquad \min(\mathtt{T^M_i}) + \Delta = \min(\mathtt{T^M_j})$$

As in the previous case, figure 5.6B gives a pictorial intuition of the rationale behind the rules. Now, neither $s_p(\tau_j)$ can be closer than δ time units to $s_p(\tau_i)$, nor $s_p(\tau_i)$ can be farther than δ units from $s_p(\tau_j)$; otherwise there would exist

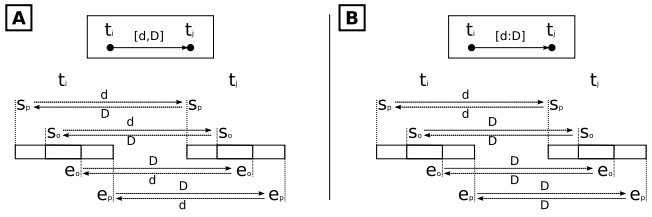


Figure 5.6: (A) Filtering rules for free constraints; (B) Filtering rules for contingent constraints

a d value in $[\delta, \Delta]$ such that τ_j could not be equal to $\tau_i + d$. This explains the second filtering rule on the leftmost column. By reasoning similarly one can derive all other rules. The described filtering runs in polynomial time and is *sufficient* to enforce consistency on the network, with the support of the on-line scheduler.

5.6.2 MCS Based Search

The solver architecture is depicted in figure 5.6.2. It is organized as two main components: (1) a Constraint Programming (CP) scheduler, performing search by adding precedence constraints (Precedence model), which are translated into task time bounds by the Time model; (2) a Minimum Conflict Set (MCS) finder that exploits knowledge on resources as described later, whereas the scheduler has no direct notion of resource constraints; the reason is that usual CP algorithms connecting task time bounds and resources are designed to check consistency for at least one duration value, rather than for all durations. We recall that a MCS is a minimal set of tasks which would cause a resource over-usage in case they overlap; here minimal means that by removing any task from the set the resource usage is no longer jeopardized. Due to its minimality, a MCS is resolved by posting a single precedence constraint between two tasks belonging to the set.

MCS Based Tree Search The search engine of the scheduler iteratively detects and solves MCSs, in tree search fashion. At each step, in first place a MCS is identifies; then the solver selects a pair of tasks $t_i, t_j \in MCS$ and opens a choice point. Let $RS = (t_{i_0}, t_{j_0}), \ldots (t_{i_{m-1}}, t_{j_{m-1}})$ be the list of pairs of nodes in the set such that a precedence constraints can be posted. Then the choice point can be recursively expressed as:

$$CP(RS) = \begin{cases} \texttt{post}(t_{i_0}, t_{j_0}) \text{ if } |RS = 1| \\ \texttt{post}(t_{i_0}, t_{j_0}) \vee [\texttt{forbid}(t_{i_0}, t_{j_0}) \wedge CP(RS \setminus (t_{i_0}, t_{j_0}))] \end{cases}$$

where (t_{i_0}, t_{j_0}) always denotes the first pair in the processed set. The operation $\texttt{post}(t_{i_0}, t_{j_0})$ amounts to adding the constraint $\eta_i \xrightarrow{[0,\infty]} \sigma_j$ in the time model, and $\texttt{forbid}(t_{i_0}, t_{j_0})$ consists in adding $\sigma_j \xrightarrow{[1,\infty]} \eta_i$ (strict precedence relation). If the addition/forbidding of any precedence constraint succeeds, the solver looks for one more MCS and the procedure is reiterated, until no more possible conflict

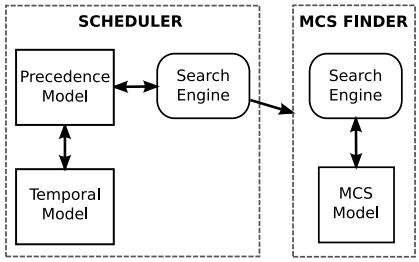


Figure 5.7: Solver architecture

exists. In case no precedence relation can be posted or forbidden on the whole RS set, the solver fails (the problem contains an unsolvable MCS).

Prior to actually building the choice point, all precedence constraints could in principle be sorted according to some heuristic criterion; note however this is not done in the current implementation: introducing some score to rank precedence constraints (e.g. the one proposed in [Lab05], based on preserved search space) is part of planned future work. The behevior of the solver strongly depends on the MCS finding procedure, for which we provide two alternative method, each one with advantages and drawbacks. The two methods are described in the following two sections.

5.6.2.1 Finding MCS via CP Based Complete Search

The first proposed MCS finding method is based on complete search; in detail the MCS detection problem is stated as a Constraint Satisfaction Problem, whose decision variables are $\mathtt{T}_i \in \{0, 1\}$ and $\mathtt{T}_i = 1$ is task t_i is part of the selected MCS, $\mathtt{T}_i = 0$ otherwise. Let R be the set of problem resources (including both clusters – with capacity 1 – and communication ports – with capacity $= bandwidth$), and let rq_{ik} be the quantity of a resource $r_k \in R$ requested by a task t_i; $rq_{ik} = 0$ if the task does not require the resource. The set of tasks in an MCS must exceed the usage of at least one resource:

$$\bigvee_{r_k \in R} \left(\sum_{t_i} rq_{ik} T_i > C_k \right) \tag{5.1}$$

where and C_k is the capacity of resource r_k. At least one of the inequalities between round brackets in expression 5.1 must be true for the constraint to be satisfied. Moreover, the selected set must be minimal, in the sense that by removing any task from the MCS, the resource over-usage no longer occurs[4]. More formally:

$$\forall r_k \in R : \sum_{t_i} rq_{ik} T_i - \min_{T_i = 1} \{rq_{ik}\} \leq C_k$$

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

Finally, a MCS must contain tasks with an actual chance of overlapping: therefore if a precedence relation exists between two tasks t_i and t_j they cannot both be part of the same MCS:

$$\forall t_i, t_j \mid t_i \prec t_j \vee t_j \prec t_i : \quad \mathtt{T}_i \cdot \mathtt{T}_j = 0$$

A list of mutually exclusive tasks is given to the MCS finder by the scheduling search engine at each step of the solution process. The MCS finding problem is solved again via tree search. Tasks are selected according to their resource requirements, giving precedence to the most demanding ones: this biases the selection towards smaller MCS, and thus choice points with fewer branches; this often reduces the search effort to solve the scheduling problem.

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

Algorithm 6: Overview of the search strategy

```
1: set best MCS so far (best) to ∅
2: for r_k ∈ R do
3:    find a conflict set S by solving a minimum flow problem
4:    if weight of S is higher than C_k then
5:       refine S to a minimal conflict set S'
6:       if S' is better than the best MCS so far then
7:          best = S'
8:       end if
9:    end if
10: end for
11: if best = ∅ then
12:    the problem is solved
13: else
14:    open a choice point as described in Section 5.6.2
15: end if
```

5.6.2.2 Finding MCS via Min-flow and Local Search

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 project graph, hence complete enumeration, or even a smarted approach such as the one described in Section 5.6.2.1 incurs the risk of combinatorial explosion.

We hence propose an alternative method to detect possible conflict, based on the solution of a minimum flow problem on a specific resource r_k, as described in [Gol04]; the method has the major advantage of having polynomial complexity. Note however the conflict set found is not guaranteed to be minimal, nor to be well suited to open a choice point. We coped with this issue by adding a conflict improvement step. An overview of the adopted search strategy is shown in Algorithm 6. In the next section each of the steps will be described in deeper detail; the adopted criterion to evaluate the quality of a conflict set will be given as well.

Conflict Set Detection The starting observation for the minimum flow based conflict detection is that, if the problem contains a *minimal* conflict set, it also contains a non necessarily minimal conflict set, i.e. a conflict set not necessarily satisfying the minimality condition in the definition of Section 5.4; let us refer to this as a CS. Therefore we can check the existence of an MCS on a given resource r_k by checking the existence of any CS.

Resource Graph: Let us consider the augmented project graph, further annotated with all precedence constraint which can be detected by time reasoning; if we assign to each activity the requirement rq_{ik} as a weight. We refer to such weighted graph as resource graph $\langle T, A_R \rangle$, where $(t_i, t_j) \in A_R$ iff $t_i \preceq t_j$ or $e_p(\eta_i) \leq s_p(\sigma_j)$. If a set of tasks is a CS, then we have $\sum_{t_i \in S} rq_{ik} > C_k$. Since the tasks in a CS must have the possibility to overlap, they always form a stable set (or independent set) on the resource graph.

Max Weight Stable Set on the Resource Graph: We can therefore check the existence of a MCS on a resource r_k by finding the maximum weight independent set on the resource graph and checking its total weight; this amounts to solve the following ILP model P':

$$\mathbf{P'}: \max \sum_{t_i \in A} rq_{ik} x_i \qquad \text{s.t.} \sum_{t_i \in \pi_j} x_i \leq 1 \quad \forall \pi_j \in \Pi \qquad (5.2) \qquad x_i \in \{0,1\}$$

$$\mathbf{P''}: \min \sum_{\pi_j \in \Pi} y_j \qquad \text{s.t.} \sum_{t_i \in \pi_j} y_j \geq rq_{ik} \quad \forall t_i \in T \qquad (5.3) \qquad y_j \in \{0,1\}$$

where x_i are the decision variables and $x_i = 1$ iff task t_i is in the selected set; Π is the set of all paths in the graph (in exponential number) and π_j is a path in Π. As for the constraints (5.2) consider that, due to the transitivity of temporal relations, a clique on the resource graph is always a path from source to sink. In any independent set no two nodes can be selected from the same clique, therefore, no more than one task can be selected from each path π_j in the set Π of all graph paths.

The corresponding dual problem is P'', where variable y_j is path π_j is selected; that is, finding the maximum weight stable set on a transitive graph amounts to find the minimum set of source-to-sink paths such that all nodes are covered by a number of paths at least equal to their requirement (constraints (5.3)). Note that, while the primal problem features an exponential number of constraints, its dual has an exponential number of variables. One can however see that the described dual is equivalent to route the least possible amount of flow from source to sink, such that a number of minimum flow constraints are satisfied; therefore, by introducing a real variable f_{ij} for each edge in A_R, we get:

$$\begin{aligned}
\min \quad & \textstyle\sum_{t_j \in T^+(t_0)} f_{0j} \\
\text{s.t.} \quad & \textstyle\sum_{t_j \in T^-(t_i)} f_{ji} \geq rq_{ik} \quad \forall t_i \in T && (5.4) \\
& \textstyle\sum_{t_j \in T^-(t_i)} f_{ji} = \sum_{t_j \in T^+(t_i)} f_{ij} \quad \forall t_i \in T \setminus \{t_0, t_{n-1}\} && (5.5) \\
& f_{ij} \geq 0
\end{aligned}$$

where $T^+(a_i)$ denotes the set of direct successors of t_i and $T^-(t_i)$ denotes the set of direct predecessors. One can note this is a flow minimization problem. Constraints (5.4) are the same as constraints (5.3), while the flow balance constraints (5.5) for all intermediate activities are implicit in the previous model. The problem can be solved starting for an initial feasible solution by iteratively reducing the flow with the any embodiment of the inverse Ford-Fulkerson's method, with complexity $O(|A_R| \cdot \mathcal{F})$ (where $\mathcal{F}$ is the value of the initial flow). Once the final flow is known, activities in the source-sink cut form the maximum weight independent set.

Finding MCS by solving a Min Flow Problem: In our approach we solve the minimum flow problem by means of the Edmond-Karp's algorithm. On this purpose each task t_i has to be split into two subnodes t'_i, t''_i; the connecting arc (t'_i, t''_i) is then given minimum flow requirement rq_{ik}; every arc $(t_i, t_j) \in A_R$ is converted into an arc (t''_i, t'_j) and assigned minimum flow requirement 0.

Observe that the structure of the resulting graph closely matches that of the time graph, which indeed can be used as a basis for the min-flow problem formulation. Figure 5.8A shows the time graph from Figure 5.4, annotated with the flow requirement corresponding to the tasks in the original graph mapped to cluster 0 (note those require each 1 unit of a cluster resource). Note the added task $t_{0,2}$, $t_{7,9}$ have 0 requirement, as they model data transfer, which are carried out by the DMA engine; similarly, all other tasks have 0 requirement, as they are mapped to cluster 1. The figure show the S/T cut corresponding to the maximum weight Conflict Set, consisting of tasks $\{t_3, t_4\}$ (highlighted in Figure 5.8B).

In general, we compute the initial solution to the min-flow problem by:

1. selecting each arc (t'_i, t''_i) in the new graph with minimum flow requirement $rq_{ik} > 0$
2. routing rq_{ik} units of flow along a backward path from t'_i to t'_0 (source)
3. routing rq_{ik} units of flow along a forward path from t''_i to t''_{n-1} (sink)

minor optimizations are performed in order to reduce the value of the initial flow. If at the end of the process the total weight of the independent set is higher than C_k, then a CS has been identified.

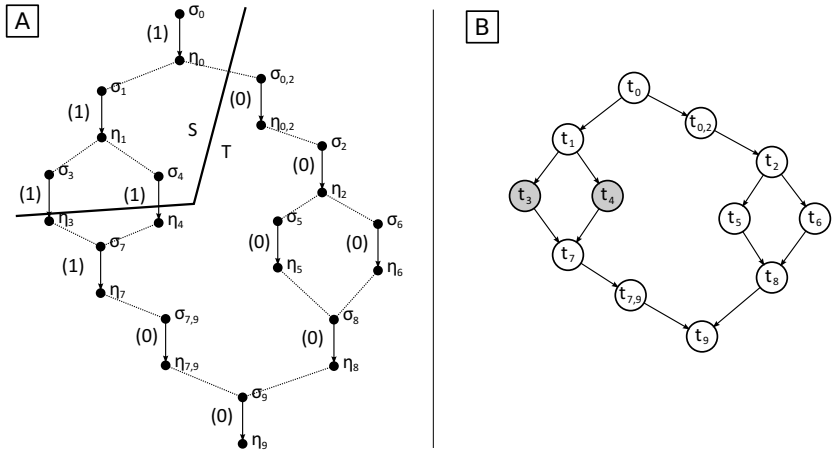


Figure 5.8: A) Minimum flow problem on the time graph; B) Corresponding Conflict Set

Reduction to MCS Once a conflict set has been identified, a number of issues are still pending; namely (1) the detected CS is not necessarily minimal and (2) the detected CS does not necessarily yield a good choice point. Branching on non-minimal CS can result in exploring unnecessary search paths, but luckily extracting a MCS from a given CS can be done very easily in polynomial time. The second issue is instead more complex, as it requires to devise a good MCS evaluation heuristic. We propose to tackle both problems by performing local search based intensification.

As evaluation criterion for a given conflict set S we use the lexicographic composition of (1) the number of precedence constraints which *can* be posted between pairs of tasks $t_i, t_j \in S$, and of (2) the size of the set itself (i.e. $|S|$);

for both criteria, lower values are better. Note a precedence constraint *cannot* be posted on t_i, t_j iff either of the followings holds:

1. $t_j \prec t_i$ in the time graph, where $\prec$ denotes a strict precedence relation. Note this can be checked in constant time by means of the time graph.
2. $s_o(\eta_i) > e_o(\sigma_j)$ in the time model, as briefly discussed in section 5.6.1, where η_i and σ_j are time points representing the end of t_i and the start of t_j. This check can be performed in constant time thanks to the specialized time windows introduced in Section 5.6.1.

As a consequence, local search naturally moves towards CS yielding choice points with a small number of branches: this goes in the same direction of the "minimum size domain" variable selection criterion (see Section 2.2.2). Once the number of branches in the resulting choice point cannot be further reduced, nodes are removed from the CS turning it into a minimal conflict set. Of course the total weight of is kept above C_k (S must remain a conflict set). In detail, given a conflict set S, we consider the following pool of local search moves:

1. $\texttt{add}(\texttt{S}, \texttt{t}_\texttt{i})$: a task $t_i \notin S$ is added to S, all tasks $t_j \in S$ such that $(t_i, t_j) \in A_R$ or $(t_j, t_i) \in A_R$ are removed from the set. The number of precedence constraints that can be posted is updated accordingly. The move has quadratic complexity.
2. $\texttt{del}(\texttt{S}, \texttt{t}_\texttt{i})$: a node $t_i \in S$ is removed from S; the number of precedence constraints that can be posted is updated accordingly. The move has linear complexity.

At each local search step all `del` moves for $t_i \in S$ are evaluated, together with all `add` moves for every immediate predecessor or successor of activities in S; the best move strictly improving the current set is then chosen. The process stops when a local optimum is reached. Note that in Figure 5.8B no LS move can be performed without if the overall requirement of cluster 0 has to be kept above 1; in fact, the set $\{t_3, t_4\}$ is already an MCS.

5.6.3 Detecting unsolvable Conflict Sets

At search time, it is possible for a Conflict Set to become unsolvable, that is: no resolver can be posted at all. This situation is in practice not very frequent and does not compromise convergence, but it can have a substantial impact on the solver performance if not promptly detected. On this purpose an additional step is added at the beginning of each search step (prior to MCS finding), where an attempt to identify unsolvable conflict sets is performed. Observe that a conflict set S is unsolvable iff for each pair $t_i, t_j \in S$ neither $\eta_i \xrightarrow{[0,\infty]} \sigma_j$ nor $\eta_j \xrightarrow{[0,\infty]} \sigma_i$ can be posted. In practice, if we build an *undirected* graph where an edge connecting t_i and t_j is present if such a situation holds, an unsolvable CS for a resource r_k always forms a clique with weight higher than C_k.

Figure 5.9 shows the graph for the TG in Figure 5.3B, in the hypothesis that a tight global deadline constraint is posted (note that 12 is the best deadline value for which feasibility can be guaranteed). Cliques in the graphs represent group of node which cannot prevent from overlapping; in case any clique exceed the capacity of some resource an unsolvable CS is detected; in the example, $\{t_3, t_4\}$ (overusing cluster 0) and $\{t_5, t_6\}$ (overusing cluster 1) are unsolvable.

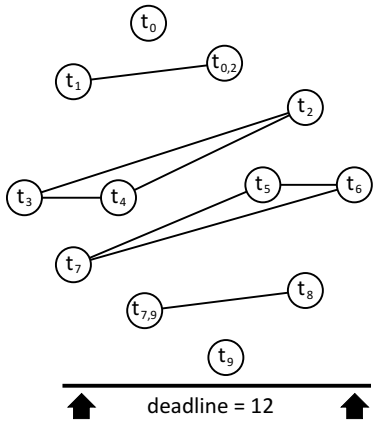


Figure 5.9: Graph for unsolvable CS identification

As neither the special graph we have just described nor its complement are transitive, the minimum flow based method cannot be used to detect an unsolvable CS. We therefore resorted to complete search, taking advantage of the very sparse structure quite often exhibited by the special graph. During search, nodes are selected according to their degree (number of adjacent edges), deprived of the currently selected set; such degree is dynamically updated as new nodes are selected. In order to limit the search effort we finally run the process with a fail limit, which was empirically set to $10\times$ the size of the transformed Task Graph used as input for the scheduling stage.

5.6.4 Feasibility and optimality

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

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

5.7 Experimental results

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

We tested the system on groups of randomly generated task graphs designed to mimic the structure of real-world applications[5]. In particular all groups contain so called *structured graphs* (see Section 2.1): they have a single starting task and a single terminating task, and feature branch nodes with variable fan out and a collector node for each branch; this reflects typical control/data flows arising with structured programming languages (such as C or C++). For all instances, the maximum task duration can be up to twice the minimum durations; data transfers within the same clusters are assumed to be instantaneous, while inter-cluster communications have finite duration (up to 1/4 of average task execution time) and use from 10% to 90% of the port bandwidth. We consider two different platforms: the first one has 16 single-instruction-stream accelerators connected with the interconnect by ports with the same bandwidth, the second contains 4 accelerators with 4 processors each. Inter-cluster communications employ ports with the same bandwidth as in the first platform.

Results concerning the approach with complete search based MCS identification are presented in Section 5.7.1; such approach is compared with a Worst Case Execution Time (WCET) scheduler, assuming the maximum duration for each task. Such method is much faster, but requires time triggered scheduling and idle time, hence achieving robustness at the cost of flexibility. Results for the Min-flow based conflict identification methods are instead in Section 5.7.2; the alternative MCS finding method is compared with the complete search one.

5.7.1 Results with Complete Search Based MCS Finding

The first group of experiments considers graphs with a fixed fan out (3 to 5 branches) and variable size (from 10 to 70 tasks). In each experiment we ran the optimality solver described in Section 5.6.4: this process requires to solve a number feasibility problems in order to identify the lowest feasible deadline and the highest infeasible deadline. All runs were performed on an Intel Core 2 Duo, with 2GB or RAM; the solver was given a time limit of 15 minutes.

Table 5.1 lists the results for the first group of experiments, related to platform type 1 and 2 (column "P"). Each row reports results for groups of 10 instances, showing the average deadline for the lowest feasible and the highest infeasible solution found (columns DL), the ratio between those two values (I/F) the number of added precedence relation in the best solution (N_{pr}). Detailed information is given about the solution time: the average and median time for the optimization process is reported in column total T_a/T_m; column TL shows the number of timed out instances (which are not considered in the average and median computation); N_{it} contains the average number of iterations performed. To give an evaluation of the feasibility version of the solver the solution time for the highest infeasible and lowest feasible deadlines is also given. Finally, the retained flexibility (FL) w.r.t. the WCET scheduler is shown: this is computed as

[5] both the generator and the instances are available at http://www.lia.deis.unibo.it/Staff/MicheleLombardi/

$(CT_{WCET} - \min(CT_{PCP}))/CT_{WCET}$, where CT_{WCET} is the completion time of the WCET schedule and $\min(CT_{PCP})$ is the minimum possible completion time of the PCP schedule. Note that time for the allocation step is negligible and is omitted in the results. Similarly, the running time for the WCET scheduler is omitted, but it is always much faster then the PCP approach.

Considering table 5.1, clearly, the solution time and the number of timed out instances grows as the number of tasks increases. The very high values for the deadline ratio show how tight bounds for the optimal deadline can be computed even for timed out instances. This aspect is crucial for designing a heuristic method based on the optimal version of the algorithm proposed that provides deadline bounds at a given time limit. Differences between the average and median solution time can be used as an indicator of the stability of the solution time; in particular very close average and median values are reported for feasible deadlines, while a much less stable behavior is observed for infeasible deadlines: detecting infeasibility seems to be often an easy task, but rare complexity peaks do arise; this could be expected, being the problem NP-hard. Mapping and scheduling the considered task graphs on the clustered platform appears to be an easier task. Not only the solution time is lower, but the behavior of the solver is much more stable; this is probably due to the smaller number of inter-cluster communications, which are scheduled as resource demanding tasks, and therefore increase the actual problem size and the number of conflict sets. The shorter deadlines are also due to the reduced number of inter-cluster communications.

To investigate the solver sensitivity to graph parallelism, in the second group of experiments we considered instances with a fixed number of tasks, but variable fan out (from 2-to-4 to 6-to-8). The results for this second group of instances, mapped respectively on platform 1 and 2, are reported in table 5.2. Again, each row summarizes results for 10 experiments. Both for platform 1 and 2, increasing the fan out makes the problem more difficult, since the higher parallelism translates into a higher number of conflict sets. For the same reason also more precedence relations have to be added to prevent the occurrence of resource over-usage. As for the previous group, mapping and scheduling on the clustered platforms is easier and produces better optimal deadlines.

The last group of instances contains an aggregation of small, independent structured task graphs with low fan out (2-3). This case is of high interest as the result of the unrolling of a simple multirate system would likely look like

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

Table 5.1: First group of instances (variable size)

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

Table 5.2: Second group of instances (variable fan-out)

this type of graphs. We generate composite graphs with 1 to 7 subgraphs, 10 to 70 tasks. The results for platform 1 and 2 are reported in table 5.3, again by groups of 10 instances. The complexity of the mapping and scheduling problem on platform 1 grows as the number of tasks increases. Note however how, despite the high parallelism calls for a large number of precedence relations, this kind of graphs seems to be easier to solve than those in the first group; this behavior requires a deeper analysis, which is subject of future work. Finally, note how mapping composite graphs on platform 2 is surprisingly easy: the nature of those graph allows the allocator to almost completely allocate different subgraph on different clusters, effectively reducing inter-cluster communications.

5.7.2 Results with Min-flow Based MCS Finding

This section reports results for the solver where the MCS finding relies on the solution of a min-flow problem and on local search (as described in Section 5.6.2.2). The method is compared with the results obtained by identifying MCS via complete search (the results in the previous section). In the following, we refer to the first approach as MF solver (Minimum Flow), and to the latter as CS solver (Complete Search).

The following testing process has been devised: for each available instance a very loose deadline requirement is first computed (namely, the sum of all worst case durations); next binary search is performed as described in Section 5.6.4; the process converges to the best possible deadline and provides information about the performance of the solvers, as well as an indication of the tightness to the best deadline constraint which can be reached by both solvers. A time

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

Table 5.3: Third group of instances (variable #subgraphs)

	N	FL	TT	T	F	N_{MCS}	T_{MCS}	T_O	MEM	FEAS T	FEAS T_{MCS}	INF T	INF T_{MCS}
A	41-49	0.59	—	7.03	22	164	2.82	0	11M	0.49	23.20	0.13	1.10
	56-66	0.56	0.96	90.12	1832	1140	67.34	1	28M	1.37	30.56	8.84	116.56
	75-82	0.55	—	49.24	30	287	18.07	0	55M	2.93	39.20	0.71	2.90
	93-103	0.56	0.97	130.34	81	533	55.79	3	110M	8.42	66.14	1.66	6.57
	111-116	0.56	0.89	251.08	71	720	129.22	4	172M	18.84	89.00	2.14	5.67
	121-128	0.59	0.92	365.05	96	666	169.58	6	222M	22.11	75.75	5.82	11.25
B	41-49	0.60	—	0.39	6	155	0.37	0	0.22M	0.05	0.05	0.00	0.01
	56-66	0.56	0.91	1.07	6	212	1.04	1	0.28M	0.15	0.15	0.00	0.01
	75-82	0.56	—	3.32	18	309	3.25	0	0.32M	0.41	0.39	0.09	0.09
	93-103	0.57	0.98	11.29	51	537	11.18	2	0.42M	1.29	1.27	0.23	0.23
	111-116	0.55	0.92	27.52	1175	1549	26.86	4	0.49M	3.28	3.22	0.09	0.08
	121-128	0.59	0.89	36.64	82	713	36.36	6	0.50M	3.19	3.19	1.80	1.78

Table 5.4: Results on the first group of instances (growing number of nodes) for (A) the CS solver and (B) the MF solver

limit of 900 seconds was set on the whole test process. Note the reported results only related to the first platform (16 single thread accelerators) and the second group of generated instances; the group representing applications with multiple independent task graphs is omitted.

Table 5.4 summarizes results on the first group of instance, respectively for the CS (A) and MF solver (B). Here the branching factor was fixed to the range 3-5. Note the results for the CS solver are indeed the same as Table 5.1, just group according to a different criterion, namely the number of tasks in the transformed graph resulting after the allocation step. Each row refers to a group of 10 instances, and shows the minimum and maximum number of nodes (**N**), the average solution time (**T**) and number of fails (**F**) for the entire test process, the number of MCS used to branch ($\mathbf{N_{MCS}}$) and the time spent to detect them ($\mathbf{T_{MCS}}$); the number of timed out instances and the total memory usage are respectively in columns **TO** and **MEM**. The solution time and the time to detect MCS is reported for single feasible and unfeasible runs as well (**FEAS**, **INF**).

Finally, columns **FL** and **TT** deserve special attention; whenever a feasible solution is found a flexibility indicator can be given by the ratio between the best case completion time and the worst case completion time for the produced graph; the average of such indicator is reported in **FL**. When a timeout occurs, the ratio between the current lower bound and the current upper bound on the best achievable deadline is computed and used as a tightness indicator; the average of such indicator is reported in **TT**.

	BF	FL	TT	T	F	N_{MCS}	T_{MCS}	TO	MEM	FEAS T	FEAS T_{MCS}	INF T	INF T_{MCS}
A	2-4	0.58	1.00	51.21	13	305	21.09	1	107M	3.05	38.22	0.62	0.00
	3-5	0.56	0.99	67.51	14	405	31.21	1	125M	4.97	55.44	0.66	0.11
	4-6	0.55	1.00	84.40	48	565	42.63	1	134M	6.72	75.00	0.83	2.78
	5-7	0.54	0.98	71.15	15	476	33.13	2	133M	5.36	66.50	0.62	0.00
	6-8	0.52	0.99	102.02	20	667	54.95	4	160M	8.40	88.00	0.67	0.33
B	2-4	0.58	—	3.74	48	385	3.67	0	0.33M	0.34	0.33	0.15	0.15
	3-5	0.56	—	4.17	7	423	4.09	0	0.36M	0.54	0.53	0.01	0.01
	4-6	0.56	0.80	41.00	5418	6230	39.65	1	0.39M	5.78	5.58	0.03	0.03
	5-7	0.54	0.49	4.88	6	455	4.78	2	0.37M	0.67	0.63	0.00	0.01
	6-8	0.51	0.93	6.16	6	576	6.06	4	0.42M	0.87	0.86	0.00	0.00

Table 5.5: Results on the second group of instances (growing branching factor) for (A) the CS solver and (B) the MF solver

As one can see, the MF solver reports improvements of around one order of magnitude both in the total solution time and in the time required to detect MCS; in particular, the latter has to be mainly ascribed to the flow based conflict detection method, while the former also benefits from the much more efficient time model used in the MF solver; the compactness of the time model also has a leading role in the drastic improvement in memory usage. The flexibility and the tightness achieved by the two solvers are comparable; note in particular the very high values of the tightness indicator reached by both the approaches. Note however the CS solver reports a smaller number of timed out instances: this is enable by the higher quality MCS found by the Constraint Programming based method.

Table 5.5 shows the same results for the second group of instances; here the number of tasks in the graph is always between 74 and 94, while the branching factor spans the interval reported for each row in the column **BF** . As one can observe, the trend of all results is around the same as the previous case, with the addition of the flexibility indicator getting higher as the branching factor increases. Note however the MF solver sometimes achieves considerably worse values for the tightness indicator value; this indicates the MF solver tends to stop earlier (i.e. for more loose bounds) in case of timeouts. Similarly to Table 5.4, the MF solver reports an higher number of timeouts compared to the CS one.

A last set of experiments was finally performed to test the impact of the local search (LS) and the unsolvable conflict set (UCS) detector; in fact, as those feature are not necessary for the method to converge, it is reasonable to question their actual effect on the solver efficiency. Table 5.6 reports results for those tests, performed on the first group of instances previously discussed. Both for the case when the UCS finder and the local search are turned off, the table shows the solution time (**T**), the number of processed MCS ($\mathbf{N_{MCS}}$) and the number of detected UCS ($\mathbf{N_{UCS}}$); the number of timed out instances (**TO**) is indicated as well. Columns $\mathbf{N_{MF}}$, $\mathbf{N_{LS}}$, $\mathbf{N_{UF}}$ respectively report the number of times the minimum flow, local search and UCS finder algorithm are run. The results show that the actual advantage of incorporating a UCS finder is shadowed by the efficacy of the local search CS improver, to the point that sometimes a better run time is achieved without the feature. On the other hand, no question arise about the effective utility of the local search method during MCS detection. Quite surprisingly however, turning LS off helps reducing the number of timeouts: this has to be further investigated.

	NO UCS FINDER						NO LOCAL SEARCH					
N	T	N_{MCS}	N_{UCS}	TO	N_{MF}	N_{LS}	T	N_{MCS}	N_{UCS}	TO	N_{MF}	N_{UF}
41-49	0.37	155	0	0	914	562	0.43	198	36	0	973	242
56-66	1.04	212	0	1	1417	1042	1.07	243	2	1	1383	253
75-82	3.52	325	0	0	3355	2933	3.96	393	41	0	3533	442
93-103	11.96	538	0	2	7660	7168	14.02	727	159	2	8685	894
111-116	18.51	631	0	5	8566	8133	48.65	1318	279	4	23744	1606
121-128	40.98	713	0	6	15499	14983	133.13	2049	1088	5	46073	3145

Table 5.6: Performance of the MF solver when some features are turned off

5.8 Conclusions and future work

We have devised a constructive algorithm for computing robust resource allocations and schedules, when tasks have uncertain, bounded, durations. The method couples a heuristic allocation step with a complete scheduling stage.

The approach has been devised to solve an important class of problems arising in the embedded system design process; namely, we address predictive mapping and scheduling for real-time applications with variable-duration tasks. Scheduling anomalies are eliminated by inserting sequencing constraints between tasks in a controlled way, i.e. only when needed to prevent resource conflicts which may appear for "unfortunate" combinations of execution times. We developed a solver that solves both feasibility and optimality problems.

Even though run-time is worst-case exponential (as expected for complete methods on NP-Hard problems), solution times in practice are very promising: task graphs with several tens of tasks can be handled in a matter of minutes. This is competitive with state-of-the-art verification methods used to test the presence of anomalies [BHM08].

Future work will focus on replacing the heuristic allocation step with (yet efficient) but complete approach; this will enable to repeat the allocation and scheduling process in a LBD like scheme. Introducing filtering for resource constraints is another necessary improvement; on this regard, the main difficulty is given by the different semantic for the task duration, requiring consistency to be enforced for each value, rather than for at least one value. Introducing a probability distribution on task durations is also subject of current research: that would enable more complex objectives, such as minimization of the expected execution time, by exploiting probability theory results.

Chapter 6

Concluding remarks

We have presented a number of hybrid algorithmic approaches for Allocation and Scheduling problems, making heavy use of decomposition techniques to provide integration support. In particular, we investigated the use of LBD for scheduling problems involving general precedence constraints in a deterministic environment. We highlighted the possibility to effectively use NP-hard cut generation schemes if the subproblem is easy enough, and the chance to apply recursive decomposition to boost the solver efficiency on subproblem parts. We devised efficient probabilistic reasoning methods for Conditional Task Graphs, and used them to introduce conditional constraints, and deterministic reduction techniques for expected value functionals (namely, assignment cost and makespan). We tackled A&S problems with tasks having uncertain, bounded durations and we showed how efficient conflict detection can be performed by solving a minimum flow problem.

Despite the developed techniques have broad applicability, they mainly target the design flow of embedded systems; in particular, we considered mapping and scheduling problems on both real (namely, the Cell BE processor) and realistic synthetic architectures (via carefully generated instances). We showed how in many cases practical size problems are within the reach of exact optimization methods; moreover, thanks to the very presence of hard real time constraints, uncertainty elements can be taken into account off-line without drastically increasing the problem complexity. Finally, we demonstrated how hybrid approaches can be used to achieve the level of flexibility and accuracy required for an optimization method to be included in a CAD tool.

Future works are detailed in Chapters 3, 4 and 5. Additionally, they include the actual embedding of the proposed techniques within a design software prototype, as well as their further development. On this purpose, heuristic approaches and specialized strategies for exact solvers (providing good solution in the early search stages) have to be investigated as well.

Part of the developed work has been published on international conferences [RLMB08, LMB09, LM09, LM07, LM06, BLMR08, BLM+08] and journals [LM10, BLMR10].

Bibliography

[ABB+06] Alexandru Andrei, Luca Benini, Davide Bertozzi, Alessio Guerri, Michela Milano, Gioia Pari, and Martino Ruggiero. A Cooperative, Accurate Solving Framework for Optimal Allocation, Scheduling and Frequency Selection on Energy-Efficient MPSoCs. In *Proc. of SOC*, 2006.

[AMD] AMD. AMD Web Site - Multicore Architectures. available at: http://multicore.amd.com/us-en/AMD-Multi-Core.aspx.

[AS02] S. Ahmed and A. Shapiro. The sample average approximation method for stochastic programs with integer recourse. available at: http://citeseer.ist.psu.edu/ahmed02sample.html, 2002.

[ASE+04] A. Andrei, M. Schmitz, P. Eles, Z. Peng, and B. Al-Hashimi. Overhead-Conscious Voltage Selection for Dynamic and Leakage Power Reduction of Time-Constraint Systems. In *Proc. of DATE*, pages 518–523, Paris, France, February 2004.

[Axe97] J. Axelsson. Architecture synthesis and partitioning of real-time systems: A comparison of three heuristic search strategies. In *Proc. of CODES*, pages 161–, 1997.

[Bac00] Fahiem Bacchus. Extending Forward Checking. In *Proc. of CP*, pages 35–51, 2000.

[Bak74] K. R. Baker. *Introduction to sequencing and scheduling.* John Wiley & Sons, 1974.

[BAM07] David A. Bader, Virat Agarwal, and Kamesh Madduri. On the Design and Analysis of Irregular Algorithms on the Cell Processor: A Case Study of List Ranking. In *Proc. of IPDPS*, pages 1–10. IEEE, 2007.

[Bar06] S. K. Baruah. The non-preemptive scheduling of periodic tasks upon multiprocessors. *Real-Time Systems*, 32(1):9–20, 2006.

[BB72] U. Bertelé and F. Brioschi. *Nonserial dynamic programming.* Academic Press, 1972.

[BB82] D. D. Bedworth and J. E. Bailey. *Integrated Production Control Systems - Management Analysis Design.* Wiley, New York, 1982.

[BBDM02] L. Benini, A. Bogliolo, and G. De Micheli. A survey of design techniques for system-level dynamic power management. *Readings in Hardware/Software Co-Design, G. De Micheli, R. Ernst, and W. Wolf, Eds. The Morgan Kaufmann Systems On Silicon Series. Kluwer Academic Publishers, Norwell, MA*, pages 231–248, 2002.

[BBGM05a] L. Benini, D. Bertozzi, A. Guerri, and M. Milano. Allocation and scheduling for mpsocs via decomposition and no-good generation. 19:1517, 2005.

[BBGM05b] Luca Benini, Davide Bertozzi, Alessio Guerri, and Michela Milano. Allocation and Scheduling for MPSoCs via Decomposition and No-Good Generation. In Peter van Beek, editor, *Proc. of CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2005.

[BBGM06] Luca Benini, Davide Bertozzi, Alessio Guerri, and Michela Milano. Allocation, Scheduling and Voltage Scaling on Energy Aware MPSoCs. In J. Christopher Beck and Barbara M. Smith, editors, *Proc. of CPAIOR*, volume 3990 of *Lecture Notes in Computer Science*, pages 44–58. Springer, 2006.

[BC94] C. Bessiere and M. O. Cordier. Arc-consistency and arc-consistency again. *Artificial intelligence*, 65(1):179–190, 1994.

[BC07] Roman Barták and Ondrej Cepek. Temporal Networks with Alternatives: Complexity and Model. In David Wilson and Geoff Sutcliffe, editors, *Proc. of FLAIRS*, Proceedings of the Twentieth International Florida Artificial Intelligence Research Society Conference, May 7-9, 2007, Key West, Florida, USA, pages 641–646. AAAI Press, 2007.

[BC09a] L. Bianco and M. Caramia. A new lower bound for the resource-constrained project scheduling problem with generalized precedence relations. *Computers and Operations Research*, 2009.

[BC09b] Lucio Bianco and Massimiliano Caramia. An Exact Algorithm to Minimize the Makespan in Project Scheduling with Scarce Resources and Feeding Precedence Relations. In *Proc. of CTW*, pages 210–214, 2009.

[BCP08] Nicolas Beldiceanu, Mats Carlsson, and Emmanuel Poder. New Filtering for the cumulative Constraint in the Context of Non-Overlapping Rectangles. In Laurent Perron and Michael A. Trick, editors, *Proc. of CPAIOR*, volume 5015 of *Lecture Notes in Computer Science*, pages 21–35. Springer, 2008.

[BCR05] N. Beldiceanu, M. Carlsson, and J. X. Rampon. Global constraint catalog. Technical Report SICS Technical Report T2005:08, 2005.

[BCS07] R. Bartak, O. Cepek, and P. Surynek. Modelling Alternatives in Temporal Networks. In *Proc. of IEEE SCIS*, pages 129–136, April 2007.

[BD02] J. C. Beck and A. J. Davenport. A survey of techniques for scheduling with uncertainty. Available from http://www.eil.utoronto.ca/profiles/chris/gz/uncertainty-survey.ps, 2002.

[BDM+99] Peter Brucker, Andreas Drexl, Rolf Möhring, Klaus Neumann, and Erwin Pesch. Resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operational Research*, 112(1):3–41, 1999.

[BEA+08] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, et al. TILE64 processor: A 64-core SoC with mesh interconnect. pages 88–598, 2008.

[Bee06] Peter Van Beek. *Handbook of constraint programming*, chapter Backtracking Search Algorithms, pages 85–134. Elsevier Science Ltd, 2006.

[Ben62] J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, (4):238–252, 1962.

[Ben96] A. Bender. MILP based task mapping for heterogeneous multiprocessor systems. In *Proc. of EURO-DAC/EURO-VHDL*, pages 190–197, Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[BF00] J. C. Beck and M. S. Fox. Constraint-directed techniques for scheduling alternative activities. *Artificial Intelligence*, 121(1-2):211–250, 2000.

[BHLS04] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting Systematic Search by Weighting Constraints. In *Proc. of ECAI*, pages 146–150, 2004.

[BHM08] Aske Brekling, Michael R. Hansen, and Jan Madsen. Models and formal verification of multiprocessor system-on-chips. *Journal of Logic and Algebraic Programming*, 77(1-2):1–19, 2008. The 16th Nordic Workshop on the Prgramming Theory (NWPT 2006).

[BJK93] P. Brucker, B. Jusrisch, and A. Kramer. A new lower bound for the job-shop scheduling problem. *European Journal of Operational Research*, (64):156–167, 1993.

[BK00] P. Brucker and S. Knust. A linear programming and constraint propagation-based lower bound for the RCPSP. *European Journal of Operational Research*, 127(2):355–362, 2000.

[BKST98] P. Brucker, S. Knust, A. Schoo, and O. Thiele. A branch and bound algorithm for the resource-constrained project scheduling problem. *European Journal of Operational Research*, 107(2):272–288, 1998.

[BLM+08] Luca Benini, Michele Lombardi, Marco Mantovani, Michela Milano, and Martino Ruggiero. Multi-stage Benders Decomposition for Optimizing Multicore Architectures. In Laurent Perron and Michael A. Trick, editors, *Proc. of CPAIOR*, volume 5015 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2008.

[BLMR08] Luca Benini, Michele Lombardi, Michela Milano, and Martino Ruggiero. A Constraint Programming Approach for Allocation and Scheduling on the CELL Broadband Engine. In Peter J. Stuckey, editor, *Proc. of CP*, volume 5202 of *Lecture Notes in Computer Science*, pages 21–35. Springer, 2008.

[BLMR10] Luca Benini, Michele Lombardi, Michela Milano, and Martino Ruggiero. Optimal Allocation and Scheduling for the Cell BE Platform. *to appear on: Annals of Operations Research*, 2010.

[BLPN01] P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-based scheduling*. Kluwer Academic Publishers, 2001.

[BMR88] M. Bartusch, R. H. Möhring, and F. J. Radermacher. Scheduling project networks with resource constraints and time windows. *Annals of Operations Research*, 16(1):199–240, 1988.

[Bor99] Shekhar Borkar. Design Challenges of Technology Scaling. *IEEE Micro*, 19(4):23–29, 1999.

[Bor07] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proc. of DAC*, pages 746–749, New York, NY, USA, 2007. ACM.

[BP00] Philippe Baptiste and Claude Le Pape. Constraint Propagation and Decomposition Techniques for Highly Disjunctive and Highly Cumulative Project Scheduling Problems. *Constraints*, 5(1/2):119–139, 2000.

[BP07] Nicolas Beldiceanu and Emmanuel Poder. A Continuous Multi-resources umulative Constraint with Positive-Negative Resource Consumption-Production. In *Proc. of CPAIOR*, pages 214–228, 2007.

[BR96] Christian Bessière and Jean-Charles Régin. MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems. In *Proc. of CP*, pages 61–75, 1996.

[Bré79] D. Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):256, 1979.

[BVLB07] Julien Bidot, Thierry Vidal, Philippe Laborie, and J. Christopher Beck. A General Framework for Scheduling in a Stochastic Environment. In *Proc. of IJCAI*, pages 56–61, 2007.

[CAVT87] N. Christofides, R. Alvarez-Valdes, and J. M. Tamarit. Project scheduling with resource constraints: A branch and bound approach. *European Journal of Operational Research*, 29(3):262–273, 1987.

[CF99] W. Y. Chiang and M. S. Fox. Protection against uncertainty in a deterministic schedule. In *Proc. of International Conference on Expert Systems and the Leading Edge in Production and Operations Management*, pages 184–197, 1999.

[CHD+04] H. Cambazard, P. E. Hladik, A. M. Déplanche, N. Jussien, and Y. Trinquet. Decomposition and learning for a hard real time task allocation problem. *Lecture Notes in Computer Science*, 3258:153–167, 2004.

[CJ05] Hadrien Cambazard and Narendra Jussien. Integrating Benders Decomposition Within Constraint Programming. In Peter van Beek, editor, *Proc. of CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 752–756. Springer, 2005.

[Cor02] Intel Corporation. Intel IXP2800 Network Processor Product Brief. Available at http://download.intel.com/design/network/ProdBrf/27905403.pdf, 2002.

[COS00] Amedeo Cesta, Angelo Oddi, and Stephen F. Smith. Iterative Flattening: A Scalable Method for Solving Multi-Capacity Scheduling Problems. In *Proc. of AAAI/IAAI*, pages 742–747, 2000.

[CS71] D. R. Cox and W. L. Smith. *Queues*. Chapman & Hall/CRC, 1971.

[CV02] Karam S. Chatha and Ranga Vemuri. Hardware-Software partitioning and pipelined scheduling of transformative applications. *IEEE Trans. Very Large Scale Integr. Syst.*, 10(3):193–208, 2002.

[DB05] R. Dechter and Roman Bartak. Constraint Processing, Morgan Kaufmann (2003). *Artif. Intell.*, 169(2):142–145, 2005.

[DBS94] M. Drummond, J. Bresina, and K. Swanson. Just-in-case scheduling. In *Proc. of ICAI*, pages 1098–1104, 1994.

[DDRH00] E. Demeulemeester, B. De Reyck, and W. Herroelen. The discrete time/resource trade-off problem in project networks: a branch-and-bound approach. *IIE transactions*, 32(11):1059–1069, 2000.

[Dec99] Rina Dechter. Bucket Elimination: A Unifying Framework for Reasoning. *Artif. Intell.*, 113(1-2):41–85, 1999.

[DH71] Edward W. Davis and George E. Heidorn. An Algorithm for Optimal Project Scheduling under Multiple Resource Constraints. *Management Science*, 17(12):–803, 1971.

[DH92] E. Demeulemeester and W. Herroelen. A branch-and-bound procedure for the multiple resource-constrained project scheduling problem. *Management science*, 38(12):1803–1818, 1992.

[DLRV03] G. Desaulniers, A. Langevin, D. Riopel, and B. Villeneuve. Dispatching and conflict-free routing of automated guided vehicles: an exact approach. *International Journal of Flexible Manufacturing Systems*, 15(4):309–331, 2003.

[DMP91] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal Constraint Networks. *Artif. Intell.*, 49(1-3):61–95, 1991.

[dSNP88] J. L. de Siqueira N. and Jean-Francois Puget. Explanation-Based Generalisation of Failures. In *Proc. of ECAI*, pages 339–344, 1988.

[EKP+98] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop. Scheduling of conditional process graphs for the synthesis of embedded systems. pages 132–138, 1998.

[Elm77] S. E. Elmaghraby. *Activity networks: Project planning and control by network models.* John Wiley & Sons, 1977.

[ELT91] J. Erschler, P. Lopez, and C. Thuriot. Raisonnement temporel sous contraintes de ressource et problèmes d'ordonnancement. *Revue d'intelligence artificielle*, 5(3):7–32, 1991.

[ERL90] H. El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9:138–153, 1990.

[FAD+05] B. Flachs, S. Asano, S. H. Dhong, P. Hotstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H. Oh, S. M. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, and N. Yano. A streaming processing unit for a CELL processor. *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pages 134–135, 10-10 Feb. 2005.

[FB06] N. Fisher and S. Baruah. The partitioned multiprocessor scheduling of non-preemptive sporadic task systems. 2006.

[FFY01] P. Faraboschi, J. A. Fisher, and C. Young. Instruction scheduling for instruction level parallel processors . *Proceedings of the IEEE*, 89(11):1638–1659, Nov 2001.

[FLM99] F. Focacci, A. Lodi, and M. Milano. Cost-based domain filtering. *Lecture Notes in Computer Science*, pages 189–203, 1999.

[FLM02a] F. Focacci, A. Lodi, and M. Milano. A hybrid exact algorithm for the TSPTW. *INFORMS Journal on Computing*, 14(4):403–417, 2002.

[FLM02b] F. Focacci, A. Lodi, and M. Milano. Optimization-oriented global constraints. *Constraints*, 7(3):351–365, 2002.

[FR97] G. Fohler and K. Ramamritham. Static Scheduling of Pipelined Periodic Tasks in Distributed Real-Time Systems. In *Proc. of EUROMICRO-RTS97*, pages 128–135, Toledo, Spain, June 1997.

[FS09] T. Feydy and P. Stuckey. Lazy clause generation reengineered. 5732, 2009.

[ftAoSISO] Organization for the Advancement of Structured Information Standards (OASIS). Web Services Business Process Execution. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.

[Gas78] J. Gaschnig. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. In *Proc. of the Second Biennial Conference, Canadian Society for the Computational Studies of Intelligence*, pages 268–277, 1978.

[GB65] Solomon W. Golomb and Leonard D. Baumert. Backtrack Programming. *J. ACM*, 12(4):516–524, 1965.

[Ger94] C. Gervet. Conjunto: constraint logic programming with finite set domains. pages 339–358, 1994.

[Gho96] S. Ghosh. *Guaranteeing fault tolerance through scheduling in real-time systems*. PhD thesis, University of Pittsburgh, 1996.

[GJ74] J. C. Gittins and D. M. Jones. A dynamic allocation index for the sequential design of experiments. *Progress in statistics*, 241:266, 1974.

[GJ+79] M. R. Garey, D. S. Johnson, et al. *Computers and Intractability: A Guide to the Theory of NP-completeness*. wh freeman San Francisco, 1979.

[GLLK79] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. R. Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5(2):287–326, 1979.

[GLM07] Alessio Guerri, Michele Lombardi, and Michela Milano. Challenging Scheduling in the field of System Design. available at http://www.lompa.it/michelelombardi/node/19, 2007.

[GLN05] Daniel Godard, Philippe Laborie, and Wim Nuijten. Randomized Large Neighborhood Search for Cumulative Scheduling. In *Proc. of ICAPS*, pages 81–89, 2005.

[GMM95] S. Ghosh, R. Melhem, and D. Mosse. Enhancing real-time schedules to tolerate transient faults. *Real-Time Systems Symposium, IEEE International*, 0:120, 1995.

[Gol04] Martin Golumbic. *Algorithmic Graph Theory And Perfect Graphs*. Elsevier, Second edition edition, 2004.

[Gom04] Carla Gomes. *Constraint and Integer Programming: Toward a Unified Metodology*, chapter Randomized Backtrack Search, pages 233–292. Kluwer, 2004.

[Gra07] John R. Graham. Integrating parallel programming techniques into traditional computer science curricula. *SIGCSE Bull.*, 39(4):75–78, 2007.

[GS00] I. P. Gent and B. M. Smith. Symmetry breaking in constraint programming. pages 599–603, 2000.

[HB09] S. Hartmann and D. Briskorn. A survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*, 2009.

[HD98] S. Hartmann and A. Drexl. Project scheduling with multiple modes: A comparison of exact algorithms. *Networks*, 32(4):283–297, 1998.

[HDRD98] W. Herroelen, B. De Reyck, and E. Demeulemeester. Resource-constrained project scheduling: a survey of recent developments. *Computers and Operations Research*, 25(4):279–302, 1998.

[HE80] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3):263–313, 1980.

[HG95] William D. Harvey and Matthew L. Ginsberg. Limited Discrepancy Search. In *Proc. of IJCAI*, pages 607–615, 1995.

[HG01] I. Harjunkoski and I. E. Grossmann. A decomposition approach for the scheduling of a steel plant production. *Computers and Chemical Engineering*, 25(11-12):1647–1660, 2001.

[HG02] I. Harjunkoski and I. E. Grossmann. Decomposition techniques for multistage scheduling problems using mixed-integer and constraint programming methods. *Computers and Chemical Engineering*, 26(11):1533–1552, 2002.

[HK06] Willem Jan van Hoeve and Irit Katriel. *Handbook of constraint programming*, chapter Global constraints, pages 169–208. Elsevier, 2006.

[HL05a] W. Herroelen and R. Leus. Project scheduling under uncertainty: Survey and research potentials. *European journal of operational research*, 165(2):289–306, 2005.

[HL05b] Willy Herroelen and Roel Leus. Project scheduling under uncertainty: Survey and research potentials. *European Journal of Operational Research*, 165(2):289–306, 2005.

[HM04] Willem Jan van Hoeve and Michela Milano. Decomposition Based Search - A theoretical and experimental evaluation. *CoRR*, cs.AI/0407040, 2004.

[HMH01] R. Ho, K. W. Mai, and M. A. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, 2001.

[HO03] J. N. Hooker and G. Ottosson. Logic-based Benders decomposition. *Mathematical Programming*, 96(1):33–60, 2003.

[Hoo05a] J. N. Hooker. Planning and scheduling to minimize tardiness. *Lecture notes in computer science*, 3709:314, 2005.

[Hoo05b] John N. Hooker. A Hybrid Method for Planning and Scheduling. *Constraints*, 10(4):385–401, 2005.

[Hoo07] J. N. Hooker. Planning and scheduling by logic-based benders decomposition. *OPERATIONS RESEARCH-BALTIMORE THEN LINTHICUM-*, 55(3):588, 2007.

[Hor07] Mark Horowitz. Scaling, Power and the Future of CMOS. In *Proc. of VLSID*, page 23, Washington, DC, USA, 2007. IEEE Computer Society.

[HSN+02] Mark A. Horowitz, Vladimir Stojanovic, Borivoje Nikolic, Dejan Markovic, and Robert W. Brodersen. Methods for true power minimization. *Computer-Aided Design, International Conference on*, 0:35–42, 2002.

[HY94] J. N. Hooker and H. Yan. Verifying logic circuits by Benders decomposition. *Saraswat and Van Hentenryck*, 105, 1994.

[ILO94] S. A. ILOG. Implementation of resource constraints in ilog schedule: A library for the development of constraint-based scheduling systems. *Intelligent Systems Engineering*, 3(2):55–66, 1994.

[IR83a] G. Igelmund and F. J. Radermacher. Algorithmic approaches to preselective strategies for stochastic scheduling problems. *Networks*, 13(1):29–48, 1983.

[IR83b] G. Igelmund and F. J. Radermacher. Preselective strategies for the optimization of stochastic project networks under resource constraints. *Networks*, 13(1):1–28, 1983.

[Jac55] James R. Jackson. Scheduling a Production Line to Minimize Maximum Tardiness. Technical Report 43, 1955.

[JFL+07] A. A. Jerraya, O. Franza, M. Levy, M. Nakaya, P. Paulin, U. Ramacher, D. Talla, and W. Wolf. Roundtable: Envisioning the Future for Multiprocessor SoC. *Design & Test of Computers, IEEE*, 24(2):174–183, March-April 2007.

[JG01] Vipul Jain and Ignacio E. Grossmann. Algorithms for Hybrid MILP/CP Models for a Class of Optimization Problems. *INFORMS Journal on Computing*, 13(4):258–276, 2001.

[JG05] R. Jejurikar and R. Gupta. Dynamic Slack Reclamation with Procrastination Scheduling in Real-Time Embedded Systems. In *Proc. of DAC*, pages 111–116, San Diego, CA, USA, June 2005.

[Jun04] Ulrich Junker. QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained Problems. In Deborah L. McGuinness and George Ferguson, editors, *Proc. of AAAI*, Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA, pages 167–172. AAAI Press / The MIT Press, 2004.

[KA01] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2001.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974.

[Kar] George Karypis. METIS Web Site. http://glaros.dtc.umn.edu/gkhome/metis/.

[KB05] George Katsirelos and Fahiem Bacchus. Generalized NoGoods in CSPs. In *Proc. of AAAI*, pages 390–396, 2005.

[Kel63] J. E. Jr. Kelley. The Critical Path Method: Resources Planning and Scheduling. *Industrial Scheduling*, pages 347–365, 1963.

[KH06] R. Kolisch and S. Hartmann. Experimental investigation of heuristics for resource-constrained project scheduling: An update. *European Journal of Operational Research*, 174(1):23–37, 2006.

[KJF07] J. Kuster, D. Jannach, and G. Friedrich. Handling alternative activities in resource-constrained project scheduling problems. pages 1960–1965, 2007.

[KK99] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359, 1999.

[Kol96] R. Kolisch. Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation. *European Journal of Operational Research*, 90(2):320–333, 1996.

[KP01] R. Kolisch and R. Padman. An integrated survey of deterministic project scheduling. *Omega*, 29(3):249–272, 2001.

[KPP06] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell Multiprocessor Communication Network: Built for Speed. *IEEE Micro*, 26(3):10–23, 2006.

[Kuc03] Krzysztof Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Trans. Design Autom. Electr. Syst.*, 8(3):355–383, 2003.

[KW03] K. Kuchcinski and C. Wolinski. Global approach to assignment and scheduling of complex behaviors based on HCDG and constraint programming. *Journal of Systems Architecture*, 49(12-15):489–503, 2003.

[KWGS] S. Kodase, S. Wang, Z. Gu, and K. G. Shin. Improving scalability of task allocation and scheduling in large distributed real-time systems using shared buffers. *Ann Arbor*, 1001:48109–2122.

[Lab03] Philippe Laborie. Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. *Artif. Intell.*, 143(2):151–188, 2003.

[Lab05] Philippe Laborie. Complete MCS-Based Search: Application to Resource Constrained Project Scheduling. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *Proc. of IJCAI*, IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005, pages 181–186. Professional Book Center, 2005.

[LEE92] P. Lopez, J. Erschler, and P. Esquirol. Ordonnancement de tâches sous contraintes: une approche énergétique. *Automatique-productique informatique industrielle*, 26(5-6):453–481, 1992.

[LG95] Philippe Laborie and Malik Ghallab. Planning with Sharable Resource Constraints. In *Proc. of IJCAI*, pages 1643–1651, 1995.

[LI08] Zukui Li and Marianthi Ierapetritou. Process scheduling under uncertainty: Review and challenges. *Computers & Chemical Engineering*, 32(4-5):715–727, 2008. Festschrift devoted to Rex Reklaitis on his 65th Birthday.

[Liu00] J. W. S. Liu. *Real-time systems*. Prentice Hall, 2000.

[LkKC+06] Liu Lurng-kuo, S. Kesavarapu, J. . Connell, A. . Jagmohan, L. . Leem, B. . Paulovicks, V. . Sheinin, . Lijung Tang, and Hangu Yeo. Video Analysis and Compression on the STI Cell Broadband Engine Processor. *Multimedia and Expo, 2006 IEEE International Conference on*, pages 29–32, 9-12 July 2006.

[LL93] G. Laporte and F. V. Louveaux. The integer l-shaped method for stochastic integer programs with complete recourse. *Operations Research Letters*, (13), 1993.

[LM87] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[LM06] Michele Lombardi and Michela Milano. Stochastic Allocation and Scheduling for Conditional Task Graphs in MPSoCs. In Frédéric Benhamou, editor, *Proc. of CP*, volume 4204 of *Lecture Notes in Computer Science*, pages 299–313. Springer, 2006.

[LM07] Michele Lombardi and Michela Milano. Scheduling Conditional Task Graphs. In Christian Bessiere, editor, *Proc. of CP*, volume 4741 of *Lecture Notes in Computer Science*, pages 468–482. Springer, 2007.

[LM09] Michele Lombardi and Michela Milano. A Precedence Constraint Posting Approach for the RCPSP with Time Lags and Variable Durations. In *Proc. of CP*, pages 569–583, 2009.

[LM10] Michele Lombardi and Michela Milano. Allocation and scheduling of Conditional Task Graphs. *Artificial Intelligence*, In Press, Corrected Proof, 2010.

[LMB09] Michele Lombardi, Michela Milano, and Luca Benini. Robust non-preemptive hard real-time scheduling for clustered multicore platforms. In *Proc. of DATE*, pages 803–808, 2009.

[LRS+08] P. Laborie, J. Rogerie, P. Shaw, P. Vilim, and F. Wagner. ILOG CP Optimizer: Detailed Scheduling Model and OPL Formulation. Technical report, 2008.

[LSZ93] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal Speedup of Las Vegas Algorithms. In *Proc. of ISTCS*, pages 128–133, 1993.

[Mar06] Grant Martin. Overview of the MPSoC design challenge. In *Proc. of DAC*, pages 274–279, New York, NY, USA, 2006. ACM.

[MG04] C. T. Maravelias and I. E. Grossmann. Using MILP and CP for the scheduling of batch chemical processes. *Lecture notes in computer science*, pages 1–20, 2004.

[MH04] Michela Milano and Willem Jan van Hoeve. Reduced cost-based ranking for generating promising subproblems. *CoRR*, cs.AI/0407044, 2004.

[MM05] Paul H. Morris and Nicola Muscettola. Temporal dynamic controllability revisited. In *Proc. of AAAI*, pages 1193–1198, 2005.

[MMRB98] A. Mingozzi, V. Maniezzo, S. Ricciardelli, and L. Bianco. An exact algorithm for the resource-constrained project scheduling problem based on a new mathematical formulation. *Management Science*, pages 714–729, 1998.

[MMV01] Paul H. Morris, Nicola Muscettola, and Thierry Vidal. Dynamic control of plans with temporal uncertainty. In *Proc. of IJCAI*, pages 494–502, 2001.

[Moo68] J. Michael Moore. An n Job, One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs. *Management Science*, 15(1):102–109, 1968.

[MRW84] R. H. Möhring, F. J. Radermacher, and G. Weiss. Stochastic scheduling problems I - General strategies. *Mathematical Methods of Operations Research*, 28(7):193–260, 1984.

[MRW85] R. H. Möhring, F. J. Radermacher, and G. Weiss. Stochastic scheduling problems II - set strategies. *Mathematical Methods of Operations Research*, 29(3):65–104, 1985.

[MS00] Rolf H. Möhring and Frederik Stork. Linear preselective policies for stochastic project scheduling. *Mathematical Methods of Operations Research*, 52(3):501–515, 2000.

[Mud01] T. Mudge. Power: A first class design constraint. *Computer*, 2001.

[Mul08] M. Muller. Embedded Processing at the Heart of Life and Style. *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 32–37, Feb. 2008.

[Mus02] Nicola Muscettola. Computing the Envelope for Stepwise-Constant Resource Allocations. In *Proc. of CP*, pages 139–154, 2002.

[MVH04] Laurent Michel and Pascal Van Hentenryck. Iterative Relaxations for Iterative Flattening in Cumulative Scheduling. In *Proc. of ICAPS*, pages 200–208, 2004.

[NA94] W. P. M. Nuijten and Emile H. L. Aarts. Constraint Satisfaction for Multiple Capacitated Job Shop Scheduling. In *Proc. of ECAI*, pages 635–639, 1994.

[NA96] W. P. M. Nuijten and E. H. L. Aarts. A computational study of constraint satisfaction for multiple capacitated job shop scheduling. *European Journal of Operational Research*, 90(2):269–284, 1996.

[NEC] NEC. NEC Multicore Systems. http://www.nec.co.jp/techrep/en/journal/g06/n03/060311.html.

[NPR98] V. I. Norkin, G. Pflug, and A. Ruszczynski. A branch and bound method for stochastic global optimization. *Mathematical Programming*, (83), 1998.

[NS03] K. Neumann and C. Schwindt. Project scheduling with inventory constraints. *Mathematical Methods of Operations Research*, 56(3):513–533, 2003.

[NSZ95] K. Neumann, C. Schwindt, and J. Zimmermann. Resource Constrained Project Scheduling with Time-Windows: Recent Developments and New Applications. *Jòzefowska and Weglarz*, pages 375–407, 1995.

[Nui94] Wim Nuijten. *Time and resource constrained scheduling : a constraint satisfaction approach.* PhD thesis, Technische Universiteit Eindhoven, 1994.

[OYS+07] M. Ohara, . Hangu Yeo, F. Savino, G. . Iyengar, . Leiguang Gong, H. Inoue, H. . Komatsu, V. . Sheinin, S. . Daijavaa, and B. . Erickson. Real-Time Mutual-Information-Based Linear Registration On The Cell Broadband Engine Processor. *Biomedical Imaging: From Nano to Macro, 2007. ISBI 2007. 4th IEEE International Symposium on*, pages 33–36, 12-15 April 2007.

[PAB+05] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock,

S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation CELL processor. *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pages 184–592, 10-10 Feb. 2005.

[PAB+06] D. C. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P. M. Harvey, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D. L. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa. Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. *Solid-State Circuits, IEEE Journal of*, 41(1):179–196, Jan. 2006.

[Pag07] M. Paganini. Nomadik: AMobile Multimedia Application Processor Platform. In *Proc. of ASP-DAC*, pages 749–750, Washington, DC, USA, 2007. IEEE Computer Society.

[PBC04] P. Palazzari, L. Baldini, and M. Coli. Synthesis of pipelined systems for the contemporaneous execution of periodic and aperiodic tasks with hard real-time constraints. 2004.

[PCOS07] Nicola Policella, Amedeo Cesta, Angelo Oddi, and Stephen F. Smith. From precedence constraint posting to partial order schedules: A CSP approach to Robust Scheduling. *AI Commun.*, 20(3):163–180, 2007.

[PEP00] P. Pop, P. Eles, and Z. Peng. Bus access optimization for distributed embedded systems based on schedulability analysis. *Design, Automation and Test in Europe Conference and Exhibition 2000. Proceedings*, pages 567–574, 2000.

[PFF+07] Fabrizio Petrini, Gordon Fossum, Juan Fernández, Ana Lucia Varbanescu, Michael Kistler, and Michael Perrone. Multicore Surprises: Lessons Learned from Optimizing Sweep3D on the Cell Broadband Engine. In *Proc. of IPDPS*, pages 1–10. IEEE, 2007.

[PP92] S. Prakash and A. C. Parker. SOS: Synthesis of application-specific heterogeneous multiprocessor systems* 1. *Journal of Parallel and Distributed computing*, 16(4):338–351, 1992.

[PS92] M. A. Peot and D. E. Smith. Conditional nonlinear planning. In *Proc. of AIPS*, page 189. Morgan Kaufmann Pub, 1992.

[PSTW89] J. H. Patterson, R. Slowinski, F. B. Talbot, and J. Weglarz. An algorithm for a general class of precedence and resource constrained scheduling problems. *Advances in project scheduling*, pages 3–28, 1989.

[PVG] Claude Le Pape, Didier Vergamini, and Vincent Gosselin. Time-versus-capacity compromises in project scheduling.

[PWW69] A. A. B. Pritsker, L. J. Watters, and P. M. Wolfe. Multiproject scheduling with limited resources: A zero-one programming approach. *Management Science*, 16(1):93–108, 1969.

[RDRK09] M. Ranjbar, B. De Reyck, and F. Kianfar. A hybrid scatter search for the discrete time/resource trade-off problem in project scheduling. *European Journal of Operational Research*, 193(1):35–48, 2009.

[Ref04] Philippe Refalo. Impact-Based Search Strategies for Constraint Programming. In Mark Wallace, editor, *Proc. of CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 557–571. Springer, 2004.

[Rég94] Jean-Charles Régin. A Filtering Algorithm for Constraints of Difference in CSPs. In *Proc. of AAAI*, pages 362–367, 1994.

[Rég96] J. C. Régin. Generalized arc consistency for global cardinality constraint. In *Proc. of AAAI*, pages 209–215, 1996.

[Rég03] Jean Charles Régin. *Constraint and integer programming: towards a unified methodology*, chapter Global Constraints and Filtering Algorithms. Kluwer, 2003.

[RGB+06] Martino Ruggiero, Alessio Guerri, Davide Bertozzi, Francesco Poletti, and Michela Milano. Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-on-chip. In Georges G. E. Gielen, editor, *Proc. of DATE*, Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2006, Munich, Germany, March 6-10, 2006, pages 3–8. European Design and Automation Association, Leuven, Belgium, 2006.

[RGB+08] M. Ruggiero, A. Guerri, D. Bertozzi, M. Milano, and L. Benini. A fast and accurate technique for mapping parallel applications on stream-oriented MPSoC platforms with communication awareness. *International Journal of Parallel Programming*, 36(1):3–36, 2008.

[RH98] Bert De Reyck and Willy Herroelen. A branch-and-bound procedure for the resource-constrained project scheduling problem with generalized precedence relations. *European Journal of Operational Research*, 111(1):152–174, 1998.

[RHEvdA05] N. Russell, A. H. M. Hofstede, D. Edmond, and W. M. P. van der Aalst. Workflow data patterns: Identification, representation and tool support. *Lecture Notes in Computer Science*, 3716:353, 2005.

[RLMB08] Martino Ruggiero, Michele Lombardi, Michela Milano, and Luca Benini. Cellflow: A Parallel Application Development Environment with Run-Time Support for the Cell BE Processor. In *Proc. of DSD*, pages 645–650, 2008.

[Rot66] M. H. Rothkopf. Scheduling with random service times. *Management Science*, 12(9):707–713, 1966.

[RWT+06] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies. In *Proc. of WCET*, 2006.

[Sch98] C. Schwindt. *Verfahren zur Lösung des ressourcenbeschränkten Projektdauerminimierungsproblems mit planungsabhängigen Zeitfenstern*. Shaker, 1998.

[SD98] A. Sprecher and A. Drexl. Multi-mode resource-constrained project scheduling by a simple, general and powerful sequencing algorithm. *European Journal of Operational Research*, 107(2):431–450, 1998.

[SDK78] J. P. Stinson, E. W. Davis, and B. M. Khumawala. Multiple Resource–Constrained Scheduling Using Branch and Bound. *IIE Transactions*, 10(3):252–259, 1978.

[Sem] ARM Semiconductor. Arm11 mpcore multiprocessor. Available at http://arm.convergencepromotions.com/catalog/753.htm.

[SFSW09] Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and Mark Wallace. Why Cumulative Decomposition Is Not as Bad as It Sounds. In *Proc. of CP*, pages 746–761, 2009.

[Sha98] Paul Shaw. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. In *Proc. of CP*, pages 417–431, 1998.

[Sim96] H. Simonis. A problem classification scheme for finite domain constraint solving. In *Proc. of CP, Workshop on Constraint Programming Applications: An Inventory and Taxonomy*, pages 1–26, 1996.

[SK01] R. Szymanek and K. Kuchcinski. A constructive algorithm for memory-aware task assignment and scheduling. In *Proc. of CODES*, page 152. ACM, 2001.

[SK03] Dongkun Shin and Jihong Kim. Power-aware scheduling of conditional task graphs in real-time multiprocessor systems. In *Proc. of ISLPED*, pages 408–413, 2003.

[SKD95] A. Sprecher, R. Kolisch, and A. Drexl. Semi-active, active, and non-delay schedules for the resource-constrained project scheduling problem. *European Journal of Operational Research*, 80(1):94–102, 1995.

[SL96] J. Sun and J. Liu. Synchronization protocols in distributed real-time systems. *Distributed Computing Systems, International Conference on*, 0:38, 1996.

[Smi56] W. E. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3(1):59–66, 1956.

[ST] ST. Nomadik Processor. http://us.st.com/stonline/stappl/cms/press/news/year2006/p2004.htm.

[Sto00] F. Stork. Branch-and-bound algorithms for stochastic resource-constrained project scheduling. Technical Report Research Report No. 702/2000, Technische Universitat Berlin, 2000.

[Sto01] F. Stork. *Stochastic resource-constrained project scheduling*. PhD thesis, Technische Universitat Berlin, 2001.

[Sys] CISCO Systems. CISCO Multicore products web page. available at: http://www.cisco.com/en/US/products/ps5763/.

[SZT+04] Todor Stefanov, Claudiu Zissulescu, Alexandru Turjan, Bart Kienhuis, and Ed Deprettere. System Design Using Kahn Process Networks: The Compaan/Laura Approach. In *Proc. of DATE*, page 10340, Washington, DC, USA, 2004. IEEE Computer Society.

[TBHH07] L. Thiele, I. Bacivarov, W. Haid, and Kai Huang. Mapping Applications to Tiled Multiprocessor Embedded Systems. In *Proc. of ACSD*, pages 29–40, july 2007.

[Tec] Cradle Technologies. The multi-core DSP advantage for multimedia. Available at http://www.cradle.com/.

[Tho01] E. S. Thorsteinsson. A hybrid framework integrating mixed integer programming and constraint programming. In *Proc. of CP*, pages 16–30, Paphos, Cyprus, November 2001.

[TMW06] S. Tarim, Suresh Manandhar, and Toby Walsh. Stochastic Constraint Programming: A Scenario-Based Approach. *Constraints*, 11(1):53–80, 2006.

[TPH00] I. Tsamardinos, M. E. Pollack, and J. F. Horty. Merging plans with quantitative temporal constraints, temporally extended actions, and conditional branches. pages 264–272, 2000.

[TVP03] Ioannis Tsamardinos, Thierry Vidal, and Martha E. Pollack. CTP: A New Constraint-Based Formalism for Conditional, Temporal Planning. *Constraints*, 8(4):365–388, 2003.

[TW04] L. Thiele and R. Wilhelm. Design for timing predictability. *Real-Time Systems*, 28(2):157–177, 2004.

[VBC05] Petr Vilim, Roman Bartak, and Ondrej Cepek. Extension of (n log n) Filtering Algorithms for the Unary Resource Constraint to Optional Activities. *Constraints*, 10(4):403–425, 2005.

[vdAHW03] W. M. P. van der Aalst, A. H. M. Hofstede, and M. Weske. Business process management: A survey. *Lecture Notes in Computer Science*, pages 1–12, 2003.

[vdASS] W.M.P. van der Aalst, M.H. Schonenberg, and M. Song. Time Prediction Based on Process Mining. Report BPM-09-04, BPM Center.

[VF99] Thierry Vidal and Hélène Fargier. Handling contingency in temporal constraint networks: from consistency to controllabilities. *J. Exp. Theor. Artif. Intell.*, 11(1):23–45, 1999.

[VHM05] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2005.

[VUPB07] V. A. Varma, R. Uzsoy, J. Pekny, and G. Blau. Lagrangian heuristics for scheduling new product development projects in the pharmaceutical industry. *Journal of Heuristics*, 13(5):403–433, 2007.

[WAHE03a] D. Wu, B. M. Al-Hashimi, and P. Eles. Scheduling and mapping of conditional task graph for the synthesis of low power embedded systems. *IEEE Proceedings-Computers and Digital Techniques*, 150(5):262–73, 2003.

[WAHE03b] D. Wu, B. M. Al-Hashimi, and P. Eles. Scheduling and mapping of conditional task graph for the synthesis of low power embedded systems. *Computers and Digital Techniques, IEE Proceedings* -, 150(5):262–73, 22 Sept. 2003.

[Wal99] Toby Walsh. Search in a Small World. In *Proc. of IJCAI*, pages 1172–1177, 1999.

[Wal02] Toby Walsh. Stochastic Constraint Programming. In Frank van Harmelen, editor, *Proc. of ECAI*, pages 111–115. IOS Press, 2002.

[Wes07] M. Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer-Verlag New York Inc, 2007.

[XMM05] F. Xie, M. Martonosi, and S. Malik. Bounds on power savings using runtime dynamic voltage scaling: an exact algorithm and a linear-time heuristic approximation. In *Proc. of ISPLED*, pages 287–292, San Diego, CA, USA, August 2005.

[XW01] Yuan Xie and Wayne Wolf. Allocation and scheduling of conditional task graph in hardware/software co-synthesis. *Design, Automation and Test in Europe Conference and Exhibition*, 0:0620, 2001.

[YDS95] F. Yao, A. Demers, and S. Shenker. A Scheduling Model for Reduced CPU Energy. In *Proc. of FOCS*, pages 374–382, Milwaukee, WI, USA, October 1995.

[YGO01] B. Yang, J. Geunes, and W. J. O'Brien. Resource-constrained project scheduling: Past work and new directions. *Research Report*, 6, 2001.

[YPB+08] David Yeh, Li-Shiuan Peh, Shekhar Borkar, John Darringer, Anant Agarwal, and Wen-mei Hwu. Thousand-Core Chips. *IEEE Des. Test*, 25(3):272–278, 2008.

[ZTL+03] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, and V. G. da Fonseca. Performance assessment of multiobjective optimizers: an analysis and review. *Evolutionary Computation, IEEE Transactions on*, 7(2):117–132, April 2003.