UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial :

Présentée par

Julien Legriel

Thèse dirigée par **Oded Maler**

préparée au sein **Verimag**
et de **EMSTII**

Multi-Criteria Optimization and its Application to Multi-Processor Embedded Systems

Thèse soutenue publiquement le **4 octobre 2011**,
devant le jury composé de :

Denis Trystram
Professeur à l'INP Grenoble, Président
Lothar Thiele
Professeur à ETHZ, Rapporteur
Eugène Asarin
Professeur à Paris 7, Rapporteur
Philippe Baptiste
Chargé de recherche à l'école Polytechnique, Examinateur
Jean-José Berenguer
Ingénieur à STMicroelectronics, Examinateur
Peter Niebert
Maître de Conférence à L'université de Provence, Examinateur
Oded Maler
Directeur de recherche à Verimag, Directeur de thèse

Acknowledgements

I first want to say that it was a pleasure and great experience to have Oded as advisor. On one hand I had much liberty, but on the other hand Oded was very available to support and guide me during all Ph.D phases (and even thereafter for finding a job !). I hope to have learned from his particular way to look at science in a positive, fresh and ambitious way.

I sincerily thank Lothar Thiele and Eugène Asarin for taking the time to review this thesis, as well as Denis Trystram, Philippe Baptiste and Peter Niebert for their participation to the jury. This thesis has been conducted in the framework of a partnership between Verimag and STMicroelectronics and I would like to acknowledge STMicroelectronics Grenoble members, Bruno Jego, Gilbert Richard, Jean-José Bérenguer and others for their sympathy and technical collaboration.

Verimag laboratory is a really enriching and convivial place for doing a Ph.D, and I would like to thank all people I interacted with and learned from. In particular I am very grateful to Scott Cotton and Colas Le Guernic for their decisive participation in the work on multi-criteria optimization. I also express my gratitude to my Ph.D mates Selma Saidi and Jean-francois Kempf, whose collaboration and moral support were really helpful.

But going daily to Verimag was also a pleasure due to the joyful atmosphere brought by Stéphane, Jean-Noël, Chaouki, Tayeb, Nicolas and all others I forgot. I want to thank these people for all the fun we had in extra-work activities (running, swimming, Grenoble Ekiden, soccer, bowling, poker etc.).

Finally I will end this acknowledgement with a thank you to my family, especially my parents Monique et Alain. I would not have gone so far without their full support from the very beginning.

Contents

Introduction (French)

Dans cette thèse nous développons de nouvelles techniques pour résoudre les problèmes d'optimisation multi-critère. Ces problèmes se posent naturellement dans de nombreux domaines d'application où les choix sont évalués selon différents critères conflictuels (coûts et performance par exemple). Contrairement au cas de l'optimisation classique, de tels problèmes n'admettent pas en général un optimum unique mais un ensemble de solutions incomparables, aussi connu comme le front de Pareto, qui représente les meilleurs compromis possibles entre les objectifs conflictuels. La contribution majeure de la thèse est le développement d'algorithmes pour trouver ou approximer ces solutions de Pareto pour les problèmes combinatoires difficiles. Plusieurs problèmes de ce type se posent naturellement lors du processus de placement et d'ordonnancement d'une application logicielle sur une architecture multi-cœur comme P2012, qui est actuellement développé par STMicroelectronics.

La première classe de méthodes que nous développons s'appuie fortement sur l'existence de puissants solveurs SMT (SAT Modulo Theories), qui peuvent fournir une réponse à l'existence de solutions réalisables pour des systèmes de contraintes mixtes logiques et numériques. Bien que n'étant pas conçu pour résoudre les problèmes d'optimisation explicitement, il est possible d'utiliser ces solveurs comme oracles dans une procédure de recherche binaire pour trouver des solutions optimales. La première contribution de la thèse est le développement d'une telle procédure pour l'optimisation multi-critère, qui peut être vu comme une généralisation multi-dimensionnelle de la recherche binaire. Nous obtenons un algorithme qui fournit une bonne approximation du front de Pareto, donne une garantie sur la qualité de l'approximation, et dirige les requêtes vers les parties inexplorées de l'espace de coûts. Cet algorithme a été implémenté et testé sur des surfaces synthétiques.

La deuxième classe de méthodes que nous considérons est la recherche stochastique locale, où les algorithmes d'optimisation parcourent l'espace des solutions de manière quasi aléatoire, mais avec un biais en faveur des améliorations locales de la fonction de coût. Ces algorithmes font occasionnellement des redémarrages pour éviter de stagner dans la même partie de l'espace, et il s'avère qu'une politique de redémarrage particulière optimise leur performance moyenne. La deuxième contribution de la thèse est l'adaptation de cette politique au contexte multi-critère : à chaque redémarrage, nous changeons la combinaison linéaire des fonctions de coût pour laquelle nous optimisons. Cet algorithme a été implémenté et appliqué à des problèmes de placement quadratique.

Après avoir discuter du rôle de l'optimisation multi-critère dans le déploiement de logiciels sur des architectures multi-processeur, nous nous tournons vers l'application des algorithmes développés à une famille de problèmes d'ordonnancement bi-critère énergie/performance. Nous définissons les problèmes à différents niveaux de complexité, en commençant par des graphes de tâches simples, puis en ajoutant des considérations de communication inter-tâches et de periodicité. Dans tous ces problèmes, nous recherchons

de bons compromis entre la latence de l'ordonnancement et le coût énergétique de la plate-forme qui est liée au nombre de processeurs utilisés et à leur vitesse.

Nous avons d'abord appliqué l'approche basée sur le solveur SMT à un problème dans lequel la vitesse des processeurs peut être configurée, et où l'on minimise le coût énergétique de la plateforme sachant qu'une contrainte est imposée sur la latence de l'ordonnancement. Nous considérons ensuite le problème comme étant bi-objectif et recherchons les compromis entre coût de la plateforme et latence de l'ordonnancement. Enfin nous avons appliqué l'algorithme de recherche locale à une classe similaire de problèmes. Les résultats expérimentaux sont prometteurs.

La thèse est organisé comme suit: le chapitre 1 présente l'optimisation multi-critère, ses motivations, ses concepts et les techniques classiques de résolution. Le chapitre 2 décrit la procédure de recherche utilisant un solveur SMT comme oracle, tandis que le chapitre 3 définit notre procédure de recherche stochastique locale. Le chapitre 4 passe en revue différentes problématiques actuelles liées aux multi-processeur, et notamment au déploiement d'applications parallèles sur ces systèmes. Enfin, le chapitre 5 présente l'application des méthodes d'optimisation proposées dans les chapitres 2 et 3 au problème d'ordonnancement sur architecture multi-processeur.

Introduction

In this thesis we develop new techniques for solving multi-criteria optimization problems. Such problems arise naturally in many (if not all) application domains where choices are evaluated according to two or more conflicting criteria such as price vs. performance. Unlike ordinary optimization, such problems typically do not admit a unique optimum but a set of incomparable solutions, also known as the Pareto Front, which represent the best possible trade-offs between the conflicting goals. The major contribution of the thesis is the development of algorithms for finding or approximating these Pareto solutions for hard combinatorial problems that arise naturally in the process of mapping and scheduling application software on multi-core architectures such as P2012 which is currently being developed by ST Microelectronics.

The first class of methods that we develop relies heavily on the existence of powerful SMT (SAT modulo theories) solvers which can provide a yes/no answer to queries about the existence of feasible solutions to systems of mixed logical and numerical constraints. Although not designed to solve optimization problems explicitly, it is straightforward to use such solvers as oracles inside a binary search procedure for finding optimal solutions. The first major contribution of the thesis is the development of such a procedure for multi-criteria optimization which can be seen as a multi-dimensional generalization of binary search. As a result we obtain an algorithm which provides a good approximation of the Pareto front, gives a guarantee on the quality of approximation and directs the queries towards unexplored parts of the cost space. This algorithm has been implemented and tested on synthetic surfaces.

The second class of methods that we consider is stochastic local search where the optimization algorithms wanders quasi-randomly in the solution space, with a bias toward local improvements of the cost function. Such algorithms occasionally make restarts to get away from stagnation and it turns out that a specific restarting schedule optimizes their expected performance. The second major contribution of the thesis is to adapt this idea to the multi-criteria setting where at each restart we also change the linear combination of cost functions for which we optimize. This algorithm has been implemented and applied to multi-criteria quadratic assignment benchmarks.

After surveying the role of multi-criteria optimization in the deployment of software on multi-processor architectures we turn to the application of the developed methodologies and algorithms to a family of problems related to energy-aware scheduling of task graphs on such platforms. We define problems at different levels of complexity starting with simple task graphs and adding considerations of communication and pipelining. In all these problems we seek good trade-offs between execution speed and the energy cost of the platform which is related to the number of processors and their speeds.

We first apply the satisfiability-based approach starting with a formulation that keeps the deadline as a fixed constraint and search for the cheapest configuration of the architecture

on which the application can be scheduled to meet the deadline. After answering this question on several models we turn to a multi-criteria formulation and apply the binary search procedure to find speed/cost tradeoffs. Finally we apply the new stochastic local search algorithm to this class of problems. The experimental results look promising.

The thesis is organized as follows: Chapter 1 is a survey of multi-criteria optimization, its motivation, concepts and common solution techniques. Chapter 2 describes the search procedure using an SMT solver as an oracle while Chapter 3 defines our stochastic local search procedure. Chapter 4 gives a survey on multi-processor architectures and the crucial role of optimization in the future development process of parallel software. Chapter 5 presents the application of the developed techniques to problems of energy-aware scheduling and reports experimental results, followed by some directions for future work.

Chapter 1

Multi-Criteria Optimization

Résumé : *Le premier chapitre de cette thèse est une introduction à l'optimisation multi-critère. Nous commençons par donner un exemple simple : le placement de tâches sur un système multi-processeur. Généralement, il est nécessaire d'exploiter le maximum de parallelisme et d'avoir un bon équilibrage de la charge de calcul entre les processeurs. D'un autre coté, il faut limiter les communications sur le réseau d'interconnections car celles-ci sont coûteuses en temps et en énergie. Etant donné que ces deux objectifs s'opposent l'un l'autre, le problème du placement des tâches est typiquement un problème d'optimisation multi-critère. Nous nous appuyons sur cet exemple pour introduire le concept fondamental d'optimalité de Pareto. Celui-ci caractérise les compromis optimaux, c'est-à-dire les solutions pour lesquelles il est impossible d'améliorer un objectif sans en détériorer un autre. Nous expliquons aussi pourquoi il est préférable d'utiliser l'approche de résolution dite a posteriori, c'est à dire de rechercher les solutions de Pareto dans un premier temps et de faire un choix parmi ces compromis dans un deuxième temps, plutôt que l'approche a priori, qui consiste en une réduction prématurée du problème au cas uni-critère (en combinant les différentes fonctions objectifs par exemple). Dans la suite du chapitre nous introduisons formellement les concepts fondamentaux (dominance, front de Pareto etc.) puis dédions une partie au problème d'estimation de performance et de comparaison des algorithmes d'optimisation multi-critère. Enfin, nous détaillons et comparons plusieurs types d'approches utilisées pour résoudre ces problèmes.*

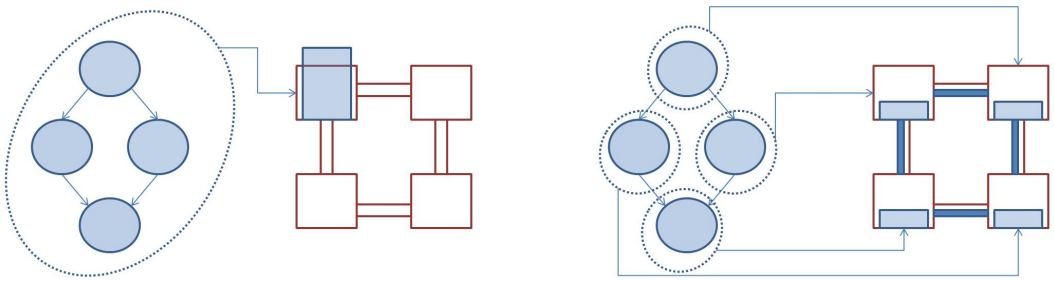
1.1 Introduction

1.1.1 Multi-Criteria Optimization through Examples

Many real-life optimization problems are multi-dimensional by nature. Indeed decisions are more than often made according to multiple and conflicting criteria : time goes against money, work against family care. Consider for instance a cellular phone that we want to purchase. Each product can be evaluated according to its cost, size, power autonomy and various performance measures, and a configuration which is better according to one criterium, can be worse according to another. Consequently, there is no unique optimal solution but rather a set of incomparable alternatives or *trade-offs*. The field of multi-criteria optimization deals with all the methods which take into account more than

one objective/cost function in their problem modeling, and seek to find one particular or a *representative set* of the trade-offs.

Our motivation to look at multi-criteria optimization comes from problems related to the efficient deployment of applications to multi-core systems. Mapping a parallel streaming application on a multi-processor architecture is an example of a problem which involves (at least) two conflicting criteria: balancing processor loads and minimizing communication overhead. A good load balancing limits the outbreak of troubles like overheating, energy leaks and other system faults. On the performance side, it may increase the throughput by maximizing the utilization of the processing capabilities. On the other hand inter-processor communication may be a bottleneck, especially if application tasks have to transfer large amounts of data. A congested link in the network can indeed have a drastic impact on the execution latency. Furthermore these two objectives are conflicting: as we move from a more centralized to a more parallel implementation the imbalance is reduced but the amount of communication increases (Figure 1.1.1). This leads to a set of optimal trade-offs, or *Pareto-optimal* [1] solutions, which are characterized by the fact that their cost cannot be improved in one dimension without being worsened in others. Solutions for which this is not true are said to be *dominated* (Figure 1.1.2) by the Pareto solutions. In the multi-criteria optimization terminology the set of non-dominated solutions is the *Pareto front* and, as we shall explain in the next section, being able to compute or approximate that set is a very useful aid in decision making.



(a) (b)

Figure 1.1.1: (a) A dummy mapping solution with all tasks mapped to the same processor (no parallelism is exploited). The solution is optimal under the objective of minimizing communication, but the imbalance is maximal. (b) A solution where load balancing is optimized which however creates a communication overhead.

1.1.2 On the Relevance of Modeling Several Criteria

When one hears about multi-criteria optimization for the first time, it may appear as an attempt to *postpone* a decision which must eventually be made. Why search for a *set* of trade-offs when only one single solution is going to be chosen and applied at the end? To answer this question we must examine the process of decision making more closely, and point out some limitations of the single-objective framework.

At an abstract level we can view any optimization phase in the decision making process as follows. There is a problem admitting a set S of feasible solutions (which may or may

1. Named after V. Pareto who first defined them in the context of economic theory.

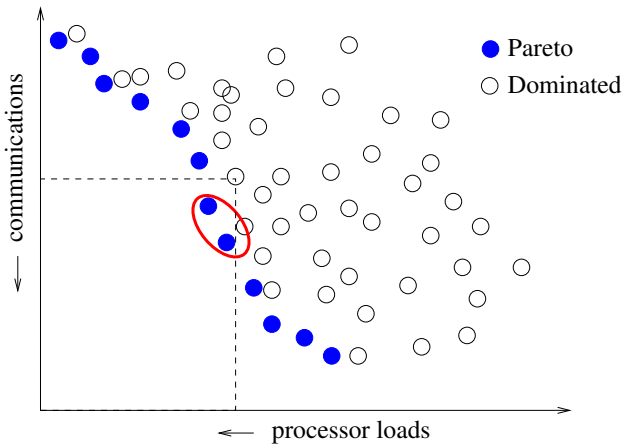


Figure 1.1.2: An example of trade-offs in the cost space for the mapping problem.

not be explicit), and the goal is to find the *best* solution among that set, what implicitly assumes the existence of a total order $<$ on S. The fundamental difference between single and multi-criteria approaches actually lies in the way the total order, which we further refer to as the preference relation, is defined and integrated into the problem model.

A single criteria optimization [2] phase starts with the definition of a utility/cost function $C : S \to \mathbb{R}$ from which a preference relation is directly induced between any pair of solutions: [3]

$$a < b \Leftrightarrow C(a) < C(b) \tag{1.1A}$$

The problem is then reduced to finding the *unique* minimum of a function over an input set S. Multi-criteria problems on the contrary seek to take d cost functions into account for defining the preferences. Therefore we only dispose of a *partial order* $\prec$ among solutions:

$$a \prec b \Leftrightarrow C_1(a) \leq C_1(b) \wedge \ldots \wedge C_d(a) \leq C_d(b) \tag{1.1B}$$

Variants of multi-criteria optimization methodologies may be distinguished on the basis of their approach for obtaining the ultimate total order on S which is derived from the partial order $\prec$. The following classification was first introduced by Cohon and Marks [Coh85].

A priori methods Methods of this category work by reformulating the problem to make it fit within the single criteria framework. The preference relation is therefore defined *a priori*, without having knowledge on the possible trade-offs between the different objectives. The most common a priori method is probably the *weighted sum* approach, which works by combining the different costs into a unique scalar function defined as $C = \lambda_1 C_1 + \ldots + \lambda_d C_d$, with $(\lambda_1 \ldots \lambda_d) \in \mathbb{R}^d$. Another common technique is the *constraint* method, which prescribes to keep a single objective to optimize while moving the others to the constraint side. As a last example, the lexicographic ordering method defines a preference order on the objectives and optimizes each of them successively following the order. All these methods have a common characteristic: they define a *strict* preference order on solutions *before* the search, and then apply single objective optimization.

2. We assume minimization problems.
3. We assume here that no two solutions have *exactly* the same cost.

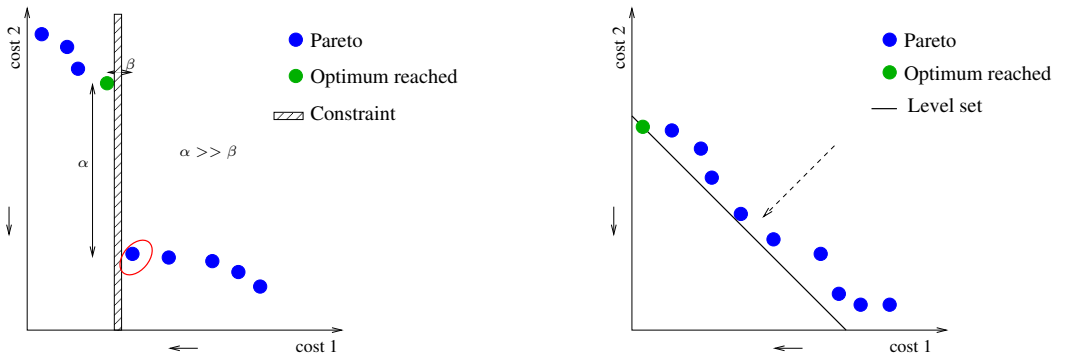


Figure 1.1.3: (a) Because of the constraint imposed on cost 1 the optimum reached is not the circled solution. However this solution would most probably have been preferred when the ratio α/β is large. (b) A weighted sum approach where equal weight is given to each objective. The cost coordinates of the solution returned are however not at all balanced, but biased towards cost 1.

A posteriori methods The a posteriori approach is decomposed into two phases. In the first phase, a specialized algorithm is used to search for a representative set of trade-offs between a (generally small) number of objectives. In the second phase, the decision maker picks a final solution from that set using higher order preferences which were not expressed explicitly in the objective functions.

Interactive methods In this approach, preferences are progressively integrated in the problem model. Unlike a posteriori techniques which work in batch mode, an interactive method presents information sparingly to the user and gets refined preferences from him/her in return. Less effort has been invested in the development of methods in this category and we do not discuss them further in this thesis.

The amount of research on *a posteriori* multi-criteria optimization has been growing during the last decades, essentially because of the belief that it leads to better decision than the *a priori* scheme which is based on an early reduction to single criteria. We try to illustrate this idea by showing the limitations of two of the most common a priori algorithms.

1. *Constraint Method*: One can eliminate an objective of a multi-objective problem by stating a bound on its cost. For instance, the mapping problem can be handled by fixing a limit on the amount of communication and adding this new constraint to the model. The remaining objective (load balancing) is optimized through a standard single criteria optimization algorithm. However choosing an appropriate bound for the communication volume is tricky. The choice may be done based on system constraints and/or a priori aspirations but, as one has no clue about the shape of the cost space, it is possible to miss a good solution (see figure 1.1.3-(a)).
2. *Weighted Sum*: Another intuitive option is to optimize a weighted sum of the objectives. If load balancing is judged as critical as communication traffic, the scaler function to minimize is obtained by multiplying the cost vector by $(1, 1)$. As shown on Figure 1.1.3-(b), choosing weights a priori may lead to a solution which would not have been preferred by the user over the alternatives.

To conclude, a priori methods bear the risk of proposing a solution which would not have been selected if more information on the available trade-offs was available. Besides, humans are usually more comfortable with choosing among a set of alternatives rather than obtaining a solution produced by a black box. The reason for this is quite simple: individual preferences are more *qualitative* than quantitative. They *cannot* be captured by a global utility function, a fact that was first identified by the 19th century sociologist and economist Vilfredo Pareto in the context of consumer theory [Par12]. Before his work, economists were supposing that the utility provided by several consumer goods to an individual could be *quantified* under a global measure (the utility). If the utility brought by a car was of 50 and the one of a cell phone was 10 then it would be 5 times better to have a car than a cell phone. Pareto criticized this view as being unrealistic and proposed a *qualitative* approach where goods are just ranked: a consumer can tell wether a car is better than a cell phone, the opposite, or if it provides equal satisfaction, but no more than that. This approach has been accepted by his peers and is now widely in used in economics. Going back to our matter of decision making, and for the same reasons, we (humans) are more or less able to order a set of cell phones when we see them in a shop using quantitative information (price, size etc) and qualitative information (this one is nice but the other looks more solid and I am clumsy). However it would be inconvenient to blindly quantify our level of satisfaction of any cell-phone within a single mathematical function.

1.2 Formalization

In its general form a multi-criteria optimization problem can be stated as

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & C(x) \\ \text{subject to} \quad & x \in \mathcal{S}. \end{aligned} \tag{1.2C}$$

together with the following definitions :
- $\mathcal{X}$ is the design/decision space
- $x \in \mathcal{X}$ is the decision variables vector
- $\mathcal{S} \subset \mathcal{X}$ is the feasible design space (set of solutions)
- $\mathcal{Y} \subset \mathbb{R}^d$ is the cost space
- $C = (C_1 \dots C_d)$ is the multi-dimensional objective function
- $C_i : \mathcal{X} \to \mathbb{R}$ is the i^{th} objective
- $\mathcal{Z} = C(\mathcal{S}) \subset \mathcal{Y}$ is the feasible cost space

Like single-objective criteria problems multi-criteria can be classified as *continuous* vs. *discrete*, *constrained* vs. *unconstrained*, *linear* vs. *non-linear* etc. The object to minimize is a vector of $\mathbb{R}^d$ so we should provide an order on the elements of that set. Generally the order is not given because the meaning is implicit: solutions are compared using the concept of Pareto dominance already introduced, and which we more formally characterize in the sequel. We also recall some basic vocabulary related to multi-objective optimization. The reader is referred to [Ehr05, Deb01] for a more general introduction to the topic.

Definition 1.2.1 (Domination) *Let c and c' be points in the cost space $\mathcal{Y}$. We say that*

1. *c dominates c' ($c \prec c'$) if for every $i \in [1..d]$, $c_i \leq c'_i$ and $c \neq c'$*

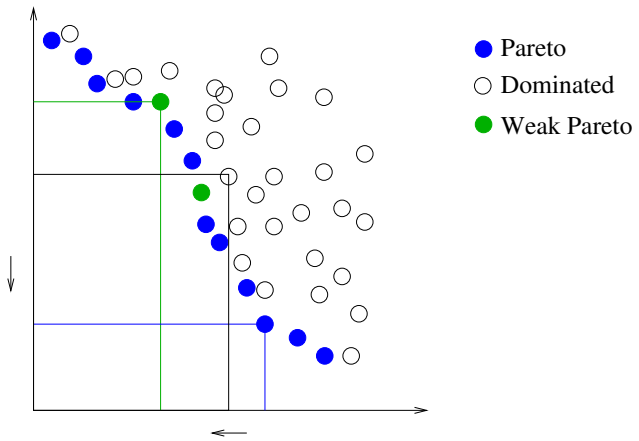


Figure 1.2.1: Examples of dominated, weak Pareto and Pareto solutions.

2. *c strictly dominates c' ($c \ll c'$) if for every $i \in [1..d]$, $c_i < c'_i$*

When neither $c \prec c'$ nor $c' \prec c$ hold we say that c and c' are incomparable which we denote $c||c'$.

The notion of domination, illustrated in Figure 1.2.1, is central in multi-criteria decision making. We will equally speak of domination among points in the cost space and domination among solutions (points in the design space) which is inherited from the relation between their respective costs. A solution s' to a problem which is dominated by s is generally discarded, as s is by default assumed to be always preferred to s'. On the contrary, a solution whose cost vector is not dominated by any point in $\mathcal{Z}$ is optimal in the Pareto sense.

Definition 1.2.2 (Pareto Optimality) *A point x in the design space $\mathcal{X}$ is said to be*

1. *Weakly Pareto optimal (or weakly efficient) if $C(x)$ is not strictly dominated by any other point: $\forall x' \in \mathcal{S}\ C(x') \not\ll C(x)$*
2. *Pareto optimal (or efficient) if $C(x)$ is non-dominated: $\forall x' \in \mathcal{S}$ s.t $C(x') \not\prec C(x)$*

Definition 1.2.3 (Pareto Front) *The Pareto front associated with a multi-objective optimization problem is the set P^* of all Pareto optimal points in the feasible cost space $\mathcal{Z}$.*

The Pareto front represents the set of optimal trade-offs that we would like to compute. There are several characteristics of the problem which complicate this task. First, many combinatorial optimization problems (including those related to mapping and scheduling which we study in this thesis) are NP-hard. Consequently they cannot be solved to optimality by a computer in an efficient manner, already for a single objective. Secondly, adding more objectives makes the resolution even more difficult, especially because the number of Pareto solutions may grow exponentially with the number of objectives. Hence, exact methods do not scale for these problems and the focus is more on designing algorithms that can find an *approximation* of the front, which means a reasonably small set of good representative solutions.

1.3 Quality Assessment

A central question in evaluating the performance of multi-objective optimizers is how to assess and compare the relative quality of their respective outputs. Recall that those algorithms return sets of solutions, so that outputs of different algorithms are more often than not mutually incomparable (Figure 1.3.1). Consequently it is not straightforward to measure the quality of a given technique in generating good approximations of the Pareto front, and finding appropriate *quality indicators* has been an active subject of work leading to the emergence of a whole niche of research on this particular topic. In this section we introduce some key concepts and problematics related to the performance assessment of multi-objective algorithms. We recommend [ZKT08] for a fairly detailed study of the subject.

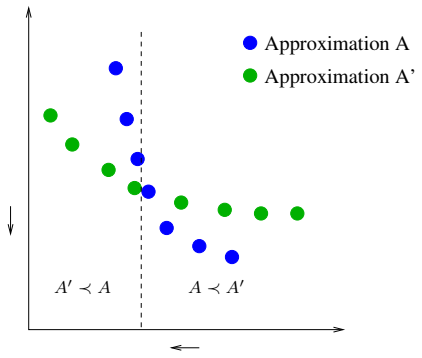


Figure 1.3.1: A and A' are incomparable approximations. There is a part of the space where each of them dominates the other.

1.3.1 Desirable Properties

There is quite a consensus on two informal properties that a good set of non-dominated solutions should verify. On one hand it has to contain individual solutions which are not far from being Pareto optimal. This way, the user does not lose a lot by making decisions according to the approximate solutions rather than the exact ones. The second property concerns the diversity of solutions in the set. As we pointed out before, it is crucial for making the right decision to have a good understanding of the available tradeoffs, so the approximation must ideally not overlook parts of the cost space where Pareto points can be found.

Developing an algorithm which finds a good approximation is therefore a problem involving two objectives that may be conflicting: closeness to the true Pareto points and diversity. While the first objective is maximized when the algorithm focuses on an *in depth* exploration of the cost space and local improvements of found solutions, the second necessitates to widen the search *in breadth*.

1.3.2 Quality Indicators

The weakness of the Pareto dominance relation in comparing different approximations is due to the fact that some of them remain incomparable. We would like indicators which,

while capturing the above considerations induce a total order over the set of approximations of a given problem. In the sequel we characterize a Pareto front approximation as being a non-dominated set of points in the cost space and give a definition of a quality indicator.

Definition 1.3.1 (Pareto Front Approximation) *An approximation of a Pareto front in a feasible cost space $\mathcal{Z}$ is a set $A \in 2^{\mathcal{Z}}$ which contains only mutually non-dominated points i.e. $\forall c, c' \in A\ c||c'$. We denote the set of all such approximations as $\Phi_{\mathcal{Z}}$.*

Definition 1.3.2 (Quality Indicator) *A quality indicator is a function $I : \Phi_{\mathcal{Z}} \to \mathbb{R}$ which assigns a real value to each approximation, the bigger being the better.*

A quality indicator is thus a function which associates a quantitative measure of goodness to each approximating set, in the same way a cost function ranks each solution in single criteria optimization. Many variants have been proposed and we present two of them which are commonly used.

Definition 1.3.3 (Epsilon Approximation) *Given c, c' in cost space and $\epsilon \in \mathbb{R}^+$*

1. *(Additive): c ϵ-approximates c' ($c \approx_\epsilon^+ c'$) if $\forall i \in [1..d]\ c_i \leq c'_i + \epsilon$*
2. *(Multiplicative): c ϵ-approximates c' ($c \approx_\epsilon^* c'$) if $\forall i \in [1..d]\ c_i \leq \epsilon.c'_i$*

Intuitively the ϵ-approximation relation expresses the fact that a vector c is not worse than c' by more than ϵ in all objective (whether it be in terms of an additive increase or a ratio). As an example a two dimensional vector $(100.1, 40)$ is a 0.1-approximation of $(100, 100)$ in the additive sense. The value of ϵ therefore quantifies the quality with which c approximates c' (Figure 1.3.2-(a)).

Definition 1.3.4 (Epsilon Indicator) *Let $R \in 2^{\mathcal{Y}}$ be a predefined reference set and $\epsilon \in \mathbb{R}^+$. For any $A \in \Phi_{\mathcal{Z}}$ the additive version of the ϵ-indicator is*

$$I_\epsilon^+(A) = \inf_{\alpha \in \mathbb{R}^+} \{\forall r \in R\ \exists c \in A\ c \approx_\alpha^+ r\}$$

The multiplicative version is defined analogously.

The epsilon-indicator value is then the smallest α such that every point in the reference set R is α-approximated by *at least one* solution of the approximation set (Figure 1.3.2-(b)). Note that the true Pareto front would be an ideal reference set but in practice we need, of course to provide another reference set. A solution we used in the course of this thesis is to take as a reference the non-dominated union of the approximations returned by the different methods we wish to compare.

Another popular indicator measures the *volume* of the portion of the cost space dominated by an approximation (Figure 1.3.3).

Definition 1.3.5 (Hypervolume Indicator) *Let $r \in \mathbb{R}^d$ be a reference point and $A \in \Phi_{\mathcal{Z}}$ an approximating set. Let $B^+(A, r) = \{z \in \mathbb{R}^d,\ \exists c \in A\ c \preceq z \preceq r\}$, namely the set of points dominated by A and which dominate r. The hypervolume indicator is the quantity*

$$I_H(A, r) = V(B^+(A, r))$$

where V indicates volume.

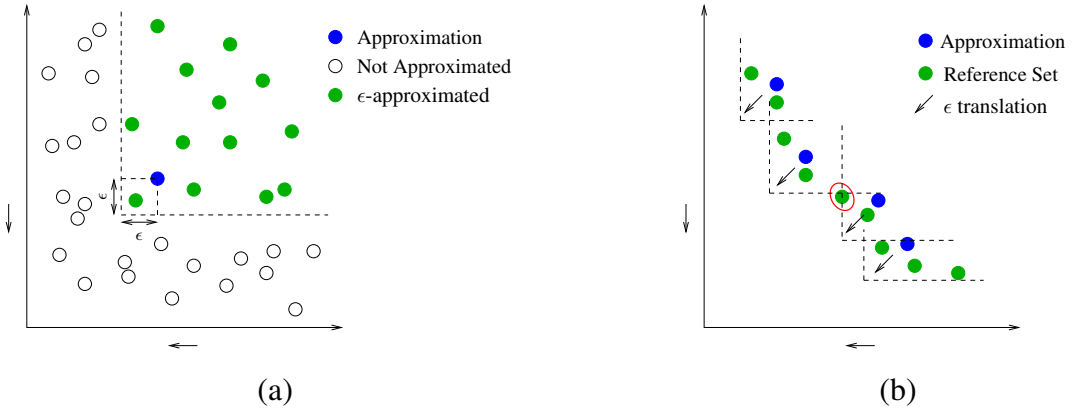


Figure 1.3.2: (a) Illustration of the additive epsilon approximation. (b) The reference set is ϵ-approximated because any of its points is approximated by at least one solution. Futhermore ϵ is the smallest such value (the circled point is not approximated if ϵ is decreased).

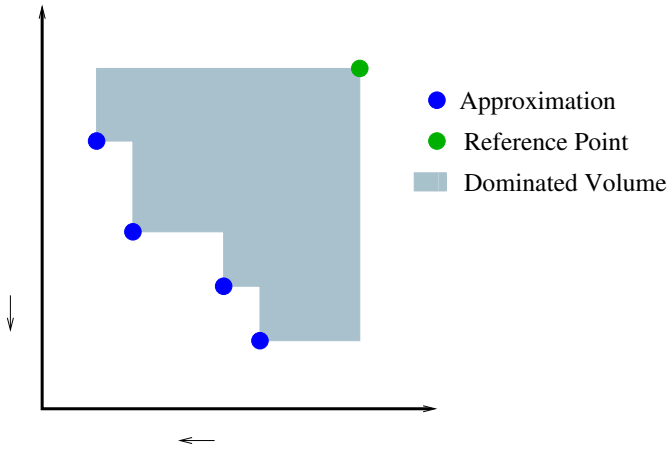


Figure 1.3.3: Illustration of the hypervolume indicator. The value according to this indicator is the volume of the shaded polygone.

This measure was first introduced in the context of multi-objective optimization by Zitzler and Thiele [ZT98] who also stated its compliance with the dominance relation: an approximation set A_1 which is strictly better than A_2 always covers a bigger hypervolume. This also appears to have been formally proved for arbitrary dimension [Fle03].

Proposition 1.3.1 (Pareto Compliance of the Volume Indicator) *Let $A_1, A_2 \in \Phi_{\mathcal{Z}}$ be two approximations such that A_1 dominates A_2 i.e*

- $\forall c_2 \in A_2\ \exists c_1 \in A_1\ c_1 \preceq c_2$
- $A_1 \neq A_2$

Then $\forall r \in \mathbb{R}^d$, $I_H(A_1) > I_H(A_2)$.

Proposition 1.3.1 particularly entails that the Pareto front of a multi-criteria optimization problem is the *only* set to achieve the maximal value of the hypervolume indicator. This observation gave birth to a big interest in studying the indicator both theoretically and practically, all the more that other available measures do not share this property [ZTL+03]. Furthermore there is a growing trend in using the hypervolume indicator online to guide the algorithm in its search. However, one can notice that calculating the indicator is not a simple task because it involves computing the volume of a complex object (union of boxes) in $\mathbb{R}^d$. This is easily achieved if $d = 2$ since the complexity is linear with the size of the approximation, a little bit more tricky if $d = 3$, but the best currently known algorithms working with arbitrary dimensions have exponential complexity (regarding dimension). Recent contributions [BF10, BF09] further state that the problem is $\#P$-complete along with some unapproximability results. Consequently, unless $P = NP$ there is no hope for a polynomial algorithm, although the best shown lower bound (derived from reducing the problem to a special case of the *Klee's Measure Problem*) is $\Omega(n \log n)$ with n the size of the approximation [BFLI+09]. The search for fast algorithms computing the hypervolume is therefore currently oriented towards stochastic methods using monte-carlo simulations [BF09, BZ11].

1.3.3 Comparing Non-deterministic Algorithms

The performance assessment measures we have presented so far neglect an important aspect: many optimization heuristics are randomized, meaning that they may return different outputs on different runs. If the variability is high it is more relevant to compare the *average* performance of two algorithms in solving a particular problem on several runs.

For instance one can run the program multiple times on a given input, compute the quality of each output with an indicator, and analyze the statistics (mean, variance, hypothesis testing) for deciding which algorithm is better. This does not provide a mean for comparing outputs graphically though, and plots showing several approximations are usually hard to interpret.

Another approach allowing graphical analysis is based on the computation of a so called *attainment function* [FF96]. Given a problem π and a program τ solving π, each point z in cost space has a certain probability denoted $\alpha_{\tau,\pi}(z)$ of being dominated (or attained) by the approximation returned on a particular run. A simple estimator of $\alpha_{\tau,\pi}(z)$ is obtained by simply computing the percentage of runs on which z gets dominated by the approximation (Figure 1.3.4).

The attainment function method is in particular used for building two or three dimensional graphical representations, those of which allow to compare the performance of two algorithms graphically. Let us first expose the concept of *attainment surface*.

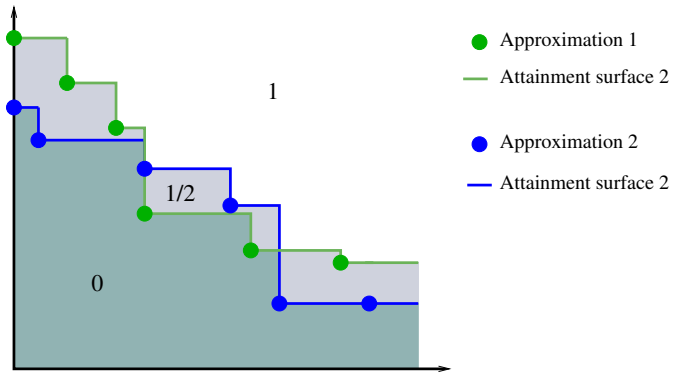


Figure 1.3.4: Estimation of the attainment function using two approximations returned on different runs. Points in the lower part are not attained by any of the approximations and thus get a probability value of zero. On the contrary points in the upper parts are attained by the two and are associated a probability of one. Points in the middle get one half.

Definition 1.3.6 (Attainment Surface) *Given an approximation A its attainment surface Δ_A is the set of weakly non-dominated elements of $B^+(A)$, where $B^+(A)$ is the subspace of $\mathcal{Z}$ dominated by A.*

The attainment surface is actually the boundary between the part dominated by the approximation and the rest of the cost space (Figure 1.3.4). Running the stochastic algorithm several times provides many different surfaces from which we can derive a kind of average : the *k% attainment surface*. This is defined as the boundary between two distinct parts of the cost space, the one where points were attained on more than k% of the runs and its complementary. It can be approximated by techniques suggested in [KC00]. A set of diagonals emanating from the origin is defined and the intersection with every surface is computed for each of them. Supposing that there is a total of N surfaces, an approximation of the p/N attainment surface ($0 \leq p \leq N$) is obtained as the attainment surface defined by the p^{th} point on each line (starting from the origin). This is illustrated on Figure 1.3.5. In three dimensions this method leads to a sorely vizualisable surface but an alternative approach was proposed to cope with the problem [Kno05].

Another analysis consists in performing a statistical test on each line in order to decide (with a certain confidence) that an algorithm τ did better than another τ', the opposite, or that no conclusion can be made with the confidence level chosen. Results are presented in the form of a pair (a, b) where a is the percentage of lines where τ is better than τ' and b the percentage where τ' is better than τ. A big value of $1-(a+b)$, which is the percentage of lines where no decision could be made, means it is hard to conclude that one algorithm was superior than the other. This approach may enable to identify parts of the cost space where an algorithm performs particularly well or bad.

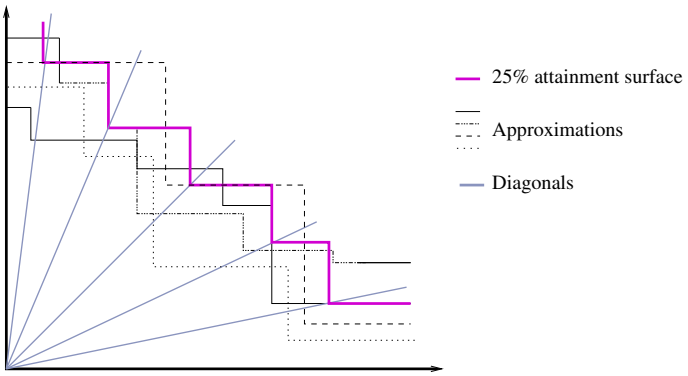


Figure 1.3.5: Approximation of a 25% attainment surface. There are four approximations so the third point is picked on each diagonal emanating from the origin.

1.4 Existing Solution Methods

1.4.1 Classical Methods

In this section we review several popular techniques that work according to the same principle : they generate a set of different trade-offs by converting the multi-criteria problem π into a parametrized single criteria version $\pi'(\lambda)$. By solving the latter problem for different values of λ many non-dominated solutions can be obtained. Most techniques are categorized both as a priori and a posteriori methods depending on whether $\pi'(\lambda)$ is solved using one or more values of the parameter λ. However, for reasons discussed in previous section we are more interested on the a posteriori usage of these techniques. Hence we are interested in the ability of each method to generate a well distributed set of non-dominated solutions when the parameter is varied.

1.4.1.1 Weighted Sum

The weighted sum method is probably the most widespread approach due to its simplicity. A scalar cost function is defined as an aggregation of costs using weights as defined below.

Definition 1.4.1 (λ-Aggregation) *Let $C = (C_1, \ldots, C_d)$ be a d-dimensional function and let $\lambda = (\lambda_1, \ldots \lambda_d)$ be a vector such that*

1. $\forall j \in [1..d], \lambda_j > 0$
2. $\sum_{j=1}^{d} \lambda_j = 1$.

The λ-aggregation of C is the function $C^\lambda = \sum_{j=1}^{d} \lambda_j C_j$.

Intuitively, the components of λ represent the relative importance (weight) one associates with each objective. The parametrized problem $\pi'(\lambda)$ in this case is

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & C^\lambda(x) \\ \text{subject to} \quad & x \in \mathcal{S}. \end{aligned} \tag{1.4D}$$

A fundamental result about the weighted sum method is that the solution of $\pi'(\lambda)$ is properly Pareto efficient for any λ with strictly positive coordinates.

Definition 1.4.2 (Proper Pareto Optimality (Geoffrion, 1968)) *A point $x^* \in \mathcal{X}$ is called properly Pareto optimal if it is Pareto optimal and there is a real number $M > 0$ such that for all $x \in \mathcal{X}$ satisfying $C_i(x) < C_i(x^*)$ for some i, there exists an index j with $C_j(x^*) < C_j(x)$ such that*

$$\frac{C_i(x^*) - C_i(x)}{C_j(x) - C_j(x^*)} \leq M$$

The properly Pareto property is stronger than Pareto optimality. It excludes the Pareto solutions which achieve a small improvement in one objective at the expense of a drastic deterioration of others.

Theorem 1A (Geoffrion (1968)) *Let $\lambda \in \mathbb{R}^d$ such that $\forall j \in [1..d], \lambda^j > 0$. Then the optimal solution $x^* \in \mathcal{X}$ of 1.4D is properly efficient.*

The theorem gives a guarantee that the solutions generated with the weighted sum method are good. Actually the reason may intuitively be understood if we look at the cost space of a linear problem. In this particular situation, linearity implies that the level sets are parallel hyperplanes perpendicular to the weight vector. Finding the optimum is equivalent to shifting the hyperplane in the direction of the weight vector, reaching the optimum when no more feasible points are located under the hyperplane (Figure 1.4.1). When the problem is not linear the level sets have a more complex shape but the property still holds. A reciprocal theorem also exists but it necessitates an additional convexity hypothesis.

Theorem 1B (Geoffrion (1968)) *Let the decision space $\mathcal{X} \subset \mathbb{R}^n$ be convex and C_i $i \in [1..d]$ be convex functions. Then $x^* \in \mathcal{X}$ is properly efficient iff there exists $\lambda \in \mathbb{R}^d$ with strictly positive coordinates such that x^* is the optimal solution of 1.4D.*

This result is the basis for exact methods solving multi-criteria linear programs, including extensions of the well known simplex algorithm to multiple objectives [Ehr05]. Solutions which can be found as solution to (1.4D) are termed *supported*. Theorem 1B can thus be rephrased as: any Pareto solution of a convex problem is supported. Unfortunately this does not hold anymore if the convexity hypothesis is relaxed. Indeed the weighted sum method cannot reach points situated on locally concave parts of a Pareto front, a fact that can also be understood graphically (figure 1.4.1). This is problematic since many concrete problems have a non convex feasible set.

Furthermore, choosing the set of λ parameters appropriately is another non trivial issue. In practice, a uniform sampling of the set Λ of weight vectors satisfying the conditions of Definition 1.4.1, is typically used. For any multi-criteria problem π with m different Pareto solutions, Λ can be partitioned into m subsets $\Lambda^1 \dots \Lambda^m$, each consisting of weight vectors that lead to the same point. The structure of this partition is a priori unknown and it may vary a lot depending on the problem. Therefore it is not always the case that a uniform sampling of Λ will generate a well distributed set of solutions. The work described in Chapter 3 provides a solution to this problem.

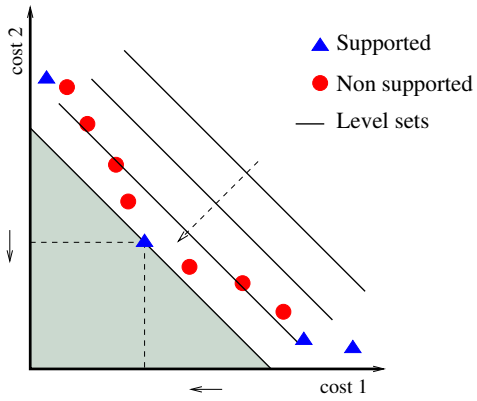


Figure 1.4.1: The optimum of 1.4D is such that there are no solutions under the level set line, which is why it is Pareto optimal. Only supported solutions are optimum under some scalarization of the objectives using weights.

1.4.1.2 Epsilon-Constraint Method

The epsilon-constraint method [HLW71] consists in solving subproblems of the following form

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & C_k(x) \\ \text{subject to} \quad & x \in \mathcal{S}, \\ & \forall i \neq k \; C_i(x) \leq \epsilon_i. \end{aligned} \tag{1.4E}$$

for some fixed k and varying ϵ_i values. The idea is to minimize one of the objective function while the others are put into constraints as outlined in 1.1.2. Unlike the weighted sum method this technique is theoretically able to reach any Pareto point of a non-convex problem. This can be observed on figure 1.1.3 where the optimum point lies on a concave part of the front. Nonetheless the epsilon-constraint method features two drawbacks:

1. The optimum of 1.4E is only garanted to be weakly Pareto [Ehr05]
2. It is difficult to find interesting values of the parameter ϵ. In particular the problem may become unfeasible due to the new constraints on objective functions.

1.4.1.3 Weighted Metric Method

Another series of techniques seeks to minimize the weighted distance to an ideal point which is impossible to reach.

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \left(\sum_{i=1}^{d} \lambda_i |C_i(x) - c^*|^p \right)^{\frac{1}{p}} \\ \text{subject to} \quad & x \in \mathcal{S}. \end{aligned} \tag{1.4F}$$

In Equation 1.4F point $c^* \in \mathcal{Y}$ is the ideal solution defined as follows: for all i c_i^* is the value obtained when minimizing the function C_i individually i.e $c_i^* = \min_{x \in \mathcal{S}} C_i(x)$. For $p = 1$ this gives the weighted sum method, for $p = 2$ the distance is a weighted euclidian

distance in $\mathbb{R}^d$. The formulation with $p = \infty$, which is known as the *weighted Tchebycheff* problem, is shown in Equation 1.4G.

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \max_{i=1\ldots d} \lambda_i |C_i(x) - c^*| \\ \text{subject to} \quad & x \in \mathcal{S}. \end{aligned} \tag{1.4G}$$

This method is theoretically interesting because it can find any Pareto point, even for non-convex problems.

1.4.1.4 Normal Boundary Intersection Method

Normal Boundary Intersection (NBI) [DD98] is a methodology originally designed to tackle non-linear continuous optimization problems. The method seeks to find evenly distributed points on the Pareto frontier. It works by projecting elements of a specific convex hull of points (the convex hull of individual minima defined below) towards the *boundary* of the feasible cost space $\mathcal{Z}$.

Definition 1.4.3 (Convex Hull of Individual Minima (CHIM)) *For every i in $[1..d]$ let x_i^* be the point in $\mathcal{X}$ which minimizes C_i. Let $c^i = C(x_i^*)$ and $c^* = (C_1(x_1^*) \ldots C_d(x_d^*))^T$ the ideal solution. The convex hull of the individual minima (CHIM) is the set*

$$CHIM = \{M\omega : \omega \in \mathbb{R}^d, \sum_{i=1}^{d} \omega_i = 1\}$$

where M is a $d \times d$ matrix whose i^{th} column is $c^i - c^*$.

Formally the NBI method consists in solving the following subproblems

$$\begin{aligned} \underset{x}{\text{maximize}} \quad & t \\ \text{subject to} \quad & x \in \mathcal{S}, t \in \mathbb{R}, \\ & M\omega + tn = C(x) - c^*. \end{aligned} \tag{1.4H}$$

where $n = M(-1 \ldots -1)^T$ is a *quasi-normal* vector pointing backwards towards the origin, and ω is a vector in $[0, 1]^d$ which is going to be varied. In the equations above $M\omega$ actually represent an arbitrary point in the CHIM whereas $M\omega + tn$ is a point on the line whose direction is the quasi-normal and which goes through $M\omega$. By imposing $M\omega + tn = C(x) - c^*$ in the model and maximizing the t, the solution to the subproblem 1.4H is the intersection of the line with the boundary of $\mathcal{Z}$. This is probably best understood by looking at a figure (1.4.2).

On the contrary to previously described techniques we can expect that by choosing a uniformly distributed set of ω in $[0, 1]^d$ we would obtain an *evenly* distributed set of solutions on the Pareto front. This nice feature makes the method really appealing in practice. Furthermore it has been proved that the method is independent of the relative scales of the objectives, meaning that the solution to 1.4H does not change if objective functions are multiplied by arbitrary constants.

One drawback of the Normal Boundary Intersection technique is that it may return points which are dominated if the boundary is concave. Indeed the Pareto front only matches the boundary if the latter is convex (figure 1.4.2). Moreover when $d > 2$ some of the Pareto points are unreachable by program 1.4H, at least if we stick to parameters

verifying $\sum_{i=1}^{d} \omega_i = 1$. A simple example from which this can be seen was given in [DD98]. Consider $\mathcal{Z}$ as being a sphere in $\mathbb{R}^3$ which touches every axes. There exists Pareto points which cannot be obtained by projecting a point from the CHIM. Instead they may be reached starting from a point on the affine hyperplan containing the CHIM, denoted CHIM$^+$. Another possible limitation is the need for the individual minima x_i^* $i \in [1..d]$, as those points may be hard to compute.

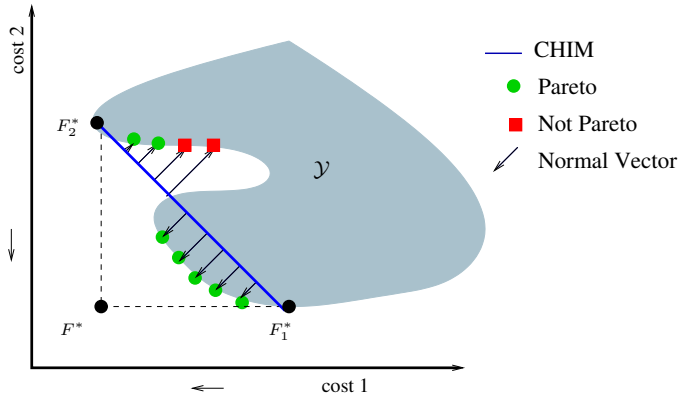


Figure 1.4.2: Illustration of the NBI method. Points are projected from the CHIM to the boundary of $\mathcal{Y}$. Some of the points obtained are not Pareto, because the boundary is not convex.

1.4.2 Evolutionary Algorithms

An evolutionary optimization algorithm is a metaheuristic which mimics the biological evolution of species in order to solve an optimization problem. Solutions are interpreted as the genotypes of individuals in a population which may be recombined (crossover) and mutated. Those algorithms also apply the principle of survival of the fittest, meaning that they feature a selection mechanism in favor of the best individuals in the population.

Among the class of evolutionary algorithm, *genetic algorithms* are well-known metaheuristics particulary well suited for tackling combinatorial optimization problems. When the search space is discrete, solutions are generally encoded as bit strings where each bit represents a gene. Each individual in the population is also attributed a fitness value using the objective functions, and this determines its chances of survival. A genetic algorithm starts from an initial population (randomly generated) and successively applies three genetic operators to give birth to an offspring.

Reproduction. The purpose of this phase is to build a mating pool containing the most promising solutions. This is therefore a selection step in which some solutions are selected based on their fitness value (note that a single solution may be added twice to the mating pool). A common method for doing this is the tournament selection, in which a tournament is repeatedly played between two members of a population, and the winner (the one with better fitness value) is added to the pool.

Crossover. The elements of the pool are then recombined using a crossover operator. Typically a crossover consists in creating one or two new individuals by mixing the genes of two parents picked in the mating pool. As an example a one-point crossover

first cuts the genotypes of both parents at the same point, and uses these pieces to build two children (Figure 1.4.3).

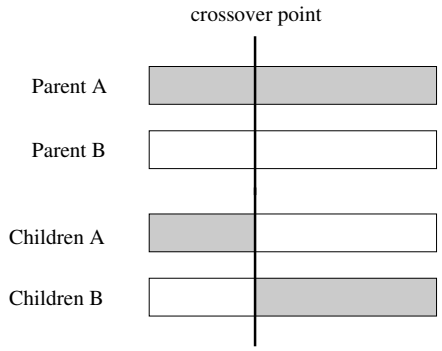


Figure 1.4.3: Example of a one-point crossover.

Mutation. Additionally, each created individual is mutated with a certain probability called the mutation rate. A mutation is just a random change in the string representing the individual. For example, if the representation is a bit-vector, a possible mutation flips one or more bits at random. Generally this phase brings better diversity in the population than the crossover does, since the latter is just a (quite simple) recombination of existing genes.

These three operators are used to give birth to a new generation of individuals. The new population is constituted by either the offspring generated or a selection of the best individuals of the offspring *and* the previous population (in the second case the algorithm is said to be *elitist*). The whole process is repeated a predefined number of times, which corresponds to the number of generations the algorithm will create. The working principle of a genetic algorithm is illustrated on Figure 1.4.4 which is borrowed from [Deb01].

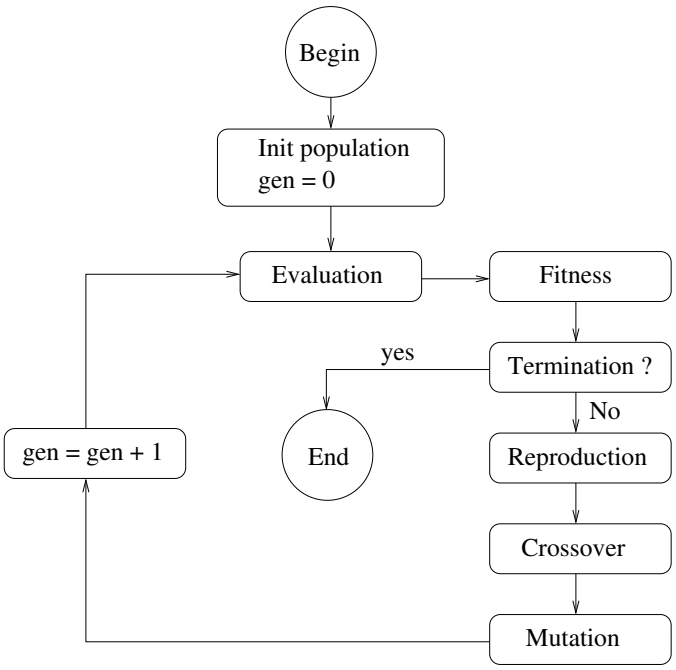


Figure 1.4.4: Genetic Algorithm flowchart.

Chapter 2

Satisfiability-Based Approximation Algorithms

Résumé : *Ce chapitre détaille la première contribution de la thèse. Il s'agit d'une méthodologie permettant d'approximer le front de Pareto d'un problème d'optimisation multi-critère, en utilisant un solveur SMT (Satisfiability Modulo Theories). Nous avons développé un algorithme qui génère des appels successifs au solver avec différentes contraintes sur les objectifs du problème, et maintient deux ensembles : celui des solutions non-dominées obtenues jusque lors (qui constitue l'approximation du front de Pareto), et celui des points "unsat", qui définit une zone de l'espace des objectifs au delà de laquelle aucune solution ne peut être trouvée. L'intérêt de garder cette zone unsat est double : d'une part, la distance entre l'approximation du front de Pareto et la zone unsat définit une borne sur la qualité de l'approximation, et d'autre part la recherche est orientée uniquement vers les zones de l'espaces des objectifs ou il y a éventuellement des solutions. Une des contributions majeures de cette partie est le développement d'un algorithme de recherche dichotomique multi-dimensionnel que nous utilisons afin de minimiser la distance entre l'approximation et la zone unsat. Cet algorithme de recherche, qui est basé sur des structures géométriques particulières, est indépendant du fait que nous traitons un problème d'optimisation multi-critère. Il s'agit seulement de minimiser la distance entre les frontières de deux ensembles multi-dimensionnel (ayant toutefois une structure particulière), en procédant par dichotomie. Il pourrait donc potentiellement servir dans d'autres contextes.*

2.1 Introduction

In this chapter we present our first contribution, a general methodology for approximating the Pareto front of multi-criteria optimization problems. Our search-based methodology consists of submitting queries to a constraint solver. Hence, in addition to a set of solutions, we can guarantee *bounds* on the distance to the actual Pareto front and use this distance to guide the search.

Multiple-criteria or multi-objective optimization problems have been studied since the dawn of modern optimization using diverse techniques, depending on the nature of the underlying optimization problems (linear, nonlinear, combinatorial) [Ste86, EG00, FGE05,

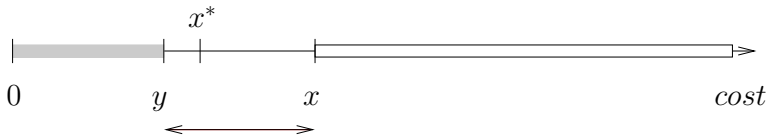


Figure 2.1.1: Example of satisfiability-based one-dimensional optimization. The lower part has been shown $unsat$ and the higher part is $dominated$ by x. An upper bound on the distance of x to the optimum x^* is given by the value $x - y$.

Ehr05]. One approach consists of defining an aggregate one-dimensional cost/utility function by taking a weighted sum of the various costs. Each choice of a set of coefficients for this sum will lead to an optimal solution for the one-dimensional problem which is also a Pareto solution for the original problem (cf. Chapter 1). Another popular class of techniques is based on heuristic search, most notably genetic/evolutionary algorithms [Deb01, ZT98], which are used to solve problems related to design-space exploration of embedded systems, the same problems that motivate our work. As we outlined in Chapter 1, a major issue in these heuristic techniques is finding meaningful measures of *quality* for the sets of solutions they provide [ZTL$^+$03].

We explore an alternative approach to solve the problem based on *satisfiability solvers* that can answer whether there is an assignment of values to the decision variables which satisfies a set of constraints. It is well known, in the single-criterium case, that such solvers can be used for optimization by searching the space of feasible costs and asking queries of the form: *is there a solution which satisfies the problem constraints* **and** *its cost is not larger than some constant?* Asking such questions with different constants we obtain both positive (*sat*) and negative (*unsat*) answers. Taking the minimal cost x among the *sat* points and the maximal cost y among the *unsat* points we obtain *both* an approximate solution x and an upper bound $x - y$ on its distance from the optimum, that is, on the quality of the approximation (Figure 2.1.1).

We extend this idea to multi-criteria optimization problems. Our goal is to use the *sat* points as an *approximation* of the Pareto front of the problem, use the *unsat* points to guarantee *computable bounds* on the distance between these points and the actual Pareto front and to *direct* the search toward parts of the cost space so as to *reduce* this distance (Figure 2.1.2). To this end we define an appropriate metric on the cost space as well as efficient ways to recompute it incrementally as more *sat* and *unsat* points accumulate. A prototype implementation of our algorithm also demonstrates the quality and efficiency of our approach on synthetic Pareto fronts.

The rest of the chapter is organized as follows. Section 2 defines the problem setting including the notions of distance between the *sat* and *unsat* points which guides our search algorithm. In Section 3 we describe some fundamental properties of certain points on the boundary of the *unsat* set (*knee points*) which play a special role in computing the distance to the *sat* points, and show how they admit a natural tree structure. In Section 4 we describe our exploration algorithm and the way it updates the distance after each query. Section 5 reports our implementation and experimental results on some purely-synthetic

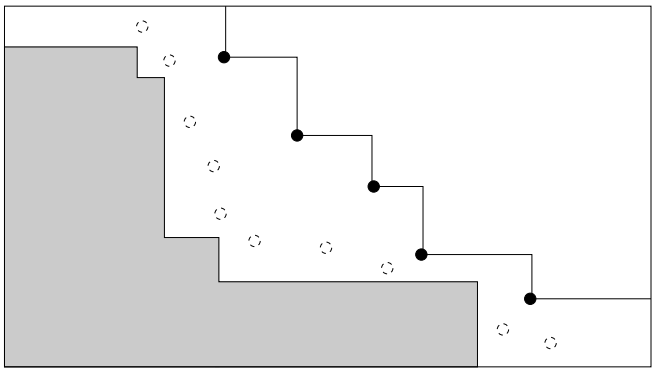


Figure 2.1.2: Multi-dimensional search. The unsat part (down left) is a lower bound to the approximation (black points) of the Pareto front (dashed points).

benchmarks of varying dimension and accuracy [1].

2.2 Preliminary Definitions

In the context of satisfiability-based optimization, a constrained problem (we use *minimization* henceforth) may be stated as the following formulation of definition 1.2C:

$$\begin{array}{ll} \underset{x}{\text{minimize}} & C(x) \\ \text{subject to} & \varphi(x). \end{array} \tag{2.2A}$$

where x is a vector of decision variables, φ is a set of *constraints* on the variables that define which solution is considered feasible and C is a cost function defined over the decision variables. We reformulate the problem by moving costs to the constraint side, that is, letting $\varphi(x, c)$ denote the fact that x is a feasible solution whose cost is c. Hence the optimum is

$$\min\{c : \exists x\ \varphi(x, c)\}.$$

Moving to multi-criteria optimization, c becomes a d-dimensional vector $(c_1, \ldots c_d)$ that we assume, without loss of generality, [2] to range over the bounded hypercube $\mathcal{Y} = [0, 1]^d$, that we call the *cost space*. In the sequel we use notation $\mathbf{r}$ for $(r, \ldots, r)$.

We assume that the maximal cost $\mathbf{1}$ is feasible and that any cost with some $c_i = 0$ is infeasible. This is expressed as an initial set of *unsat* points $\{\mathbf{0}_i\}_{i=1..d}$ where $\mathbf{0}_i$ is a point with $c_i = 0$ and $c_j = 1$ for every $j \neq i$. The set $\mathcal{Y}$ is a lattice with a partial-order relation defined as:

$$s \leq s' \equiv \forall i\ s_i \leq s'_i \tag{2.2B}$$

Pairs of points such that $s \not\leq s'$ and $s' \not\leq s$ are said to be *incomparable*, denoted by $s||s'$. The strict version of $\leq$ is

$$s < s' \equiv s \leq s' \wedge \exists j\ s_j < s'_j \tag{2.2C}$$

1. The technique is later on used to solve scheduling problems where we show the trade-offs between execution time and power consumption (Chapter 5).

2. One can normalize the cost functions accordingly.

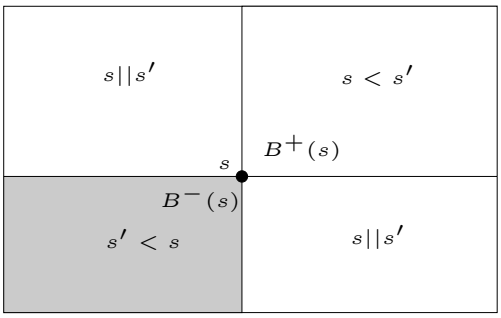


Figure 2.2.1: A point s and its backward and forward cones.

meaning that s strictly improves upon s' in at least one dimension without being worse on the others. In this case we say that s *dominates* s'. We will make an assumption that if cost s is feasible so is any cost $s' > s$ (one can add a slack variable to the cost). The *meet* and *join* on $\mathcal{Y}$ are defined as

$$s \sqcap s' = (\min\{s_1, s'_1\}, \ldots, \min\{s_d, s'_d\})$$
$$s \sqcup s' = (\max\{s_1, s'_1\}, \ldots, \max\{s_d, s'_d\})$$

We say that a point in the cost space s' is an i-*extension* of a point s if $s'_i > s_i$ and $s'_j = s_j$ for every $i \neq j$.

A point s in a subset $S \subseteq \mathcal{Y}$ is *minimal* if it is not dominated by any other point in S, and is *maximal* if it does not dominate any point in S. We denote the sets of minimal and maximal elements of S by $\underline{S}$ and $\overline{S}$, respectively. We say that a set S of points is domination-free if all pairs of elements $s, s' \in S$ are incomparable, which is true by definition for $\underline{S}$ and $\overline{S}$. The domination relation associates with a point s two *rectangular cones* $B^+(s)$ and $B^-(s)$ consisting of points dominated by (resp. dominating) s:

$$B^-(s) = \{s' \in \mathcal{Y}, s' < s\} \text{ and } B^+(s) = \{s' \in \mathcal{Y}, s < s'\}.$$

These notions are illustrated in Figure 2.2.1. Note that both $B^-(s) \cup \{s\}$ and $B^+(s) \cup \{s\}$ are closed sets. If cost s is feasible it is of no use to look for solutions with costs in $B^+(s)$ because they are worse than s. Likewise, if s is infeasible, we will not find solutions in $B^-(s)$.[3] We let $B^-(S)$ and $B^+(S)$ denote the union of the respective cones of the elements of S and observe that $B^+(S) = B^+(\underline{S})$ and $B^-(S) = B^-(\overline{S})$.

Suppose that we have performed several queries and the solver has provided us with the sets S_0, and S_1 of *unsat* and *sat* points, respectively. Our state of knowledge is summarized by the two sets $K_1 = B^+(S_1)$ and $K_0 = B^-(S_0)$. We know that K_1 contains no Pareto points and K_0 contains no solutions. The domain for which S_0 and S_1 give us *no information* is $\tilde{K} = (\mathcal{Y} - K_0) \cap (\mathcal{Y} - K_1)$. We use $bd(K_0)$ and $bd(K_1)$ to denote the boundaries between $\tilde{K}$ and K_0 and K_1, respectively. It is the "size" of $\tilde{K}$ or the *distance* between the boundaries $bd(K_0)$ and $bd(K_1)$ which determines the quality of our current approximation (Figure 2.2.2). Put another way, if S_1 is our approximation of the Pareto surface, the boundary of K_0 defines the limits of potential improvement of the approximation, because no solutions can be found beyond it. This can be formalized as an

3. Note that the query is formulated as $c \leq s$ and if the problem is discrete and there is no solution whose cost is exactly s, the solver would provide a solution with $c = s' < s$ if such a solution exists.

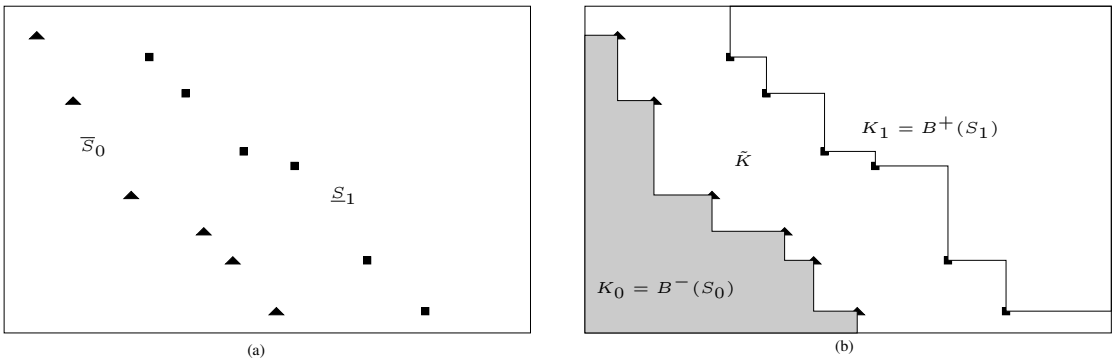


Figure 2.2.2: (a) Sets S_0 and S_1 represented by their extremal points $\overline{S}_0$ and $\underline{S}_1$; (b) The gaps in our knowledge at this point as captured by K_0, K_1 and $\tilde{K}$. The actual Pareto front is contained in the closure of $\tilde{K}$.

appropriate (directed) distance between S_1 and K_0. Note that no point in S_1 can dominate a point in K_0.

Definition 2.2.1 (Directed Distance between Points and Sets) *The directed distance $\rho(s, s')$ between two points is defined as*

$$\rho(s, s') = \max\{s'_i \dot{-} s_i : i = 1..d\},$$

where $x \dot{-} y = x - y$ when $x > y$ and 0 otherwise. The distance between a point s and a set S' is the distance between s to the closest point in S':

$$\rho(s, S') = \min\{\rho(s, s') : s' \in S'\}.$$

The Hausdorff directed distance between two sets S and S'

$$\rho(S, S') = \max\{\rho(s, S') : s \in S\}.$$

In all these definitions we assume $s' \not\preceq s$ for any $s \in S$ and $s' \in S'$.

In other words

$$\rho(S, S') = \max_{s \in S} \min_{s' \in S'} \max_{i=1..d} s'_i \dot{-} s_i.$$

Definition 2.2.2 (ϵ-Approximation) *A set of points S is an ϵ-approximation of a Pareto front P if $\rho(P, S) \leq \epsilon$.*

It may be noted that the definition of ϵ-approximation is related to the notion of additive ϵ-indicator presented in Chapter 1. A set is an ϵ-approximation if and only if the value of the indicator with the Pareto set as reference is less than ϵ. Now, since the Pareto surface is bounded from below by $bd(K_0)$ we have:

Observation 1 *Consider an optimization problem such that S_0 is included in the set of infeasible solutions, with $K_0 = B^-(S_0)$. Then any set S_1 of solutions which satisfies $\rho(bd(K_0), S_1) \leq \epsilon$ is an ϵ-approximation of the Pareto set P.*

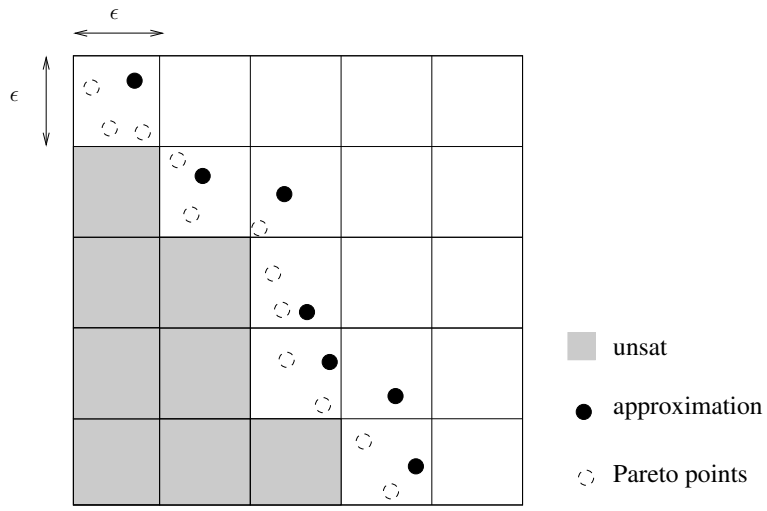


Figure 2.2.3: ϵ-approximation with the grid.

Our goal is to obtain an ϵ-approximation of P by submitting as *few* queries as possible to the solver. To this end we will study the structure of the involved sets and their distances. We are not going to prove new complexity results because the upper and lower bounds on the number of required queries are almost tight:

Observation 2 (Bounds)

1. *One can find an ϵ-approximation of any Pareto front $P \subseteq \mathcal{Y}$ using $(1/\epsilon)^d$ queries;*
2. *Some Pareto fronts cannot be approximated by less than $(1/\epsilon)^{d-1}$ points.*

Proof 1 *For (1), similarly to [PY00], define an ϵ-grid over $\mathcal{Y}$, ask queries for each grid point and put them in S_0 and S_1 according to the answer. Then take S_1 as the approximation whose distance from $bd(S_0)$ is at most $1/\epsilon$ by construction (Figure 2.2.3). For (2), consider a "diagonal" surface*

$$P = \{(s_1, \ldots, s_d) : \sum_{i=1}^{d} s_i = 1\}$$

which has dimension $d-1$. ∎

Remark: The lower bound holds for continuous Pareto surfaces. In discrete problems where the solutions are sparse in the cost space one may hope to approximate P with less than $(1/\epsilon)^d$ points, maybe with a measure related to the actual *number* of Pareto solutions. However since we do not work directly with P but rather with S_0, it is not clear whether this fact can be exploited. Of course, even for continuous surfaces the lower bound is rarely obtained: as the orientation of the surface deviates from the diagonal, the number of needed points decreases. A surface which is almost axes-parallel can be approximated by few points.

Updating the distance $\rho(bd(K_0), S_1)$ as more *sat* and *unsat* points accumulate is the major activity of our algorithm hence we pay a special attention to its efficient implementation. It turns out that it is sufficient to compute the distance $\rho(G, S_1)$ where G is a finite set of special points associated with any set of the form $B^-(S)$.

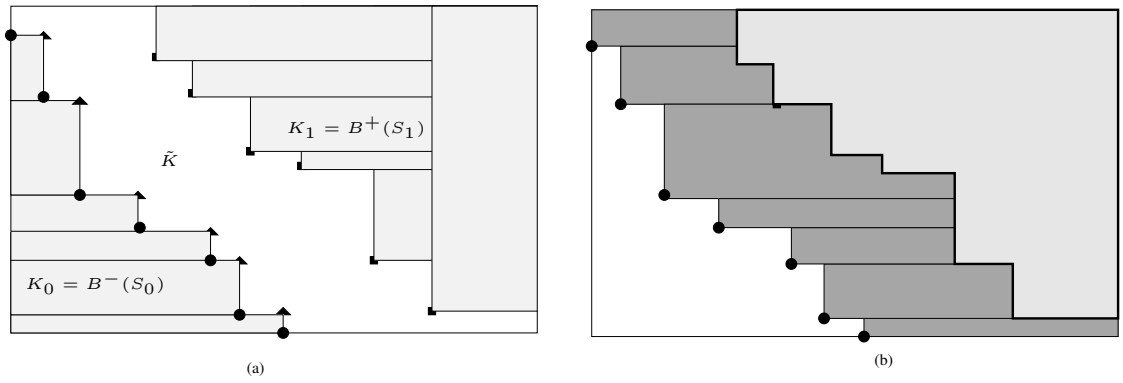


Figure 2.3.1: (a) Knee points, denoted by circles; (b) Knee points viewed as the minimal points of $\mathcal{Y} - K_0$.

2.3 Knee Points

Definition 2.3.1 (Knee Points) *A point s in $bd(K_0)$ is called a knee point if by subtracting a positive number from any of its coordinates we obtain a point in the interior of K_0. The set of all such points is denoted by G.*

In other words the knee points, illustrated in Figure 2.3.1-(a), represent the most *unexplored corners* of the cost space where the maximal potential improvement resides. This is perhaps best viewed if we consider an alternative definition of G as the minimal set such that $\mathcal{Y} - int(K_0) = cl(B^+(G))$ (Figure 2.3.1-(b)). Since $\rho(s, s')$ can only increase as s moves *down* along the boundary we have:

Observation 3 (Distance and Knee Points) $\rho(bd(K_0), S_1)) = \rho(G, S_1)$.

Our algorithm keeps track of the evolution of the knee points as additional *unsat* points accumulate. Before giving formal definitions, let us illustrate their evolution using an example in dimension 2. Figure 2.3.2-(a) shows a knee point g generated by two *unsat* points s^1 and s^2. The effect of a new *unsat* point s on g depends, of course, on the relative position of s. Figure 2.3.2-(b) shows the case where $s \not> g$: here knee g is not affected at all and the new knees generated are extensions of other knees. Figure 2.3.2-(c) shows two *unsat* points dominated by g: point s^5 induces two extensions of g and point s^6 which does not. The general rule is illustrated in Figure 2.3.2-(d): s will create an extension of g in direction i iff $s_i < h_i$ where h_i is the extent to which the hyperplane perpendicular to i can be translated forward without eliminating the knee, that is, without taking the intersection of the d hyperplanes outside K_0.

Let S be a set of incomparable points and let $\{s^1, \ldots, s^d\} \subseteq S$ be a set of d points such that for every i and every $j \neq i$ $s^i_i < s^j_i$. The *ordered meet* of $s^1, \ldots, s^d$ is

$$[s^1, \ldots, s^d] = (s^1_1, s^2_2, \ldots, s^d_d). \tag{2.3D}$$

Note that this definition coincides with the usual meet operation on partially-ordered sets, but our notation is ordered, insisting that s^i attains the minimum in dimension i. The knee points of S are maximal elements of the set of points thus obtained (Figure 2.3.3). With every knee $g \in G$ we associate a vector h defined as $h = \langle s^1, s^2, \ldots, s^d \rangle = (h_1, \ldots, h_d)$ with $h_i = \min_{j \neq i} s^j_i$ for every i, characterizing the extendability of s in direction i.

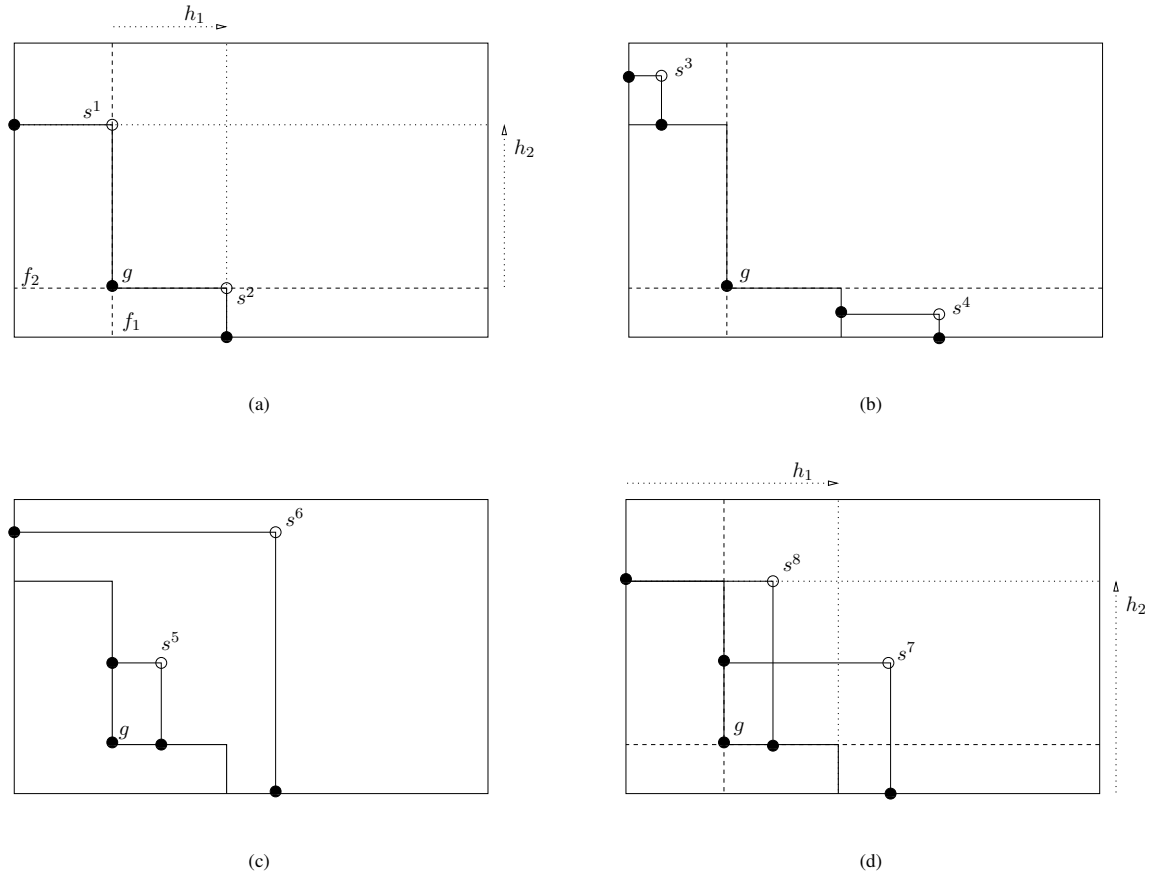


Figure 2.3.2: (a) A knee g generated by s^1 and s^2. It is the intersection of the two hyperplanes f_1 and f_2 (dashed lines); (b) new *unsat* points s^3 and s^4 which are not dominated by g and have no influence on it; (c) new *unsat* points s^5 and s^6 which are dominated by g and hence eliminate it as a knee point. Point s^5 generates new knees as "extensions" of g while the knees generated by s^6 are not related to g; (d) point s^7 generates an extension of g in direction 2 and point s^8 generates an extension in direction 1. These are the directions i where the coordinates of the *unsat* points are strictly smaller than h_i (dotted lines).

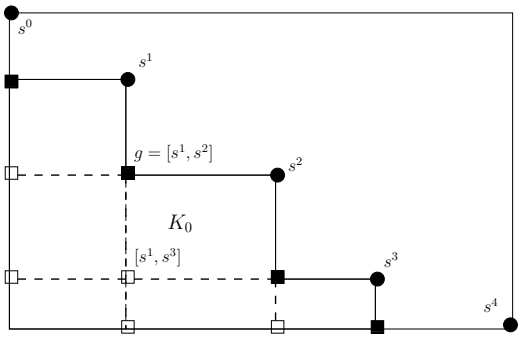


Figure 2.3.3: Knee points are the maximal elements obtained by the meet of d unsat points. The meet $[s^1, s^2]$ is maximal and therefore defines a knee point g, whereas $[s^1, s^3]$ does not. Dimension two is a degenerate case, since knee points are easily computable by taking the meet of any two unsat successors: in our example knee points correspond to $[s^0, s^1]$, $[s^1, s^2]$, $[s^2, s^3]$, and $[s^3, s^4]$. In arbitrary dimension generating knee points is not trivial.

Proposition 2.3.1 (Knee Generation) *Let S be a set of* unsat *points with a set of knees G, let s be a new* unsat *point and let G' be the new set of knees associated with $S \cup \{s\}$. Then the following holds for every $g \in G$ such that $g = [s^1, \ldots, s^d]$ and $h = \langle s^1, s^2, \ldots, s^d \rangle$*

1. *Knee g is kept in G' iff $g \not< s$*
2. *If $g \in G - G'$, then for every i such that $s_i < h_i$, G' contains a new knee g', the i-descendant of g, defined as $g' = [s^1, \ldots, s, \ldots, s^d]$, extending g in direction i.*

Before describing the tree data structure we use to represent the knee points let us make another observation concerning the potential contribution of a new *sat* point in improving the minimal distance to a knee or a set of knees.

Observation 4 (Distance Relevance) *Let g, g^1 and g^2 be knee points with $\rho(g, S) = r$, $\rho(g^1, S) = r^1$ and $\rho(g^2, S) = r^2$ and let s be a new* sat *point. Then*

1. *The distance $\rho(g, s) < r$ iff $s \in B^-(g + \mathbf{r})$*
2. *Point s cannot improve the distance to any of $\{g^1, g^2\}$ if it is outside the cone $B^-((g^1 + \mathbf{r^1}) \sqcup (g^2 + \mathbf{r^2}))$.*

Note that for the second condition, being in that cone is necessary but not sufficient. The sufficient condition for improving the distance of at least one of the knees is $s \in B^-(g^1 + \mathbf{r^1}) \cup B^-(g^2 + \mathbf{r^2})$ as illustrated in Figure 2.3.4.

We represent G as a tree whose nodes are either leaf nodes that stand for current knee points, or other nodes which represent points which were knees in the past and currently have descendant knees that extend them (Figure 2.3.5). A node is a tuple

$$N = (g, [s^1, \ldots, s^k], h, (\mu^1, \ldots, \mu^k), r, b)$$

where g is the point, $[s^1, \ldots, s^k]$ are its *unsat* generators and h is the vector of its extension bounds. For each dimension i, μ^i points to the i-descendant of N (if such exists) and the set of all direct descendants of N is denoted by μ. For leaf nodes $N.r = \rho(N.g, S_1)$ is just the distance from the knee to S_1 while for a non-leaf node $N.r = \max_{N' \in N.\mu} N'.r$, the maximal distance to S_1 over all its descendants. Likewise $N.b$ for a leaf node is the

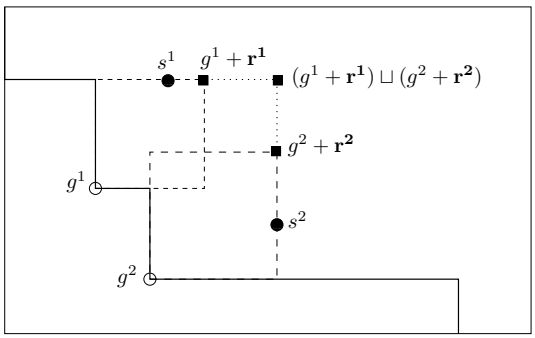


Figure 2.3.4: Two knees g^1 and g^2 and their respective nearest points s^1 and s^2. Points outside the upper dashed square will not improve the distance to g^1 and those outside the lower square will not improve the distance to g^2. Points outside the enclosing dotted rectangle can improve neither of the distances.

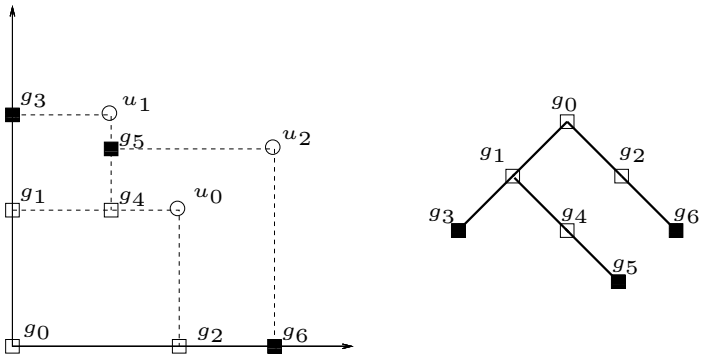


Figure 2.3.5: Example of a knee tree in two dimensions, after adding successively three unsat points u_0, u_1 and u_2. Transparent squares are past knee points that have descendants, and black squares are current knees on $bd(K_0)$. Note that in the depicted situation, g_2 and g_4 have become useless in the tree: they are not knee points anymore and they have a single descendant. In practice our algorithm discard these points.

maximal point such that any *sat* point in the interior of its back cone improves the distance to $N.g$. For a non leaf node $N.b = \bigsqcup_{N' \in N.\mu} N'.b$, the join of the bounds associated with its descendants.

2.4 The Algorithm

Algorithm 2.4.1 iteratively submits queries to the solver in order to decrease the distance between S_1 and G.

The initialization procedure lets $S_0 = \{\mathbf{0}_1, \ldots, \mathbf{0}_d\}$, $S_1 = \{(1, \ldots, 1)\}$ and hence initially $G = \{g^0\}$ with $g^0 = [\mathbf{0}_1, \ldots, \mathbf{0}_d] = (0, \ldots, 0)$ and $h = \langle \mathbf{0}_1, \ldots, \mathbf{0}_d \rangle = (1, \ldots, 1)$. The initial distance is $\rho(G, S_1) = 1$. The *update-sat* and *update-unsat* procedures recompute distances according to the newly observed point by propagating s through the knee tree. In the case of a *sat* point, the goal is to track the knee points g such that $\rho(g, s) < \rho(g, S_1)$, namely points whose distance has decreased due to s. When s is an *unsat* point, we have

Algorithm 2.4.1 Approximate Pareto Surface

```
initialize
repeat
  select(s)
  query(s)    %  ask whether there is a solution with cost ≤ s
  if sat then
    update-sat(s)
  else
    update-unsat(s)
  end if
until ρ(G, S_1) < ε
```

to update G (removing dominated knees, adding new ones), compute the distance from the new knees to S_1 as well as the new maximal distance. The algorithm stops when the distance is reduced beyond ϵ. Note that since $\rho(G, S_1)$ is maintained throughout the algorithm, even an impatient user who aborts the program before termination will have an approximation guarantee for the obtained solution.

The propagation of a new *sat* point s is done via a call to the recursive procedure *prop-sat*(N_0, s) where N_0 is the root of the tree (Algorithm 2.4.2).

Algorithm 2.4.2 Prop-Sat

```
procedure prop-sat(N, s)
if s < N.b then {s may reduce the distance to N.g or its descendants}
  r := 0    %    temporary distance over all descendants
  b := 0    %  temporary bound on relevant sat points
  if N.μ ≠ ∅ then {a non-leaf node}
    for every i s.t. N' = N.μ^i ≠ ∅ do {for every descendant }
      prop-sat(N', s)
      r := max{r, N'.r}
      b := b ⊔ N'.b
    end for
  else {leaf node }
    r := min{N.r, ρ(N.g, s)}    %  improve if s is closer
    b := N.g + r
    N.r := r
    N.b := b
  end if
end if
```

The propagation of a new *unsat* point s, which is more involved, is done by invoking the recursive procedure *prop-unsat*(N_0, s) (Algorithm 2.4.3). The procedure returns a bit ex indicating whether the node still exists after the update (is a knee or has descendants).

The *prop-unsat* procedure has to routinely solve the following sub problem: given a knee point g and a set of non-dominating *sat* points S, find a point $s \in S$ nearest to g and hence compute $\rho(g, S)$. The distance has to be non negative so there is at least one dimension i such that $g_i \leq s_i$. Hence a lower bound on the distance is

$$\rho(g, S) \geq \min\{s_i - g_i : (i = 1..d) \wedge (s \in S) \wedge (s_i \geq g_i)\},$$

Algorithm 2.4.3 Prop-Unsat

procedure prop-unsat(N, s)
$ex := 1$
if $N.g < s$ **then** {knee is influenced}
  $ex := 0$ % temporary existence bit
  $r := 0$ % temporary distance over all descendants
  $b := \mathbf{0}$ % temporary relevance bound
  if $N.\mu \neq \emptyset$ **then** {a non-leaf node}
    for every i s.t. $N' = N.\mu^i \neq \emptyset$ **do** {for every descendant}
      ex' :=prop-unsat(N', s)
      if $ex' = 0$ **then**
        $N.\mu^i := \emptyset$ % node N' is removed
      else
        $ex := 1$
        $r := \max\{r, N'.r\}$
        $b := b \sqcup N'.b$
      end if
    end for
  else {leaf node}
    for $i = 1..d$ **do**
      if $s_i < N.h_i$ **then** {knee can extend in direction i }
        $ex := 1$
        create a new node $N' = N.\mu^i$ with
        $N'.g = [N.s^1, \ldots, s, \ldots, N.s^k]$
        $N'.h = \langle N.s^1, \ldots, s, \ldots, N.s^k \rangle$
        $N'.r = \rho(N'.g, S_1)$
        $N'.b = N'.g + N'.r$
        $N'.\mu^i = \emptyset$ for every i
        $r := \max\{r, N'.r\}$
        $b := b \sqcup N'.b$
      end if
    end for
    $N.r := r$
    $N.b := b$
  end if
end if
return (ex)

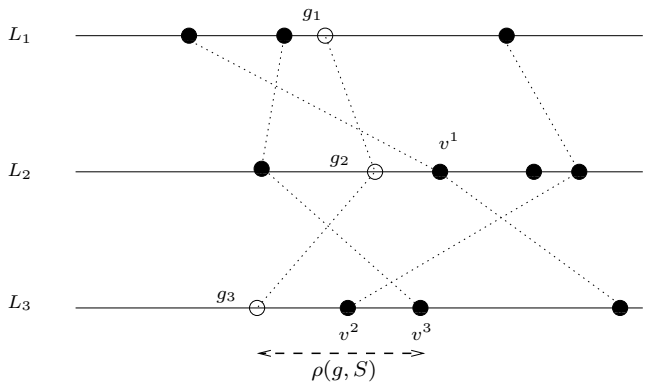


Figure 2.4.1: Finding the nearest neighbor of g: the first candidate for the minimal distance is v^1, the nearest projection which is on dimension 2 , but the point associated with it has a larger distance on dimension 3; The next candidate is v^2, the closest in dimension 3 but the corresponding point also has larger coordinates. Finally, the point associated with v^3, the next value on L_3, has all its distances in other dimensions smaller and hence it is the closest point which defines the distance (dashed line).

and an upper bound is

$$\rho(g, S) \leq \max\{s_i - g_i : (i = 1..d) \wedge (s \in S)\}.$$

We now present (informally) an algorithm and a supporting data structure for computing this distance. Let $(L_i, <_i)$ be the linearly-ordered set obtained by projecting $S \cup \{g\}$ on dimension i. For every $v \in L_i$ let $\Theta_i(v)$ denote all the points in S whose i^{th} coordinate is v. Let $\sigma^i(s)$ be the successor of s_i according to $<_i$, that is, the smallest s'_i such that $s_i < s'_i$. Our goal is to find the minimal value v in some L_i such that there exists $s \in \Theta_i(v)$, s_i defines the maximal distance to g, that is, $s_i - g_i > s_j - g_j$ for every $j \neq i$.

The algorithm keeps a frontier $F = \{f_1, \ldots, f_d\}$ of candidates for this role. Initially, for every i, $f_i = \sigma^i(g_i)$, the value next to g_i and the candidate distances are kept in $\Delta = \{\delta_1, \ldots, \delta_d\}$ with $\delta_i = f_i - g_i$. The algorithm is simple: each time we pick the minimal $\delta_i \in \Delta$. If for some $s \in \Theta_i(f_i)$ and for every $j \neq i$ we have $s_j - g_j < s_i - g_i$ then we are done and found a nearest point s with distance δ_i. Otherwise, if every $s \in \Theta(f_i)$ admits some j such that $s_j - g_j > s_i - g_i$ we conclude that the distance should be greater than δ_i. We then let $f_i = \sigma^i(f_i)$, update δ_i accordingly, take the next minimal element of Δ and so on. This procedure is illustrated in Figure 2.4.1. The projected order relations are realized using an auxiliary structure consisting of d ordered lists.

2.4.1 The Selection Procedure

Selecting the next query to ask is an important ingredient in any heuristic search algorithm, including ours. We employ the following simple rule. Let g and s be a knee and a *sat* point whose distance $\rho(g, s)$ is maximal and equal to $r = s_i - g_i$ for some i. The next point for which we ask a query is $s' = s + \mathbf{r}/2$. If s' turns out to be a *sat* point, then the distance from g to S_1 is reduced by half. If s' is an *unsat* point then g is eliminated and is replaced by zero or more new knees, each of which is r-closer to S_1 in one dimension. For the moment we do not know to compute an upper bound on the worst-case number

of queries needed to reach distance ϵ except for some hand-waving arguments based on a discretized version of the algorithm where queries are restricted to the ϵ-grid. Empirically, as reported below, the number of queries was significantly smaller than the upper bound.

2.5 Experimentation

We have implemented Algorithm 1 and tested it on numerous Pareto fronts produced as follows. We generated artificial Pareto surfaces by properly intersecting several convex and concave halfspaces generated randomly. Then we sampled $10,000$ points in this surface, defined the Pareto front as the boundary of the forward cone of these points and run our algorithm for different values of ϵ. Figure 2.5.1 shows how the approximate solutions and the set of queries vary with ϵ on a 2-dimensional example. One can see that indeed, our algorithm concentrates its efforts on the neighborhood of the front. Table 2.1 shows some results obtained as follows. For every dimension d we generate several fronts, run the algorithm with several values of ϵ, compute the average number of queries and compare it with the upper bound $(1/\epsilon)^d$. As one can see the number of queries is only a small fraction of the upper bound. Note that in this class of experiments we do not use a constraint solver, only an oracle for the feasibility of points in the cost space based on the generated surface.

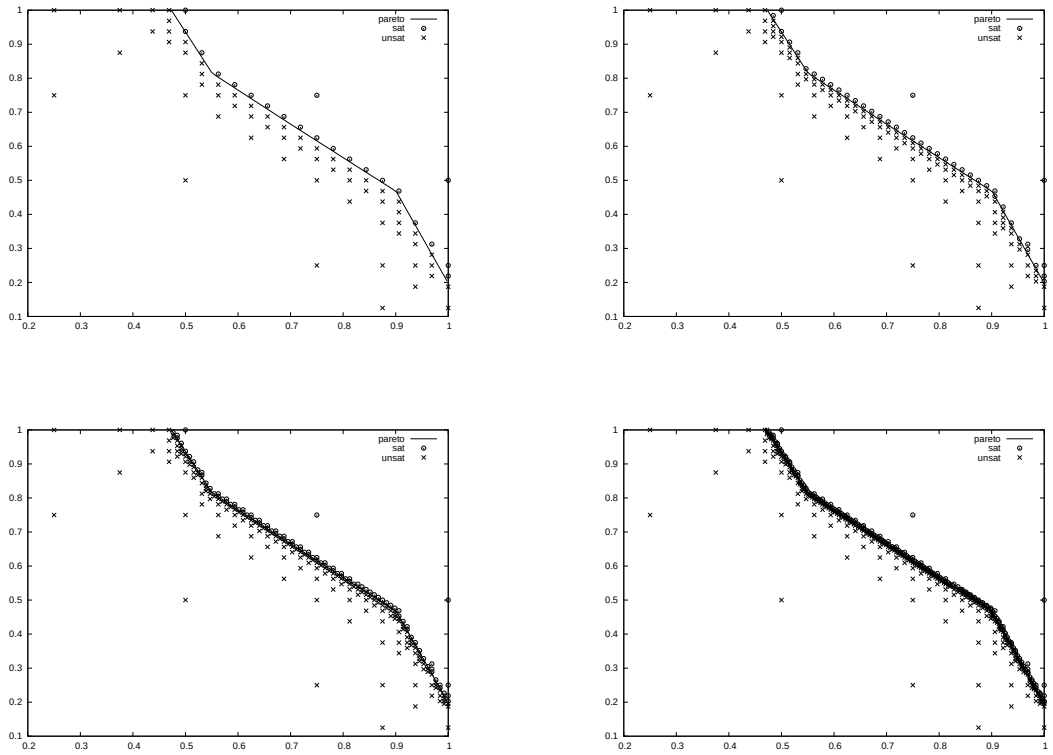


Figure 2.5.1: The results of our algorithm for the same front for $\epsilon = 0.05, 0.125, 0.001, 0.0005$.

We also combined our algorithm with the SMT solver Z3 [DMB08], and have encoded the following problem which triggered this research: given an application expressed as

d	no tests	ϵ	$(1/\epsilon)^d$	min no queries	avg no queries	max no queries
2	40	0.050	400	5	11	27
		0.025	1600	6	36	111
		0.001	1000000	21	788	2494
3	40	0.050	8000	5	124	607
		0.025	64000	6	813	3811
	20	0.002	125000000	9	30554	208078
4	40	0.050	160000	5	1091	5970
		0.025	2560000	10	11560	46906

Table 2.1: The average number of queries for surfaces of various dimensions and values of ϵ.

a task-data graph (a partially-ordered set of tasks with task duration and inter-task communication volume) and a heterogenous multi-processor architecture (a set of processors with varying speeds and energy consumptions, and a communication topology), find a mapping of tasks to processors and a schedule so as to optimize some performance criteria. Chapter 5 which is dedicated to the study of this mapping/scheduling problem, reports the results we obtained using Algorithm 2.4.1 and Z3 SMT solver.

2.6 Extensions and Future Work

This section contains some possible future extensions to the work achieved so far. In the first part an extension of the algorithm to non-terminating calls is detailed. We introduce queries with a limited time budget and discuss how the algorithm may escape regions of the cost space where the call generally ends up in a timeout [4]. In the second part we propose an alternative to the binary search method, where instead of orienting the search in a specific part of the cost space we formulate an additional weak constraint asking for an improvement of the current approximation. The answer returned by the solver to this extended query is *unsat* if the approximation is already ϵ-close to the Pareto front. Otherwise, the query is satisfied and the provided solution brings a quantified improvement over the current approximation.

2.6.1 Timeouts

A simple method to handle non-terminating calls is to abort the query after a given amount of time (timeout) and to consider the answer as *unsat*, based on the fact that no solution was found within the time limit. This is a straightforward way to alleviate the problem but the quality of the approximation is no longer guaranteed. There is another approach to this problem. Given a time budget per query after which we renounce to the search, we can maintain a set $S_\perp$ of the points where the call ended by a timeout. The algorithm can further use this information to predict and avoid parts of the cost space where answering queries is too difficult to be done within the time limit.

4. When the oracle is a SAT solver, which may take years to answer, it is necessary to cope with non-terminating calls.

In order to predict where the solver is likely to fail, its behavior must be axiomatized in some way. In the sequel we suggest two reciprocal hypotheses for this purpose. Let s and s' be two points in the cost space such that $s' < s$ and, hence $B^-(s') \subset B^-(s)$. We assume that:

1. It takes more time to find a solution in $B^-(s')$ than in $B^-(s)$;
2. It takes more time to prove unsatisfiability in $B^-(s)$ than in $B^-(s')$.

Intuitively, the first property follows the intuition that it is harder to find a solution when the constraints are tighter and a solution in $B^-(s')$ is also a solution in $B^-(s)$. Likewise, it is easier to provide a proof of unsatisfiability for tighter queries. Despite the intuitive appeal of these assumption, we should keep in mind that they are not always correct empirically due to the complex working of SAT/SMT solvers. Such solvers make random choices and may by chance be faster than expected. Another reason for violating these assumptions is the technique known as learning which in a nutshell consists in adding clauses that summarize previous deductions. This practice may lead to quick *unsat* answers once useful information has accumulated. Despite these reservations, we stick in this section to these reasonable assumptions that should hold on the average.

The two hypotheses entail a particular organization of the cost space between *sat*, *unsat* and *timeout* parts. Let T be the (unknown) subspace of *all* timeout points, let $\underline{T}$ and $\overline{T}$ be its minimal and maximal elements, respectively. Then T satisfies the property

$$T = B^-(\overline{T}) \cap B^+(\underline{T}) \tag{2.6E}$$

This is due to the fact that each point in the intersection both dominates *and* is dominated by at least one point of T and hence cannot be neither *sat* nor *unsat*. Given a finite set $S_\perp \subset T$ of already-discovered timeout points, any point in the intersection $B^-(\overline{S}_\perp) \cap B^+(\underline{S}_\perp)$ is a timeout point for the same reasons. Figure 2.6.1 depicts a partition between the *sat*, *unsat* and *timeout* parts of the space, which is valid under the assumption. To reduce the number of timeouts we should not ask queries with costs in $B^-(\overline{S}_\perp) \cap B^+(\underline{S}_\perp)$.

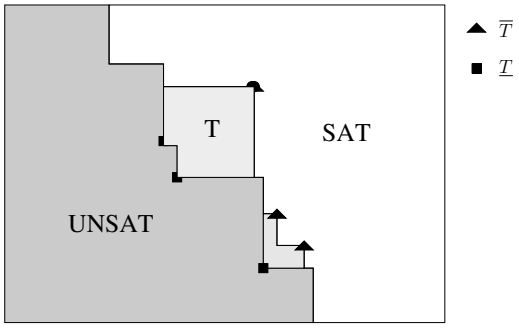
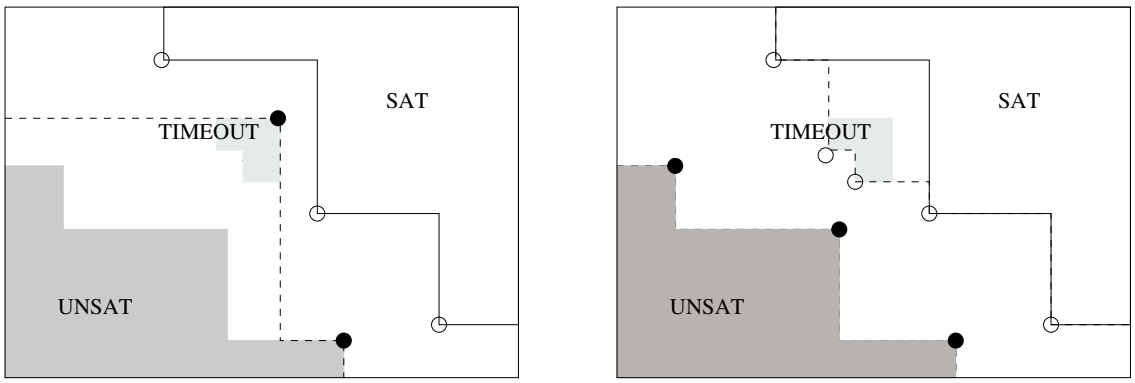


Figure 2.6.1: A partition between the *sat*, *unsat* and *timeout* parts of the cost space respecting property 2.6E.

Given this situation, two independent processes may be decoupled: the search for solutions and the search for a lower bound, the latter becoming an option in this case. In order to approximate the Pareto front, Algorithm 2.4.1 can be used to minimize the distance between $bd(B^-(S_0 \cup S_\perp))$ and S_1 (Figure 2.6.2 a). This is equivalent to considering that timeout points are *unsat*, just like we suggested at the beginning of this section. This

algorithm would produce an ϵ-approximation of the best approximation obtainable under a given time budget per query. Reciprocally, Algorithm 2.4.1 can be used to refine the lower bound by minimizing the distance between $B^-(S_0)$ and $S_1 \cup \underline{S_\perp}$ (Figure 2.6.2-b).



(a) (b)

Figure 2.6.2: (a) The dashed line represents the boundary of $B^-(S_0 \cup \overline{S_\perp})$. Under the assumptions we have, this represents the limit of improvement for the current approximation of the Pareto set. (b) The dashed line forms the boundary of $B^-(S_1 \cup \underline{S_\perp})$. This is the limit over which unsatisfiability cannot be proved anymore.

2.6.2 A Search by Successive Improvements

Another interesting variant of the querying methodology does not maintain the *unsat* information, but instead directly asks the solver whether the current approximation S is an ϵ-approximation of the Pareto set. This query that we denote as $\psi(S)$ is encoded using the following formula:

$$\psi(S): \ \nexists x \ (\varphi(x) \wedge \forall s \in S \ \rho(C(x), s) > \epsilon)$$

If we develop the distance ρ and negate $\psi(S)$ we obtain:

$$\neg\psi(S): \ \exists x \ \varphi(x) \wedge_{s \in S} \vee_{i=1}^{d} (C_i(x) + \epsilon < s_i)$$

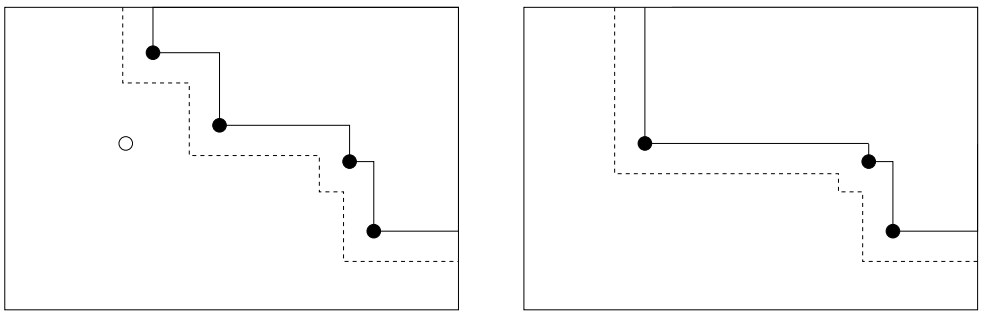
This formula can be submitted to an SMT solver. If $\neg\psi(S)$ is satisfied then the quality of S is improved by adding the point $C(x)$ whose distance to any point in S is larger than ϵ. If it is unsatisfied, S is proved to be an ϵ-approximation of the Pareto front. This leads to a new approximation algorithm (Algorithm 2.6.1).

Algorithm 2.6.1 Approximate Pareto Surface 2

$S = (1 \ldots 1)$
while $\neg\psi(S)$ **do** {call to SMT solver}
  $S = S \cup \{C(x)\}$ % add $C(x)$ to S where x is a model of $\psi(S)$
end while
return S % S is an ϵ-approximation of the Pareto front

The algorithm makes successive improvements of the current approximation until the formula is unsat, meaning that the distance to the Pareto front is less than ϵ (Figure 2.6.3).

Note that in this case there is a *unique* unsat call (the one showing the ϵ-optimality), and the rest of the time is spent on finding solutions rather than proving unsatisfiability. Because of this, this method could be an interesting alternative to the binary search approach.



(a) (b)

Figure 2.6.3: (a) The dashed line is the upper frontier of the part of the cost space where a solution is searched. This is the boundary of $B^+(S)$ translated by ϵ. (b) After a solution is found, S is updated and the search continues.

An additional advantage of this method is that each call is only weakly constrained, because we do not specify where in the cost space to look for an improving solution. Therefore the solver should quickly find solutions at the beginning of the search, and slow down its pace when approaching the Pareto front. On the other hand there might be a problem of convergence on problems admitting a dense space of solutions: with a small ϵ, the rate of improvement of the approximation could be really slow. In this case, the algorithm would benefit from an adaptive scheme where ϵ is decreased as the Pareto frontier gets closer. Or, we could maintain an under and over-approximation of the frontier and perform binary search by asking each query somehow in the middle between the two approximations. Note however that from a technical viewpoint this technique may be hard to implement efficiently because it requires asserting and de-asserting a lot of cost-related constraints in the solver.

2.7 Conclusions

We have presented a novel approach for approximating the Pareto front. The difficulty of the problem decomposes into two parts which can, at least to some extent, be decoupled. The first is related to the hardness of the underlying constraint satisfaction problem, which can be as easy as linear programming or as hard as combinatorial or nonlinear optimization. The second part is less domain specific: approximate the boundary between two mutually-exclusive subsets of the cost space which are not known a priori, based on adaptive sampling of these sets, using the constraint solver as an oracle. We have proposed an algorithm, based on a careful study of the geometry of the cost space, which unlike some other approaches, provides *objective* guarantees for the quality of the solutions in terms of a bound on the approximation error. Our algorithm has been shown to behave well on numerous examples. The knee tree data structure represents effectively the state of the algorithm and reduces significantly the number of distance calculations per query. We speculate that this structure and further geometrical insights can be useful as well to other approaches for solving this problem. We have investigated additional efficiency enhancing

tricks, most notably, *lazy* updates of the knee tree: if it can be deduced that a knee g does not maximize the distance $\rho(S_0, S_1)$, then the distance $\rho(g, S_1)$ need not be updated in every step.

Finally we have presented extensions of the current work which have not been implemented. Most urgently, the scalability of the method would be improved if we implement a scheme to cope with calls that practically do not terminate. In the future we could also investigate specializations and adaptations of our general methodology to different classes of problems. For example, in convex linear problems the Pareto front resides on the surface of the feasible set and its approximation may benefit from convexity and admit some symbolic representation via inequalities. To conclude, we believe that the enormous progress made during the last decade in SAT and SMT solvers will have a strong impact on the optimization domain [NO06] and we hope that this work can be seen as a step in this direction.

Chapter 3

Universal Restarting Strategies in Stochastic Local Search

Résumé : *Ce chapitre présente un algorithme de recherche stochastique locale permettant de résoudre des problèmes d'optimisation multi-critère. L'algorithme est basé sur la méthode des sommes pondérées. Cette méthode consiste à définir comme nouvel objectif une somme pondérée des fonctions objectif du problème multi-critère. En faisant varier les poids de cette somme et en résolvant à chaque fois le problème d'optimisation uni-dimensionnel associé, on obtient un ensemble de solutions non-dominées approximant le front de Pareto. Chaque vecteur de poids correspond en fait à une direction de recherche particulière dans l'espace des objectifs. Toutefois, il n'est pas évident de choisir ces vecteurs de manière à générer un ensemble de solutions bien diversifié (c'est à dire bien réparti dans l'espace des coûts). Nous proposons une stratégie dont le but est de multiplexer de manière équilibrée la recherche dans différentes directions, partant du principe qu'aucune de ces directions ne doit être privilégiée a priori (c'est à dire que l'on ne dispose pas d'informations particulières sur le problème traité). Cette stratégie est inspirée de résultats concernant le redémarage optimal d'un algorithme stochastique. Le redémarage est une technique simple qui permet d'éviter à l'algorithme de recherche locale d'être piégé autour d'optima locaux. Il existe une stratégie de redémarage présentant des propriétés théoriques intéressantes qui s'applique à l'optimisation classique. Nous avons combiné cette stratégie avec la méthode des sommes pondérées afin d'obtenir un algorithme de résolution de problèmes multi-critère. L'algorithme a été testé sur le problème de placement quadratique, ce qui a permis de valider l'efficacité de l'approche proposée.*

3.1 Introduction

In this chapter we present an adaptation of *stochastic local search* (SLS) algorithms to the multi-objective setting. SLS algorithms perform a guided probabilistic exploration of the decision space where at each time instance, a successor is selected among the neighbors of a given point, with higher probability for locally-optimal points. They are used extensively for solving hard combinatorial optimization problems for which exact methods do not scale [HS04]. Among the well known techniques we find *simulated*

annealing [KGV83], *tabu search* [GM06] and *ant colony optimization* [AC91], each of which has been applied to hard combinatorial optimization problems such as the traveling salesman, scheduling or assignment problems. Also related to this work are population-based *genetic algorithms* [Mit98, Deb01] which naturally handle several objectives as they work on several individual solutions that are mutated and recombined. They are vastly used due to their wide applicability and good performance and at the same time they also benefit from combination with specialized local search algorithms, leading to the class of so called *memetic algorithms* [KC05].

Many state-of-the-art local search algorithms have a multi-objective version [PS06a]. For instance, there exists extensions of simulated annealing [CJ98, BSMD08] and tabu search [GMF97] to multiple objectives. As mentioned in Section 1.4.1.1, a popular approach for handling multi-objective problems is based on *scalarization*: optimizing a one-dimensional cost function defined as a *weighted sum* of the individual costs according to a weight vector λ. Repeating the SLS process with different values of λ may lead to a good approximation of the Pareto front. A possible option is to pick a representative set of weight vectors a priori and run the algorithm for each scalarization successively. This has been done in [UTFT99] using simulated annealing as a backbone. More sophisticated methods have also emerged where the runs are made in parallel and the weight vector is adjusted during the search [CJ98, Han97] in order to improve the diversity of the population. Weight vectors are thus modified such that the different runs are guided towards distinct unexplored parts of the cost space.

One of the main issues in using the weighted sum method is to appropriately *share* the exploration time between different promising search directions induced by weight vectors. Generally deterministic strategies are used: weight vectors are predetermined, and they may be modified dynamically according to a deterministic heuristic. However, as pointed out in Chapter 1.1.2, unless some knowledge about the cost space has been previously acquired, one may only speculate about the directions which are worth exploring. For this reason we study in this work a *fully stochastic* approach, based on the assumption that all search directions are equally likely to improve the final result. The goal we seek is therefore to come up with a scheme ensuring a *fair* time sharing between different weight vectors.

To this end we borrow some ideas developed in the context of *restarts* for single-criteria SLS. Restarting means abandoning the current location in the search space and starting again from scratch. It is particularly efficient in avoiding stagnation of the algorithm and escaping a region with local optima. Another aspect in which restarts may help is to limit the influence of the random number generator. Random choices partly determine the quality of the output and starting from scratch is a *fast* way to cancel bad random decisions that were made earlier.

Most local search methods already feature a mechanism to combat stagnation, but it can be insufficient. For example a tabu search process is sometimes trapped inside a long cycle that it fails to detect. For this reason, restarts have been used extensively in stochastic local search for combating problems associated with the cost landscape, and they usually bring a significant improvement in the results. For instance greedy randomized adaptive search procedures (GRASP) [FR95] or iterated local search [LMS03] are well-known methods which run successive local search processes starting from different initial solutions. In [HS04, Ch. 4] restarts in the context of single-criteria SLS are studied based on an empirical evaluation of run-time distributions for some classes of problems.

It has been observed (and proved [LSZ93]) that in the absence of a priori knowledge about the cost landscape, scheduling such restarts according to a specific pattern boosts the performance of such algorithms in terms of expected time to find a solution. This has led to efficient algorithms for several problems, including SAT [PD07]. The major contribution of this chapter is an algorithmic scheme for distributing the multi-objective optimization effort among different values of λ according to the pattern suggested in [LSZ93]. The algorithm thus obtained is very efficient in practice.

The rest of the chapter is organized as follows. In Section 3.2, we introduce an abstract view of stochastic local search optimizers. Section 3.3 discusses the role of restarts in randomized optimization. Section 3.4 presents our algorithm and proves its properties. Section 3.5 provides experimental results and Section 3.6 concludes.

3.2 Stochastic Local Search

In the sequel we provide a general view of a stochastic local search algorithm which we use for the rest of this chapter.

Definition 3.2.1 (Neighborhood) *Let $\rho : \mathcal{X}^2 \to \mathbb{N}$ be a distance between vectors of the decision space. The intended meaning of $\rho(x, x')$ is the minimal number of modifications needed to transform x to x' (and symmetrically x' to x). The associated neighborhood on $\mathcal{X}$ is then the symmetric binary relation defined as*

$$\mathcal{N}(x, x') \textit{ iff } \rho(x, x') = 1$$

The set $\mathcal{V}(x) = \{x' \text{ s.t. } \mathcal{N}(x, x')\}$ is called the neighborhood of x. The neighborhood relation $\mathcal{N}$ can be viewed as a non-directed graph structure on $\mathcal{X}$ where two points x, x' are connected when $\mathcal{N}(x, x')$.

An SLS algorithm is a process which explores the decision space $\mathcal{X}$ through a *run* over the neighborhood graph. The algorithm most of the time goes from one solution to a *better* one in its local neighborhood (i.e it follows a path in the graph), but sometimes restarts from another location. We represent a run as a word composed of the sequence of nodes visited by the algorithm.

Definition 3.2.2 (Run) *A run over the decision space $\mathcal{X}$ is a word $\omega \in \mathcal{X}^*$. We denote by $\underline{\omega}$ and $\overline{\omega}$, respectively, the first and last symbol in ω.*

An SLS algorithm generates a run by selecting at each step a a new node according to a specific probability distribution over $\mathcal{X}$. In the sequel we define an SLS strategy as a function which, given the current run, returns a probability distribution over the next state.

Definition 3.2.3 (SLS strategy) *An SLS strategy is a function $\mathcal{S} : \mathcal{X}^* \to (\mathcal{X} \to [0, 1])$ which associates to each run a next-state probability distribution. For each $\omega \in \mathcal{X}^*$ either one of the two following holds*

- $\forall x \notin \mathcal{V}(\overline{\omega})$, $S(\omega)(x) = 0$ *(local move)*
- $S(\omega) = S(\epsilon)$ *(restart)*

The first condition ensures that except for restarts, all steps of the process are local. The second condition characterizes restarts as drawing the next state from the *initial* probability distribution $S(\epsilon)$ rather than the next-state probability. Typically $S(\epsilon)$ may be a uniform distribution over $\mathcal{X}$, or a deterministic choice meaning that restarts are always triggered from pre-computed solution. In the following sections we study how to schedule such restarts during a run of an SLS algorithm.

3.3 Restarting Single Criteria SLS Optimizers

This practice of restarting in SLS optimization has been in use at least since [SKC93]. At the same time, there is a theory of restarts formulated in [LSZ93] which applies to *Las Vegas* algorithms which are defined for *decision problems* rather than optimization. Such algorithms are characterized by the fact that their run-time until giving a correct answer to the decision problem is a random variable. In the following we recall the results of [LSZ93] and analyze their applicability to optimization using SLS. We begin by defining properly what a strategy for restarting is.

Definition 3.3.1 (Restart Strategy) *A restart strategy S is an infinite sequence of positive integers $t_1, t_2, t_3, \ldots$. The t time prefix of a restart strategy S, denoted $S[t]$ is the maximal sequence $t_1, t_2, \ldots, t_k$ such that $\Sigma_{i=1}^{k} t_i \leq t$.*

Running a Las Vegas algorithm according to strategy S means running it for t_1 units of time, restarting it and running is for t_2 units of time and so on.

3.3.1 The Luby Strategy

A strategy is called *constantly repeating* if it takes the form $c, c, c, c, \ldots$ for some positive integer c. As shown in [LSZ93] every Las Vegas algorithm admits an optimal restart strategy which is constantly repeating (the case of infinite c is interpreted as no restarting). However, this fact is not of much use because typically one has no clue for finding the right c. For these reasons, [LSZ93] introduced the idea of *universal* strategies that "efficiently simulate" every constantly repeating strategy. To get the intuition for this notion of simulation and its efficiency consider first the periodic strategy $S = c, c', c, c', c, c', \ldots$ with $c < c'$. Since resets are considered independent, following this strategy for $m(c+c')$ steps amounts to spending mc time according to the constant strategy c and mc' time according to strategy c'.[1] Putting it the other way round, we can say that in order to achieve (expected) performance as good as running strategy c' for t time, it is sufficient to run S for time $t + c(t/c')$. The function $f(t) = t + c(t/c')$ is the *delay* associated with the simulation of c' by S.[2] It is natural to assume that a strategy which simulates numerous other strategies (i.e has sub-sequences that fit each of the simulated strategies) admits some positive delay.

1. In fact, since running an SLS process for c' is at least as good as running it for c time, running S for $m(c+c')$ is at least as good as running c for $m(c+c)$ time.

2. For simplicity here we consider the definition of the delay only at time instants $t = kc'$, $k \in \mathbb{N}$. In the case where $t = kc' + x$, where x is an integer such that $0 < x < c'$, the delay function would be $\delta(t) = \lfloor t/c' \rfloor c + t$.

Definition 3.3.2 (Delay Function) *A monotonic non-decreasing function $\delta : \mathbb{N} \to \mathbb{N}$, satisfying $\delta(x) \geq x$ is called a delay function. We say that S simulates S' with delay bounded by δ if running S' for t time is not better than running S for $\delta(t)$ time.*

It turns out that a fairly simple strategy, which has become known as the *Luby strategy*, is universally efficient.

Definition 3.3.3 (The Luby Strategy) *The Luby Strategy is the sequence*

$$c_1, c_2, c_3, \ldots$$

where

$$c_i \doteq \begin{cases} 2^{k-1} & \text{if } i = 2^k - 1 \\ t_{i-2^{k-1}+1} & \text{if } 2^{k-1} \leq i < 2^k - 1 \end{cases}$$

which gives

$$1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, \ldots \tag{3.3A}$$

We denote this strategy by $\mathcal{L}$.

A key property of this strategy is that it naturally multiplexes different constant strategies. For instance the sequence 3.3A contains eight ones, four twos, two fours and one eight, and the total time dedicated to each constantly repeating strategy 1, 2, 4 and 8 is the same. More formally, let A denote a Las Vegas algorithm and $A(c)$ denote the algorithm which runs A for c time units. Then, the sum of the execution times spent on $A(2i)$ is equal for all $1 \leq i < 2^{k-1}$ over a $2^k - 1$ length prefix of the series. As a result, every constant restart strategy where the constant is a power of 2 is given an equal time budget. This can also be phrased in terms of delay.

Proposition 3.3.1 (Delay of $\mathcal{L}$) *Strategy $\mathcal{L}$ simulates any power of two constant strategy c with delay $\delta(t) \leq t(\lfloor \log t \rfloor + 1)$.*

Proof 2 (Sketch) *Consider a constant strategy $c = 2^a$ and a time $t = kc$, $k \in \mathbb{N}$. At the moment where the k^{th} value of c appears in $\mathcal{L}$, the previous ones in the sequence are all of the form 2^i for some $i \in \{0..\lfloor \log t \rfloor\}$. This is because the series is built such that time is doubled before any new power of two is introduced. Furthermore after the execution of the k^{th} c, every 2^i constant with $i \leq a$ has been run for exactly t time, and every 2^i constant with $i > a$ (if it exists in the prefix) has been executed less than t time. This leads to $\delta(t) \leq t(\lfloor \log t \rfloor + 1)$.* ∎

This property implies that restarting according to $\mathcal{L}$ incurs a logarithmic delay over using the unknown optimal constant restart strategy of a particular problem instance. Additionally the strategy is optimal in the sense that it is not possible to have better than logarithmic delay if we seek to design a strategy simulating *all* constant restart strategies. The optimality of $\mathcal{L}$ is proved in [LSZ93], and we reformulate it here using the notion of delay and give a sketch of proof.

Proposition 3.3.2 ($\mathcal{L}$ Optimality) *Any time strategy which simulates every constant time strategy does it with delay $\delta(t) \geq t/2(\lfloor \log t \rfloor / 2 + 1)$.*

Proof 3 (Sketch) *let S be such a strategy and t a time. Consider the constant restart strategies $\{S_i = 2^i, 2^i \ldots\}_0^{\lfloor logt \rfloor}$. Each t-prefix of these strategies must be simulated by a $\delta(t)$-prefix of S. In particular the t-prefix of $S_{\lfloor logt \rfloor}$ which is $2^{\lfloor logt \rfloor}$ has to be simulated. So the $\delta(t)$-prefix of S needs an element bigger than or equal to $2^{\lfloor logt \rfloor}$. Also the t-prefix of $S_{\lfloor logt \rfloor - 1}$ which is $2^{\lfloor logt \rfloor - 1}, 2^{\lfloor logt \rfloor - 1}$ must be simulated. One of the two values can be mapped to the value greater than $2^{\lfloor logt \rfloor}$, but there should be another value greater than $2^{\lfloor logt \rfloor - 1}$. From the previous observations we have that $\delta(t) \geq 2^{\lfloor logt \rfloor} + 2^{\lfloor logt \rfloor - 1}$. If we denote N_i the minimum number of 2^i values that must be present in the $\delta(t)$-prefix of S we can formulate this as 1. $N_{\lfloor logt \rfloor} = 1$ and 2. $\forall i \leq \lfloor logt \rfloor - 1$, $N_i = 2^{\lfloor logt \rfloor - i} - \sum_{j=i+1}^{\lfloor logt \rfloor - 1} N_j - 1$ (i.e for each i values bigger than 2^i can be used to simulate a 2^i). By recursion we can show that $\forall i \leq \lfloor logt \rfloor - 1$, $N_i = 2^{\lfloor logt \rfloor - i - 1}$. We therefore get $\delta(t) \geq 2^{\lfloor logt \rfloor} + \sum_{i=0}^{\lfloor logt \rfloor - 1} 2^{\lfloor logt \rfloor - 1 - i}\ 2^i$. After simplifying the sum we obtain $\delta(t) \geq t/2(\lfloor \log t \rfloor / 2 + 1)$.*

The next section investigates how these fundamental results can be useful in the context of optimization using stochastic local search.

3.3.2 SLS optimizers

An SLS optimizer is an algorithm whose run-time distribution for finding a particular cost o is a random-variable.

Definition 3.3.4 (Expected Time) *Given an SLS process and a cost o, the random variable Θ_o indicates the time until the algorithm outputs a value at least as good as o.*

There are some informal reasons suggesting that using strategy $\mathcal{L}$ in the context of optimization would boost the performance. First a straightforward extension of results of 3.1 to SLS optimizers can be made.

Corollary 1 *Restarting an SLS process according to strategy $\mathcal{L}$ gives minimum expected run-time to reach any cost o for which Θ_o is unknown.*

The corollary follows directly because the program that runs the SLS process and stops when the cost is at least as good as o is a Las Vegas algorithm whose run-time distribution is Θ_o. In particular using strategy $\mathcal{L}$ in that context gives a *minimal expected time to reach the optimum* o^*. However, finding the optimum is too ambitious for many problems and in general we would like to run the process for a fixed time t and obtain the best approximation possible within that time. Unfortunately there is no reason for which minimizing the expected time to reach the optimum would also maximize the approximation quality at a particular time t.

On the contrary for each value of o we have more or less chances to find a cost at least as good as o within time t depending on the probability $P(\Theta_o \leq t)$.[3] Knowing the distributions one could decide which o is more likely to be found before t and invest more time for the associated optimal constant restart strategies. But without that knowledge every constant restart strategy might be useful for converging to a good approximation of o^*. Therefore whereas Las Vegas algorithms run different constant restart strategies

3. Note that these probabilities are nondecreasing with o since the best cost value encountered can only improve over time.

because it is impossible to know the optimal c^*, an SLS process runs them because it does not know which approximation o is reachable within a fixed time t, and every such o might be associated to a different optimal constant restart strategy. This remark further motivates the use of strategy $\mathcal{L}$ for restarting an SLS optimizer.

3.4 Multicriteria Strategies

In this section we extend strategy $\mathcal{L}$ to become a multi-criteria strategy, that is, a restart strategy which specifies *what* combination of criteria to optimize and for how long. We assume throughout a d-dimensional cost function $f = (f^1, \ldots, f^d)$ convertible into a one-dimensional function f_λ associated with a weight vector λ ranging over a bounded set Λ of a total volume V.

At this point it is also important to justify the choice of the weighted sum approach despite its major drawback mentioned in Chapter 1: the impossibility to reach Pareto points on concave parts of the front.[4] Actually, theorems 1A and 1B only state that some Pareto solutions admit no weight vector for which they are *optimal* in the scalar sense. This does not mean that these are unreachable if we optimize weighted sums of the objectives, because we may encounter them while making steps in the decision space. As our algorithm utilizes a Pareto filter and keeps track of the non dominated set of *all* the points it came across, it can theoretically find any Pareto point if random choices are allowed. Given the graph induced on $\mathcal{X}$ by the neighborhood relation, this amounts to saying that any node in the graph has a non zero probability of being visited. This fact was also confirmed experimentally (Section 3.5), as we found the whole Pareto front (including non-supported solutions) for small instances where it is known.

Definition 3.4.1 (Multicriteria Strategy) *A multi-criteria search strategy is an infinite sequence of pairs*

$$S = (t(1), \lambda(1)), (t(2), \lambda(2)), \ldots$$

where for every i, $t(i)$ is a positive integer and $\lambda(i) \in \Lambda$ is a weight vector.

The intended meaning of such a strategy is to run an SLS process to optimize $f_{\lambda(1)}$ for $t(1)$ steps, then $f_{\lambda(2)}$ for $t(2)$ and so on. Following such a strategy and maintaining the set of non-dominated solutions encountered along the way yields an approximation of the Pareto front of f.

Had Λ been a finite set, one could easily adapt the notion of simulation from the previous section and devise a strategy which simulates with a reasonable delay any constant strategy (λ, c) for any $\lambda \in \Lambda$. However since Λ is infinite we need a notion of *approximation*. Looking at two optimization processes, one for f_λ and one for $f_{\lambda'}$ where λ and λ' are close to each other, we observe that the functions may not be very different and the effort spent in optimizing f_λ is almost in the *same direction* as optimizing $f_{\lambda'}$. This motivates the following definition.

Definition 3.4.2 (ϵ-Approximation) *A strategy S ϵ-approximates a strategy S' if for every i, $t(i) = t'(i)$ and $|\lambda(i) - \lambda'(i)| < \epsilon$.*

4. By concave we mean a part which does not belong to the Pareto front's convex hull.

From now on we are interested in finding a strategy which simulates with good delay an ϵ-approximation of any constant strategy (λ, c). To build such a ϵ-*universal* strategy we construct an ε-net D_ε for Λ, that is, a minimal subset of Λ such that for every $\lambda \in \Lambda$ there is some $\mu \in D_\varepsilon$ satisfying $|\lambda - \mu| < \varepsilon$. In other words, D_ε consists of ε-representatives of all possible optimization directions. The cardinality of D_ε depends on the metric used and we take it to be [5] $m_\epsilon = V(1/\varepsilon)^d$. Given D_ε we can create a strategy which is a cross product of $\mathcal{L}$ with D_ε, essentially interleaving m_ϵ instances of $\mathcal{L}$. Clearly, every $\lambda \in \Lambda$ will have at least $1/m_\epsilon$ of the elements in the sequence populated with ε-close values.

Definition 3.4.3 (Strategy $\mathcal{L}_{D_\epsilon}$) *Let D be a finite subset of Λ admitting m elements. Strategy $\mathcal{L}_D = ((t(1), \lambda(1)), \ldots$ is defined for every i as*

1. $\lambda(i) = \lambda_{(i \mod m)}$
2. $t(i) = \mathcal{L}(\lceil \frac{i}{m} \rceil)$

Proposition 3.4.1 ($\mathcal{L}_{D_\epsilon}$ delay) *Let D_ϵ be an ε-net for Λ. Then $\mathcal{L}_{D_\epsilon}$ simulates an ϵ-approximation of any constant strategy (λ, c) with delay $\delta(t) \leq t m_\epsilon(\lfloor \log t \rfloor + 1)$.*

Proof 4 (Sketch) *For any constant (λ, c) there is an ϵ-close $\mu \in D_\epsilon$ which repeats every m_ϵ^{th} time in $\mathcal{L}_{D_\varepsilon}$. Hence the delay of $\mathcal{L}_{D_\epsilon}$ with respect to $\mathcal{L}$ is at most $m_\epsilon t$ and combined with the delay $t(\lfloor \log t \rfloor + 1)$ of $\mathcal{L}$ wrt any constant strategy we obtain the result.* ⌟

For a given ε, $\mathcal{L}_{D_\varepsilon}$ is optimal as the following result shows.

Proposition 3.4.2 ($\mathcal{L}_{D_\epsilon}$ Optimality) *Any strategy that ϵ-simulates every constant strategy has delay $\delta(t) \geq m_\epsilon t/2(\lfloor \log t \rfloor/2 + 1)$ with respect to each of those.*

Proof 5 *Consider such a multicriteria strategy and t steps spent in that strategy. Let $S_{i,j}$ denote the multicriteria constant strategy $(\lambda_i, 2^j), (\lambda_i, 2^j) \ldots$ for all $\lambda_i \in D_\epsilon$ and $j \in \{0..\lfloor \log t \rfloor\}$. The minimum delay when simulating all $S_{i,j}$ for a fixed i is $t/2(\lfloor \log t \rfloor/2 + 1)$ (Proposition 3.3.2). Because any two $\lambda_i, \lambda_i' \in D_\epsilon$ do not approximate each other, the delays for simulating constant strategies associated with different directions just accumulate. Hence $\delta(t) \geq m_\epsilon t/2(\lfloor \log t \rfloor/2 + 1)$.*

Despite these results, the algorithm has several drawbacks. First computing and storing elements of an ε-net in high dimension is not straightforward. Secondly, multi-dimensional functions of different cost landscapes may require different values of ε in order to explore their Pareto fronts effectively and such an ε cannot be known in advance. In contrast, strategy $\mathcal{L}_D$ needs a different D_ε for each ε with D_ε growing as ε decreases. In fact, the only strategy that can be universal for every ε is a strategy where D is the set of all rational elements of Λ. While such a strategy can be written, its delay is, of course, unbounded.

For this reason, we propose a *stochastic* restart strategy which, for any ϵ, ϵ-simulates all constant multi-criteria strategies with a good expected delay. Our stochastic strategy $\mathcal{L}^r$ is based on the fixed sequence of durations $\mathcal{L}$ and on random sequences of uniformly-drawn elements of Λ.

Definition 3.4.4 (Strategy $\mathcal{L}^r$)

5. Using other metrics the cardinality may be related to lower powers of $1/\varepsilon$ but the growth is at least linear.

– *A stochastic multi-criteria strategy is a probability distribution over multi-criteria strategies;*
– *Stochastic strategy $\mathcal{L}^r$ generates strategies of the form*

$$(t(1), \lambda(1)), (t(2), \lambda(2)), \ldots$$

where $t(1), t(2), \ldots$ is the sequence $\mathcal{L}$ and each $\lambda(i)$ is drawn uniformly from Λ.

Note that for any ϵ and λ the probability of an element in the sequence to be ϵ-close to λ is $1/m_\epsilon$. Let us give the intuition why for any $\epsilon > 0$ and constant strategy (λ, c), $\mathcal{L}^r$ probabilistically behaves as $\mathcal{L}_{D_\epsilon}$ in the limit. Because each $\lambda(i)$ is drawn uniformly, for any $\epsilon > 0$ the expected number of times $\lambda(i)$ is ϵ-close to λ is the same for any $\lambda \in \Lambda$. So the time is equally shared for ϵ-simulating different directions. Moreover the same time is spent on each constant c as we make use of the time sequence $\mathcal{L}$. Consequently $\mathcal{L}^r$ should ϵ-simulate fairly every strategy (λ, c). We did not compute the *expected* delay with which a given multicriteria strategy is simulated by $\mathcal{L}^r$.[6] Nonetheless a weaker reciprocal result directly follows: on a prefix of $\mathcal{L}^r$ of length $tm_\epsilon(\lfloor \log t \rfloor + \log m_\epsilon + 1)$, the expected amount of time ε-spent on any multi-criteria constant strategy (λ, c) is t.

Proposition 3.4.3 ($\mathcal{L}^r$ Expected Efficiency) *For all $\epsilon > 0$, after a time $T = tm_\epsilon(\lfloor \log t \rfloor + \log m_\epsilon + 1)$ in strategy $\mathcal{L}^r$, the random variable $w_{\lambda,c}(T)$ of the time spent on ϵ-simulating any constant strategy (λ, c) verifies $\mathbb{E}[w_{\lambda,c}(T)] = t$.*

Proof 6 *In time T, $\mathcal{L}^r$ executes tm_ϵ times any constant c (Proposition 1). On the other hand the expected fraction of that time spent for a particular λ is $1/m_\epsilon$.*

3.5 Experiments

In previous sections we provided a theoretical analysis of several restart strategies for multicriteria optimization. We have also tested their performance experimentally by embedding them inside a local search solver for the *quadratic assignment problem*. The results obtained are presented in this section.

3.5.1 Quadratic Assignment

The quadratic assignment problem (QAP) introduced in [KB57] is a hard unconstrained combinatorial optimization problem with many real-life applications related to the spatial layout of hospitals, factories or electrical circuits. The traveling salesman problem itself may be seen as a special case of this problem which is in fact one of the most challenging problem in combinatorial optimization.

An instance consists of n facilities whose mutual interaction is represented by an $n \times n$ matrix F with F_{ij} characterizing the quantity of material flow from facility i to j. In addition there are n locations with mutual distances represented by an $n \times n$ matrix D. A solution is a bijection from facilities to locations whose cost corresponds to the total amount of operational work, which for every pair of facilities is the product of their flow

6. It involves computing the expectation of the delay of $\mathcal{L}$ applied to a random variable with negative binomial distribution.

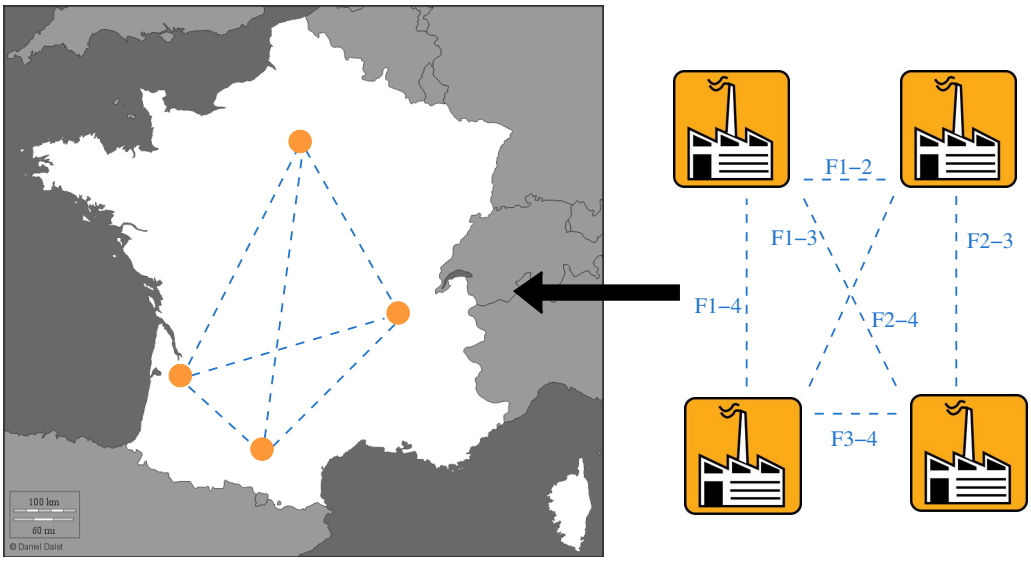


Figure 3.5.1: Illustration of a quadratic assignment problem: how to place a set of factories in cities of France. The cost of exchanging goods between two facilities is the product of the material flow with their distance.

by the distance between their respective locations. Viewing a solution as a permutation π on $[1..n]$ the cost is formalized as

$$C(\pi) = \sum_{i=1}^{n} \sum_{j=1}^{n} F_{ij} \cdot D_{\pi(i),\pi(j)}.$$

The problem is NP-complete, and even approximating the minimal cost within some constant factor is NP-hard [SG76].

3.5.1.1 QAP SLS Design

We implemented an SLS-based solver for the QAP, as well as multi-criteria versions described in the preceding section. The search space for the solver on a problem of size n is the set of permutations of $\{1, .., n\}$. We take the neighborhood of a solution π to be the set of solutions π' that can be obtained by swapping two elements. This kind of move is quite common when solving QAP problems using local search. Our implementation also uses a standard *incremental* algorithm [Tai91] for maintaining the costs of all neighbors of the current point, which we briefly recall here. Given an initial point, we compute (in cubic time) and store the cost of all its neighbors. After swapping two elements (i, j) of the permutation, we compute the effect of swapping i with j on all costs in the neighborhood. Since we have stored the previous costs, adding the effect of a given swap to the cost of another swap gives a new cost which is valid under the new permutation. This incremental operation can be performed in amortized constant time for each swap, and there are quadratically many possible swaps, resulting in a quadratic algorithm for finding an optimal neighbor.

In the sequel we give a more detailed description of the algorithm for computing the cost of solution π' based on the cost of π from which it has been obtained by a single swap.

Since this computation occurs in every step of the algorithm, its efficiency strongly affects the overall performance.

Consider a step $\pi \xrightarrow{i,j} \pi'$ which swaps facilities i and j in permutation π, yielding permutation π'. We assume having a matrix C^π indicating for each pair (m, n) the change in cost induced by the swap of facilities m and n in π. Therefore the cost C' of π' after the step is computed in constant time from the cost C of π as $C' = C + C^\pi_{i,j}$.

The tricky part is to derive the new matrix $C^{\pi'}$ from C^π. Actually we need to compute values of the form $C^{\pi'}_{g,h}$ which is the effect of swapping g with h on π', assuming we know $C^\pi_{g,h}$, the effect of swapping g with h on π. Note that we need only these values when $i < j$ and $g < h$. We first consider the case that $g \neq i, g \neq j, h \neq i, h \neq j$. In this event, $C'_{g,h} = C_{g,h} - O_{ghij} + N_{ghij}$ where

$$
\begin{aligned}
O_{ghij} = \quad & F_{gi}(D_{\pi(h),\pi(i)} - D_{\pi(g),\pi(i)}) \\
+ \quad & F_{gj}(D_{\pi(h),\pi(j)} - D_{\pi(g),\pi(j)}) \\
+ \quad & F_{hi}(D_{\pi(g),\pi(i)} - D_{\pi(h),\pi(i)}) \\
+ \quad & F_{hj}(D_{\pi(g),\pi(i)} - D_{\pi(h),\pi(j)}) \\
+ \quad & F_{ig}(D_{\pi(i),\pi(h)} - D_{\pi(i),\pi(g)}) \\
+ \quad & F_{jg}(D_{\pi(j),\pi(h)} - D_{\pi(j),\pi(g)}) \\
+ \quad & F_{ih}(D_{\pi(i),\pi(g)} - D_{\pi(i),\pi(h)}) \\
+ \quad & F_{jh}(D_{\pi(j),\pi(g)} - D_{\pi(j),\pi(h)})
\end{aligned}
$$

and N_{ghij} is defined likewise but under the permutation π' instead of π. The expression O_{ghij} accounts for the cost of swapping g with h which comes from the cells of the distance matrix which have some index amongst $\{g, h, i, j\}$ before the swap. Likewise, the expression N_{ghij} accounts for the cost of swapping g with h which comes from these cells after the swap. There are only $O(n)$ other neighbor swaps, *i.e* when either of g, h is equal to i or j. The costs of each of these other swaps can be recomputed from scratch in $O(n)$ time. At the end the global complexity of making a step is quadratic.

Concerning the search method, we use a simple greedy selection randomized by noise (Algorithm 3.5.1). This algorithm is very simple but, as shown in the sequel, is quite competitive in practice. The selection mechanism works as follows: with probability $1 - p$ the algorithm makes an optimal step (ties between equivalent neighbors are broken at random), otherwise it goes to a randomly selected neighbor. Probability p can be set to adjust the *noise* during the search. Note that we have concentrated our effort on the comparison of different adaptive strategies for the weight vector rather than on the performance comparison of Algorithm 3.5.1 with the other classical local search methods (simulated annealing, tabu search).

Algorithm 3.5.1 Greedy Randomized Local Search

```
if rnd() ≤ p then
  rnd_swap()
else
  optimal_swap()
end if
```

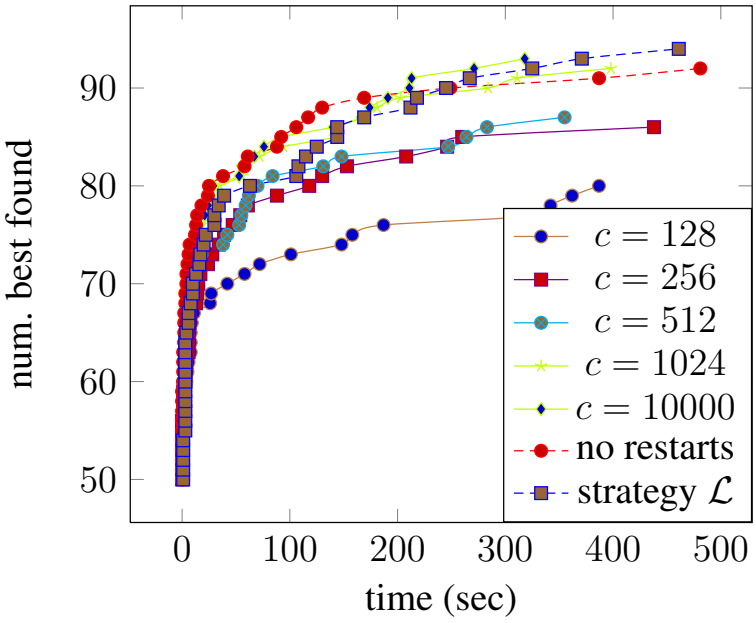


Figure 3.5.2: Results of different restart strategies on the 134 QAPLIB problems with optimal or best known results available. Strategy $\mathcal{L}$ solves more instances than the constant restart strategies, supporting the idea that it is efficient in the absence of inside knowledge.

3.5.1.2 Experimental Results on QAP library

The QAP library [BKR91] is a standard set of benchmarks for one dimensional quadratic assignment problems. Each problem in the set comes with an optimal or best known value. We ran Algorithm 3.5.1 (implemented in C) using various constant restart strategies as well as strategy $\mathcal{L}$ on each of the 134 instances of the library. [7] Within a time bound of 500 seconds per instance the algorithm finds the best known value for 93 out of 134 instances. On the remaining problems the average deviation from the optimum was 0.8%. Also the convergence is fast on small instances ($n \leq 20$) as most of them are brought to optimality in less than 1 second of computation. Complete results of the simulations are reported in Table 3.1.

Figure 3.5.2 plots for each strategy the number of problems brought to optimal or best-known values as a function of time. For a time budget of 500 seconds, strategy $\mathcal{L}$ was better than any constant strategy we tried. This fact corroborates the idea that strategy $\mathcal{L}$ is *universal*, in the sense that multiplexing several constant restart values makes it possible to solve (slightly) more instances. This is however done with some *delay* as observed in Figure 3.5.2 where strategy $\mathcal{L}$ is outperformed by big constant restarts (1024,10000,∞) when the total time budget is small.

3.5.2 Multi-objective QAP

The multi-objective QAP (mQAP), introduced in [KC03] admits multiple flow matrices $F_1, \ldots, F_d$. Each pair (F_i, D) defines a cost function as presented in Section 3.5.1, what renders the problem multi-objective.

7. The machine used for the experiments has an Intel Xeon 3.2GHz processor.

name	size	best value	mqap best	steps	time	name	size	best value	mqap best	steps	time
bur26a	26	5426670	5426670	87333	10.16	bur26b	26	3817852	3817852	25874	3.00
bur26c	26	5426795	5426795	52541	6.10	bur26d	26	3821225	3821225	4146	0.49
bur26e	26	5386879	5386879	26507	3.12	bur26f	26	3782044	3782044	9485	1.14
bur26g	26	10117172	10117172	26016	3.03	bur26h	26	7098658	7098658	9363	1.09
chr12a	12	9552	9552	10134	0.23	chr12b	12	9742	9742	1782	0.04
chr12c	12	11156	11156	19000	0.43	chr15a	15	9896	9896	5924	0.22
chr15b	15	7990	7990	91494	3.36	chr15c	15	9504	9504	88003	3.23
chr18a	18	11098	11098	196208	10.53	chr18b	18	1534	1534	5654	0.31
chr20a	20	2192	2192	3650116	245.31	chr20b	20	2298	2298	2154279	144.99
chr20c	20	14142	14142	205534	13.76	chr22a	22	6156	6156	2624486	212.39
chr22b	22	6194	6272	6182333	500.00	chr25a	25	3796	3822	4671871	500.00
esc128	128	64	64	937	3.44	esc16a	16	68	68	122	0.00
esc16b	16	292	292	40	0.00	esc16c	16	160	160	154	0.00
esc16d	16	16	16	43	0.00	esc16e	16	28	28	112	0.00
esc16f	16	0	0	1	0.00	esc16g	16	26	26	47	0.00
esc16h	16	996	996	42	0.00	esc16i	16	14	14	16	0.00
esc16j	16	8	8	16	0.00	esc32a	32	130	130	169182	30.31
esc32b	32	168	168	5063	0.95	esc32c	32	642	642	242	0.04
esc32d	32	200	200	1144	0.20	esc32e	32	2	2	11	0.00
esc32f	32	2	2	11	0.00	esc32g	32	6	6	14	0.00
esc32h	32	438	438	894	0.17	esc64a	64	116	116	110	0.10
had12	12	1652	1652	383	0.00	had14	14	2724	2724	1065	0.04
had16	16	3720	3720	702	0.03	had18	18	5358	5358	2975	0.16
had20	20	6922	6922	1477	0.10	kra30a	30	88900	88900	685946	106.45
kra30b	30	91420	91420	2981280	461.82	kra32	32	88900	26604	1	0.00
lipa20a	20	3683	3683	7321	0.50	lipa20b	20	27076	27076	1162	0.08
lipa30a	30	13178	13178	144126	22.49	lipa30b	30	151426	151426	11909	1.91
lipa40a	40	31538	31538	376838	108.15	lipa40b	40	476581	476581	15751	4.56
lipa50a	50	62093	62684	1087569	500.00	lipa50b	50	1210244	1210244	44406	20.99
lipa60a	60	107218	108152	738889	500.00	lipa60b	60	2520135	2520135	58474	39.91
lipa70a	70	169755	171057	524491	500.00	lipa70b	70	4603200	4603200	232953	218.46
lipa80a	80	253195	254942	398776	500.00	lipa80b	80	7763962	7763962	264266	325.77
lipa90a	90	360630	362964	314604	500.00	lipa90b	90	12490441	12490441	89043	144.09
nug12	12	578	578	126	0.00	nug14	14	1014	1014	14557	0.46
nug15	15	1150	1150	2120	0.08	nug16a	16	1610	1610	30191	1.29
nug16b	16	1240	1240	1984	0.08	nug17	17	1732	1732	16209	0.76
nug18	18	1930	1930	118373	6.39	nug20	20	2570	2570	8685	0.58
nug21	21	2438	2438	47793	3.59	nug22	22	3596	3596	1696	0.14
nug24	24	3488	3488	16016	1.58	nug25	25	3744	3744	81772	8.84
nug27	27	5234	5234	275874	34.38	nug28	28	5166	5166	26105	3.61
nug30	30	6124	6124	1093652	169.83	rou12	12	235528	235528	3259	0.07
rou15	15	354210	354210	4021	0.15	rou20	20	725522	725522	264911	17.90
scr12	12	31410	31410	2145	0.04	scr15	15	51140	51140	3511	0.13
scr20	20	110030	110030	14984	1.02	sko100a	100	152002	152696	250279	500.00
sko100b	100	153890	154890	253747	500.00	sko100c	100	147862	148614	253422	500.00
sko100d	100	149576	150634	253601	500.02	sko100e	100	149150	150004	253401	500.01
sko100f	100	149036	150032	254968	500.00	sko42	42	15812	15852	1552963	500.00
sko49	49	23386	23474	1146362	500.00	sko56	56	34458	34602	856498	500.00
sko64	64	48498	48648	631733	500.00	sko72	72	66256	66504	500701	500.00
sko81	81	90998	91286	391585	500.00	sko90	90	115534	116192	309691	500.02
ste36a	36	9526	9586	2163008	500.00	ste36b	36	15852	15852	494591	115.02
ste36c	36	8239110	8249952	2195600	500.00	tai100a	100	21125314	21591086	255199	500.00
tai100b	100	1185996137	1193503855	254527	500.00	tai12a	12	224416	224416	45	0.00
tai12b	12	39464925	39464925	2496	0.06	tai150b	150	498896643	507195710	101380	500.00
tai15a	15	388214	388214	1136	0.04	tai15b	15	51765268	51765268	438	0.02
tai17a	17	491812	491812	76666	3.68	tai20a	20	703482	703482	960799	63.94
tai20b	20	122455319	122455319	8079	0.56	tai256c	256	44759294	44866220	28941	500.00
tai25a	25	1167256	1171944	4729773	500.00	tai25b	25	344355646	344355646	285069	30.29
tai30a	30	1818146	1828398	3236381	500.00	tai30b	30	637117113	637117113	1722202	267.30
tai35a	35	2422002	2462770	2334384	500.00	tai35b	35	283315445	283315445	571374	125.05
tai40a	40	3139370	3190650	1763043	500.00	tai40b	40	637250948	637250948	56899	16.74
tai50a	50	4941410	5067502	1083719	500.00	tai50b	50	458821517	458966955	1048395	500.00
tai60a	60	7208572	7392986	739306	500.00	tai60b	60	608215054	608758153	742214	500.00
tai64c	64	1855928	1855928	3636	3.02	tai80a	80	13557864	13846852	407800	500.00
tai80b	80	818415043	825511580	404012	500.00	tho150	150	8133398	8215216	101551	500.04
tho30	30	149936	149936	2396680	371.55	tho40	40	240516	241598	1729732	500.00
wil100	100	273038	273854	253337	500.00	wil50	50	48816	48862	1071145	500.00

Table 3.1: Results of running Algorithm 3.5.1 on the QAP library instances. Each row shows the best value obtained within a time limit of 500 seconds, compared to the best known or optimal value.

We have developed an extension of Algorithm 3.5.1 where multiple costs are agglomerated with weighted sums and the set of all non-dominated points encountered during the search is stored in an appropriate structure.

To implement strategy $\mathcal{L}^r$ one also needs to generate d-dimensional random weight vectors which are uniformly distributed. Actually, generating random weight vectors is equivalent to sampling uniformly random points on a unit simplex, which is in turn equivalent to sampling from a Dirichlet distribution where all parameters are equal to one. The procedure for generating random weight vectors is therefore the following: first generate d IID random samples $(a_1, \ldots, a_d)$ from a unit-exponential distribution, which is done by sampling a_i from (0,1] uniformly and returning $-log(a_i)$, and then normalize the vector thus obtained by dividing each coordinate by the sum $\sum_{i=0}^{d} a_i$.

Multi-criteria strategies $\mathcal{L}^r$ and $\mathcal{L}_{D_\epsilon}$ have been tested and compared on the benchmarks of the mQAP library, which includes 2-dimensional problems of size $n = 10$ and $n = 20$, and 3-dimensional problems of size $n = 30$ [KC03]. The library contains instances generated uniformly at random as well as instances generated according to flow and distance parameters which reflect the probability distributions found in real situations. Pre-computed Pareto sets are provided for all 10-facility instances. Note that our algorithm found over 99% of all Pareto points from all eight 10-facility problems within a *total* computation time of under 1 second.

For the remaining problems where the actual Pareto set is unknown, we resort to comparing performance against the non-dominated union of solutions found with any configuration. As a quality metric we use the epsilon indicator (see Definition 1.3.4) which is the largest *increase* in cost of a solution in the reference set, when compared to its best approximation in the set to evaluate. The errors are normalized with respect to the difference between the maximal and minimal costs in each dimension over all samples of restart strategies leading to a number $\alpha \in [0, 1]$ indicating the error with respect to the set of all found solutions.

Figures 3.5.3, 3.5.4 and 3.5.5 depict the results of the multi-criteria experiments. We compared $\mathcal{L}^r$ against constant restarts combined with randomly chosen directions and strategy $\mathcal{L}_{D_\varepsilon}$ for different values of ε. Despite its theoretical properties $\mathcal{L}_{D_\varepsilon}$ does not perform so good for the ε values that we chose. This corroborates the fact that it is hard to guess a good ϵ value beforehand. Constant restarting works well but the appropriate constant has to be carefully chosen. At the end strategy $\mathcal{L}^r$ gives decidedly *better performance amongst all the strategies we tried.* It is worth noting that we also tried using the weighted infinity norm $\max_i \lambda_i f^i$ as a measure of cost but, despite its theoretical superiority (any Pareto point on a concave part of the front is optimal for some λ), the method did perform worse than the weighted sum approach in our experiments.

3.6 Conclusion and Discussion

Through this work, we have demonstrated how efficient universal strategies can accelerate SLS-based optimization, at least in the absence of knowledge of good restart constants. In the multi-criteria case, our approximating universal strategy is efficient in solving quadratic assignment problems and gives a thorough and balanced coverage of the Pareto front. In chapter 5 we also apply this algorithm to solve mapping and scheduling problems. As a future work, it would be interesting to combine our multi-criteria strategy with tabu search or simulated annealing to confirm its effectiveness.

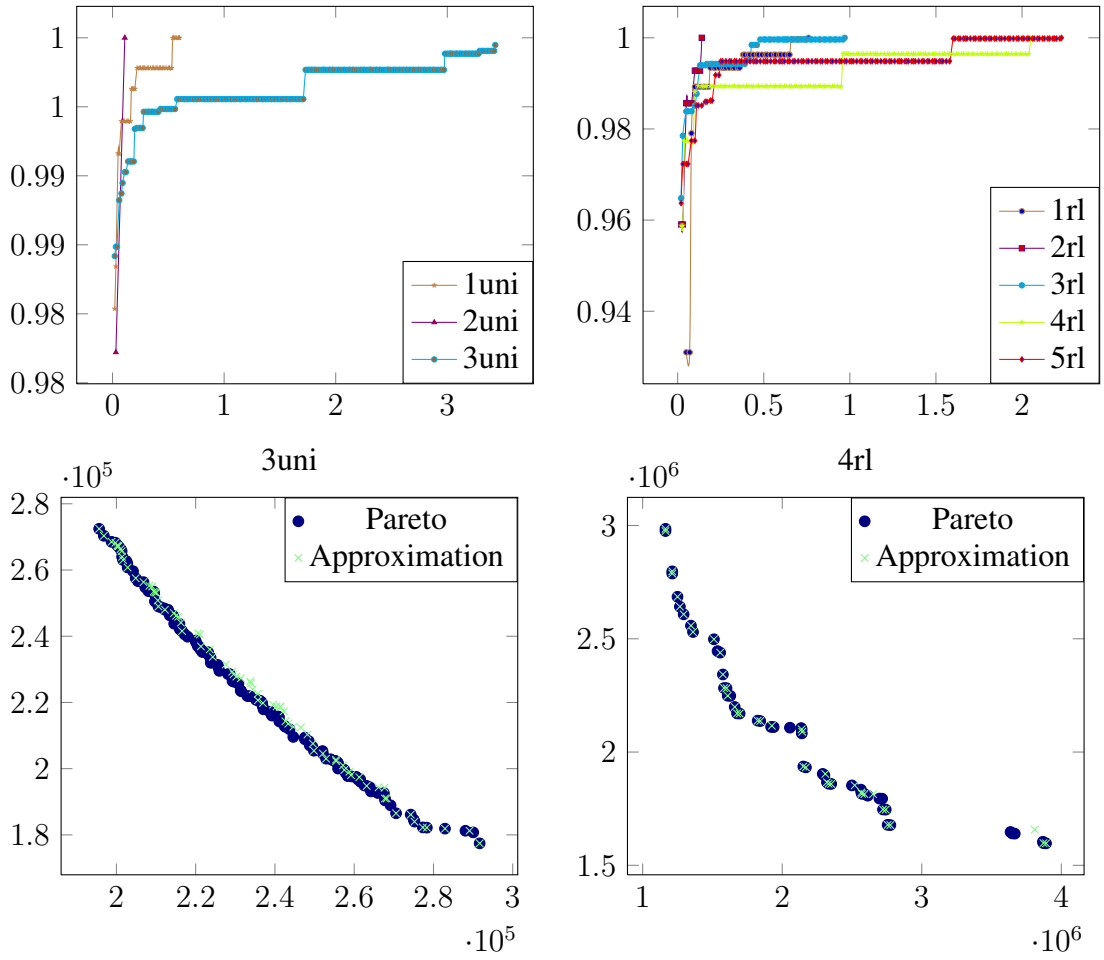


Figure 3.5.3: Overview of performance on several 10-facility instances. The top row shows plots of the evolution of quality $(1 - \alpha)$ as a function of time. The bottom plots give examples of the approximations found along the way compared to the actual Pareto set.

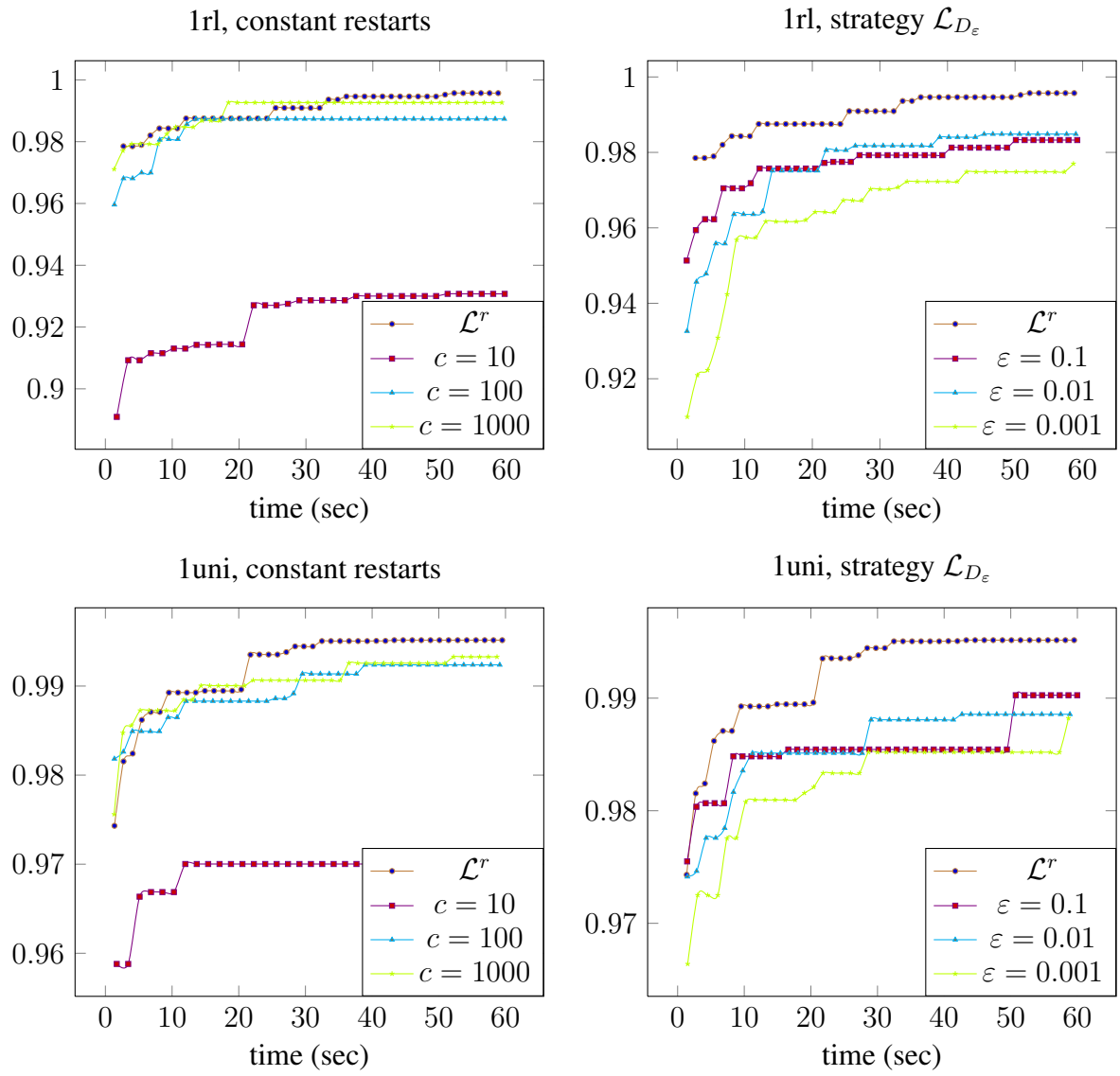


Figure 3.5.4: Performance of various multi-critieria strategies against $\mathcal{L}^r$ on the 1rl and 1uni 20-facility problems. Left : constant restarts, $c = 10, 100, 1000$ combined with randomly generated directions. Right : strategy $\mathcal{L}_{D_\varepsilon}$ for $\varepsilon = 0.1, 0.01, 0.001$

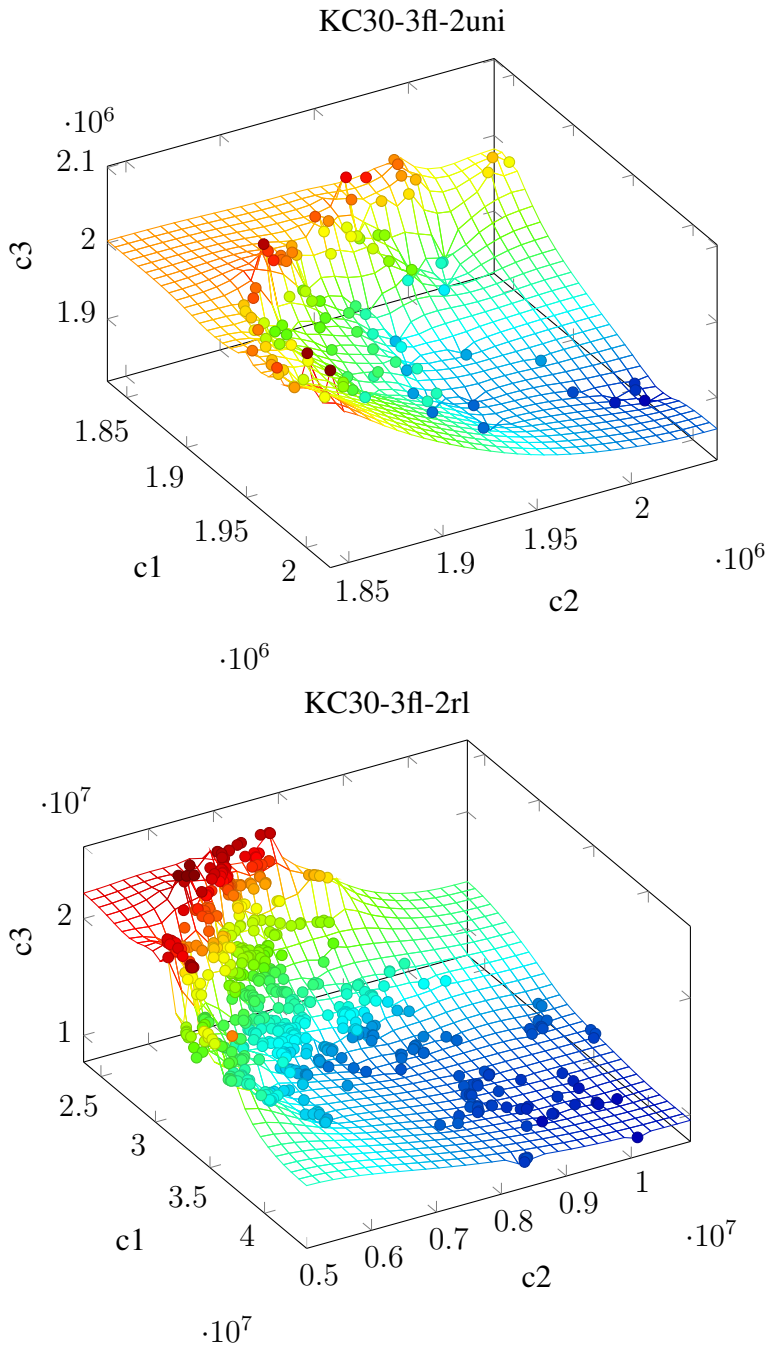


Figure 3.5.5: Examples of approximations of the Pareto front for two 30-facility, 3-flow QAP problems. KC30-3fl-2uni was generated uniformly at random while KC30-3fl-2rl was generated at random with distribution parameters, such as variance, set to reflect real instances. The approximations above are a result of 1 minute of computation using $\mathcal{L}^r$.

More generally, because stochastic local search algorithms may be really effective in solving combinatorial optimization problems, we believe that studying extensions of these to multiple criteria is an exciting topic of research. From our experience one major advantage of local search is that it is very fast: our program for solving quadratic assignment problems was indeed able to perform a large number of steps in a very short time (at least on medium-sized problems).

However, whether or not good performance can be achieved by local search is quite problem-dependent. The core activity of an SLS algorithm is to look for an improving solution in a local neighborhood. It is therefore crucial to design the algorithm such that this operation can be performed quickly. In the case of the QAP, a standard 2-exchange relation leads to an efficient incremental algorithm which performs a local optimization step in a quadratic number of operations. This is however less easy to do with constrained problems such as scheduling. Representing a solution, choosing a neighborhood relation and performing a step in an incremental manner indeed involves tougher algorithmic issues, as explained later on in Chapter 5.

More importantly, having multiple criteria makes it even more tricky to define an appropriate neighborhood relation, because decision variables may have contradictory effects on the different costs. For the purpose of illustration, let us consider a bi-criteria scheduling problem. The decision variables come into two types: mapping variables which encode task-to-processor assignments, and scheduling variables which define the start time (or priority) of each task. In the simple energy model we use, the power dissipated by a task depends on the processor on which it executes, but not on its start time. Therefore we have two types of variables that do not affect the two costs in the same way. If we consider changing the mapping and scheduling of a task in a single step, the neighborhood size could be too large to allow for efficiently moving to a locally optimal solution. Another option here would be to consider mapping and scheduling separately: either move a task from its processor or alter its start time. Now imagine that we want to perform design space exploration and add the possibility of changing the speed/voltage of each processor. We get yet another type of variable and the design of the neighborhood relation is even more complicated. We believe that this issue of combining variables which are differently related to the cost functions is really important to the performance of multi-criteria optimization using SLS, especially if we are to solve practical problems.

Chapter 4

Optimization for the Parallel Execution of Software

Résumé : *Ces dernières années on assiste à un changement stratégique de la part de l'industrie des semi-conducteurs, qui concentre désormais ses efforts à la réalisation d'architectures multi-processeur. Bien que la "loi" de Moore soit toujours d'actualité, la miniaturisation accrue ne permet plus d'augmenter la fréquence des processeurs d'avantage, car plusieurs barrières physiques ont été atteintes (consommation d'énergie, latence mémoire etc.). L'amélioration de la puissance passe donc par l'augmentation du nombre de coeurs de calcul, plutôt que par l'augmentation de la fréquence comme ce fut le cas auparavant. Cette ligne de développement est particulièrement indispensable pour les systèmes embarqués. En effet, les applications cibles sont très gourmande en performance (traitement vidéo ou applications de réalité augmentée par exemple), alors que la capacité des batteries n'augmente pas. Il faut donc plus de puissance de calcul mais un niveau de consommation énergétique raisonnable, ce que permettent les sytèmes multi-processeur. Une tendance vise également à remplacer les accelérateurs matériels par des processeurs multi-coeur, car ceux-ci sont en théorie capable d'offrir des performances comparables mais avec une meilleure flexibilité aux changements du marché (le logiciel est plus facilement modifiable). Toutefois, la performance des multi-processeur est conditionnée par l'efficacité des décisions de déploiement des applications parallèles, tels le placement de tâches et l'ordonnancement. Optimiser ces décisions pour différents critères (performance, énergie etc.) est compliqué. Il est donc intéressant de traiter ces problèmes à un haut niveau d'abstraction par des méthodes d'optimisation multi-critère, afin d'identifier un certain nombre de solutions prometteuses qui seront ensuite analysées à un plus bas niveau.*

4.1 The Comeback of Multi-Processors as a Standard

In the past few years, there has been a clear strategy shift of the chip industry which strongly renewed its interest in building parallel machines targeting various market segments, from desktop computers to servers in data centers or embedded systems. Basically designers were facing unbearable difficulties for keeping up with the pace of Moore's law, which predicts that the number of transistors integrated on a single chip doubles roughly

every two years [1]. Because of physical limits in frequency and power, the ability to build integrated circuits at deeper submicron levels does not easily translate anymore into a proportional speedup in software execution.

Up to now, additional logic was used to boost the performance of single processors, although this was achieved at the expense of a faster growth in power consumption and design complexity. Today it is common that modern processors integrate several pipelines and advanced mechanisms for handling *instruction-level* parallelism, making them able to issue multiple out-of-order instructions during a single cycle. However, this model of progress is seriously endangered by physical barriers, a.k.a *walls*, which prevent further improvement of traditional processors under affordable costs.

First, power consumption is a major concern. From an end-user perspective, we require always more computing power and battery lifetime for our daily-used embedded devices (cell phones, netbooks etc.). Likewise, companies running large data centers (Google for instance) realize that their power budget is responsible for a non negligible part of their costs, making them also interested in energy savings [FWB07]. More generally, there is a growing demand for energy-efficient systems offering high computing power and programmability.

However, the yield in performance obtained by increasing the frequency of a general purpose processor gets really poor beyond a certain point. This is because the required voltage and hence the energy consumption and heat are growing faster than the clock speed, a limitation commonly known as the *power wall*. A strong motivation for building multi-processor platforms comes from the effective performance to power ratio they theoretically provide. Running two processors at half the speed is less energy-consuming but *potentially* equally powerful. Consequently, given that power saving is becoming no less important than performance, augmenting the number of processing cores rather than their frequency is the most sustainable development strategy for chip manufacturers.

Beside power, there are numerous other limitations to the increase of processor frequency, starting with the well-known *memory wall*. Accessing off-chip memory entails an incompressible delay due to the physical limits on the latency and bandwidth of connections. If the speed of processors were to be raised further, it is expected that memory would shortly become a performance bottelneck [WM95].

Exploiting parallelism at the instruction level, a practice which has significantly boosted the speed of processors, also reaches a point of diminishing returns, hitting the *ILP wall* [ABC+06]. Whether it is done by the hardware or a compiler, the amount of instruction-level parallelism which can be automatically extracted from a single stream of instructions is quite limited. Given that latest-generation processors already exploit all of the instruction-level parallelism which can be detected in applications, the most promising lead for performance enhancement is to look at higher-level sources of parallelism. This fact led to the recent development of multi-core processors, which can manage thread and data level parallelism.

The complexity of modern processors also brings a batch of troubles. Sophisticated designs tend to be error-prone, hard to verify and less scalable. Consequently, it takes longer for engineers to design the circuit and the productivity is reduced, a fact known as the *designer productivity gap*. A complex processor also features more logic, and hence occupies a larger silicon area. A well-known rule from the industry states that, by experience, the performance of a single core is roughly proportional to the square root of

1. http://www.itrs.net

its logical complexity (measured by the number of transistors) [Bor07]. In other words, the return in performance on building faster processors is increasingly overwhelmed by area considerations. Fortunately, as it is with power, the increase in complexity may be overcome by multi-processor systems which achieve high-performance by multiplying less-powerful but simpler units.

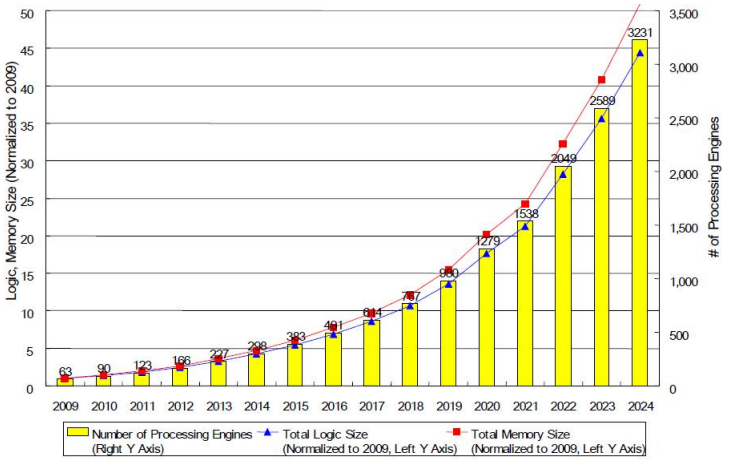


Figure 4.1.1: SoC Consumer Portable Design Complexity Trends (ITRS 2009 forecast).

For all these reasons, multi-core is becoming a de facto standard for the design of processors and embedded systems, and the number of parallel units on a single chip is expected to increase considerably over the next few years, as shown on Figure 4.1.1. Nonetheless, if the shift to multi-processor systems is a reasonable roadmap, it still represents a challenge for the software industry. Most existing code is sequential, and programmers will need innovative parallel programming solutions for the efficient development of software on future many-core processors.

The good news is that parallelism itself is not really a novel idea. It has a more than 25-years long history in the domain of scientific and high performance computing (HPC). The field is mainly focused on building supercomputers [Meu08] which provide huge processing power, and are able to attack computationally intensive scientific simulations such as weather forecast, nuclear reactions, or the evolution of galaxies in the universe. Although the knowledge and experience accumulated by the HPC community can certainly be useful, the software solutions they developed are not directly applicable to other domains like embedded systems, where goals and requirements are intrinsically divergent.

Moreover, considering that parallel programming is becoming the *mainstream* for general purpose processing, any programmer should be able to quickly develop and maintain parallel code. Unfortunately, writing programs is far more difficult in a parallel than in a sequential setting, as it involves many tricky concepts such as locks, barriers or data races. For parallel programming to become accessible to a larger public, future software solutions should abstract these low-level details as much as possible. This is all the more needed because developing handcrafted code is getting more and more tedious as the number of cores increases. Consequently, there is a need for high-level programming languages capable of capturing user-expressed parallelism, and associated compilers which can produce efficient low-level and machine-dependent code. This is a necessary step for sustaining software productivity in the long-term.

Although both research and industry already worked on parallel programming models in the past, no solution really emerged as a standard. Maybe the best explanation is that, as long as sequential code performance was boosted by increasing processor speeds, software developers did not see the necessity of parallelizing their applications. This results in the fact that, whereas the hardware industry is more or less ready to make the shift, the burden is pushed towards software developers who are responsible for bridging the gap between the applications and multi-processor systems [ABD+09].

4.2 Design Trends for Embedded Multi-Processors

In the following section we first review some of the major constraints, requirements and trends distinguishing the design of multi-processor systems-on-chip, and then present an ongoing project targeting the development of a many-core MPSoC. The project, carried out conjointly by a leading chip manufacturer (STMicroelectronics) and a research institute (CEA), did actually provide some motivations for the topics addressed in this thesis.

4.2.1 Overview

The embedded systems industry is following the same path as most devices are henceforth based on multi-processor architectures, what we call Multi-Processor Systems on Chip (MPSoC). Embedded systems are increasingly present in our daily lives through technological accessories such as mobile phones, PDAs or laptops. The demand for mobility is very high: for many of us the cell phone has completely replaced the fixed phone, the laptop the desktop computer. The industry has therefore good days ahead.

At the same time, consumers show a constant attraction to technology-advanced functionalities. A trendy smart phone has to provide WLAN connection, as well as to offer the possibility of playing 3D-games and to encode and decode video streams. Multimedia applications based on augmented reality [A+97] are also very trendy, and they require high performance (Figure 4.2.1). Now, given that embedded systems are operated on battery, their computing power must be provided under a very low energy cost. As a result, the use of massive parallelism is becoming mandatory in order to meet the conflicting goals of high-performance and low-power.

Unlike general purpose multi-processors which are optimized for peak performance, embedded systems are usually tailored to a specific class of applications (e.g multimedia, signal), and under stringent power and timing constraints. They are specialized to given requirements, and often combine both software and dedicated hardware: while the former brings flexibility and reconfigurability, the latter allows to speed up specific types of computations (e.g audio, video, bitstream operations) at a moderate energy cost. Therefore, embedded multi-processors have traditionnally been designed from a selected mix of *heterogeneous* processors, memories and interconnects, so as to trade-off between performance, power consumption, and programmability. Examples of commercial MPSoCs include the Philips Nexperia platform for digital video and television applications [DJR01], the STMicroelectronics Nomadik targeting mobile multimedia systems [Art05], or the STI CELL [KDH+05], which is the central processor of the PlayStation 3 games console.

Although heterogeneous HW/SW SoCs have achieved the best performance/flexibility trade-off, newly developed systems should be even more programmable. Actually, the potential of parallel architectures in offering cheap high-performance, clears a way for the

Figure 4.2.1: Augmented reality refers to the insertion of virtual elements into a real video scene. As an example, the figure shows a screenshot of a smartphone application by Nokia Beta Labs called LineView, which was recently announced (July 2011). This augmented reality browser, with the use of GPS and camera, displays real time information about the points of interest around your location.

execution of commonly hardwired functions in software. In the domain of multimedia applications, flexibility is of outmost importance because a single system often has to handle multiple audio/video standards: a music player, for instance, need to play MP3, Dolby Digital, Ogg Vorbis (etc.). Even though the decoding processes of the standards share many similarities, it would require an important effort to design a single hardware block handling every music format. Part of the decoding application is generally implemented in software, which is more flexible [Wol07]. This entails that if a standard is changed during the product's lifetime, the software part can be updated quickly, whereas an hardware component would have to undergo a new pass through the flow of design, verification and production. In the signal processing domain, the research efforts put towards developing embedded cognitive radio [Fet09] also witnesses the fact that, with the multiplication of standards, flexibility is a key requirement for the next-generation of multi-processor systems-on-chip.

In addition, the costs due to non-recurring engineering and mask sets are very expensive as the technology of integrated circuits progresses. According to experts, the cost of a 22nm mask will exceed one million dollars. This represents more than ten times the cost of a mask in the technology used ten years ago. To remain competitive, companies must offset these costs by an increase of production volumes. This can be achieved by designing a versatile chip to be used inside different systems. Platform based design [KNRSV00] is a paradigm which was proposed to develop this concept. It advocates the creation of a reconfigurable and programmable platform, for which applications are developed using a high-level API which abstracts the architectural details. Thus, the same design can be configured to meet the requirements of different applications, what increases the production volume of the chip. In addition, the time-to-market is improved, because an obsolete product is simply upgraded by mapping a new version of the application to a new configuration of the *same* architecture.

As the number of components integrated on a single chip keeps rising, it is also mandatory to develop new systems to manage communication. System-on-chip interconnects are

generally based on buses, point-to-point links, or a combination thereof, and these solutions do not scale regarding both performance and energy. Quite recently, the network-on-chip (NoC) paradigm [BDM02], in which a SoC is viewed as a micronetwork of components, has been broadly accepted as a promising alternative to alleviate this bottelneck. A NoC is composed of a set of links, routing nodes, and interfaces connecting the components to the network, on top of which is implemented a light protocol stack. Basically, messages are forwarded from source to destination like packets in a large-scale macronetwork.

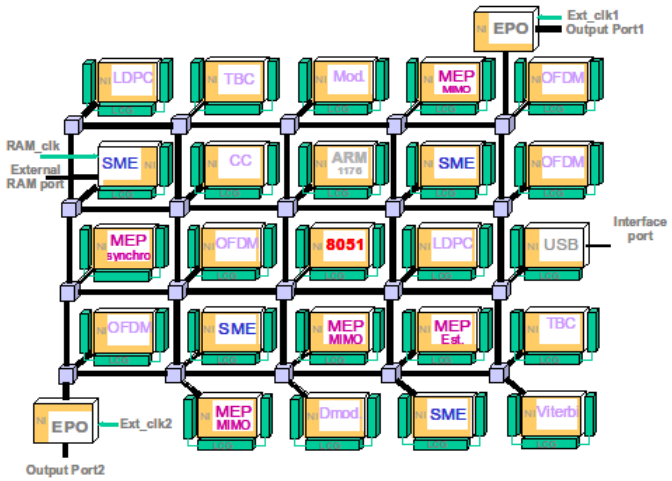


Figure 4.2.2: Example of a NoC-based multi-core system-on-chip: MAGALI, a digital baseband circuit for software-defined and cognitive-radio developed at CEA LETI [CBL+10]. Thanks to the asynchronous NoC and GALS interfaces, the system is partitioned into 23 frequency islands which can be programmed dynamically to ensure the best performance-versus-energy ratio.

The NoC paradigm addresses several of the recent MPSoC design issues, in particular better flexibility and scalability. One important feature is the decoupling of communication from computation which is provided by a high-level standardized communication protocol implemented in the network interface. Any existing component which conforms to that interface can be directly connected to the system. Consequently, components may be reused and the system is more flexible. Moreover, a NoC may be designed to be scalable if, like in a large scale distributed network, the building elements such as routers, links or protocols are independent of the number of nodes and communication links. This is a major improvement over bus-based systems in which the arbitration mechanisms rapidly saturate the traffic as the number of masters grows.

NoCs are also suitable for Globally Asynchronous Locally Synchronous (GALS) designs in which a set of autonomous processing units are working at their own speed, and exchange messages through an asynchronous communication network (Figure 4.2.2). This synchronization scheme is envisioned to be at the heart of future network-on-chips for two reasons. First, it is difficult to properly distribute a unique clock over a large multi-processor system. Secondly, autonomous components may be controled via DVFS (Dynamic Voltage and Frequency Scaling), in order to adapt the speed of each processing unit to the application demand. DVFS provides fine-grained power savings and temperature control, as well as means for coping with the variabilities in software and hardware [TBS+10].

Unlike distributed systems, networks-on-chip are not constrained by standardization. Their design may be customized to the traffic patterns generated by a specific type of applications (e.g multimedia) and to specific requirements regarding performance, power and predictability. In fact the interconnection network customization is taking more importance in the design than it previously had. Indeed, the rise of streaming data-intensive applications, together with the physical limits of wires in terms of power and delay, often make this element the critical part of the system. For these reasons, there is a trend to adopt a communication-centric design approach for the conception of MPSoCs. This is witnessed by the growing amount of research on application-specific NoC design methodologies [BJM+05, MOP+09].

4.2.2 The P2012 Platform

Today, an MPSoC is a highly heterogeneous system which embodies several hardwire accelerators for domain-specific computations (eg. video decoding, gaming, augmented reality). These dedicated subsytems achieve better power and area efficiency than general purpose processors. Nonetheless, as argued before, the great potential of multi-core processing is now providing a mean to replace these hardwire accelerators by programmable and customizable multi-core processors, in order to improve the system flexibility.

The P2012 platform developed conjointly by STMicroelectronics and CEA is such a programmable multi-core fabric targeted at domain-specific acceleration, which goal is to fill the area and power efficiency gap between general-purpose processors and fully hardwire accelerators. It is made of up to 32 clusters connected through a 2D-mesh asynchronous NoC (Figure 4.2.3). A cluster features a multi-core processing engine (Encore16) with up to 16 tightly-coupled processors STxP70-V4, and shared L1 memories for instructions and data. The STxP70-V4 is a dual issue 32-RISC processor running at 600 MHz, with two possible extensions, one for SIMD and one for bit manipulations needed for parsing video streams. The platform also integrate an array of specialized hardware processing elements (HWPE) in order to implement functions for which the software remains inefficient in terms of area and energy (eg. motion compensation in video decoding). Intra-cluster communications are handled by an asynchronous local interconnect.

The global system follows the GALS paradigm. Each cluster therefore has its own voltage and frequency domain, in order to finely control the power, reliability and variabilities of the platform. The clocking scheme is based on an innovative FLL (Frequency Locked Loop) which provide frequencies between 500MHz and 1.5GHz and can be reprogrammed quickly (within 200 ns). The clock generator is duplicated inside the cluster, and the Encore16 engine and the hardware PEs are located inside different power/frequency islands. DVFS may be performed at the cluster level in order to fine tune the frequency of each block. This is achieved through the VDD hopping technique, which consists in obtaining an average voltage/frequency point by alternating between a high and a low voltage, which are distributed over the fabric. The power management of each frequency island is controlled by a Clock Variability and Power module (CVP) which decides the configuration of each domain based on voltage, temperature, and timing faults information collected by sensors.

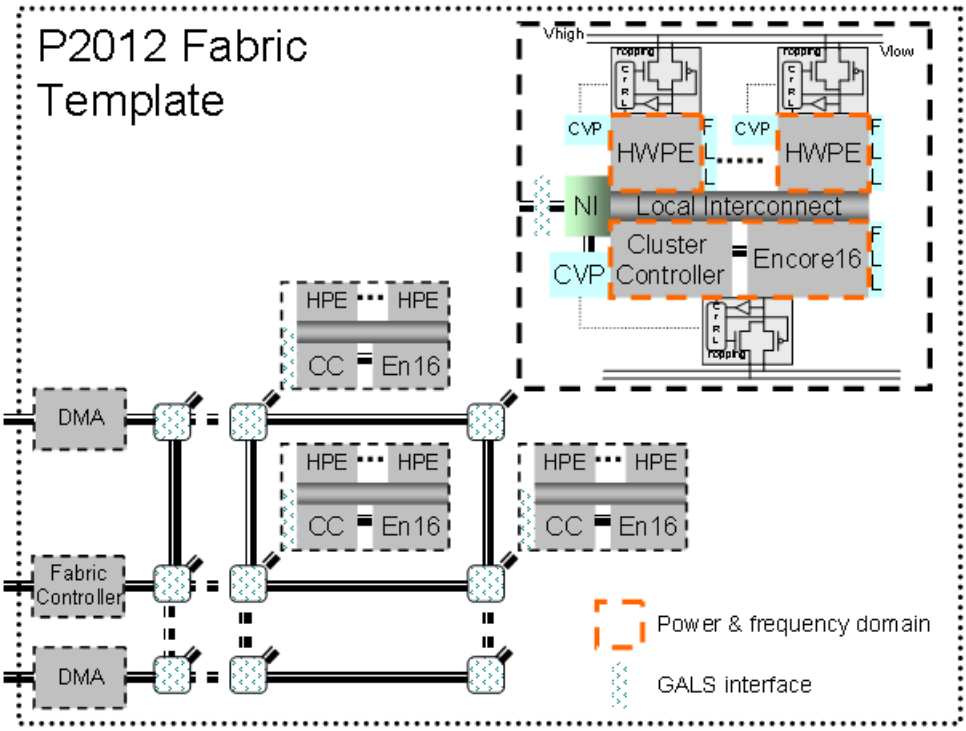


Figure 4.2.3: The P2012 platform.

4.3 Parallelizing and Deploying Software

As we already mentioned in section 4.1, the computing power of multi-processors will not bring real benefits in terms of software performance if the application parallelism is not exploited effectively. In fact, there are two major phases that must be performed to execute a program on a parallel architecture: the parallelization of the application, and its deployment (ie mapping and scheduling). In the remainder of this section, we briefly introduce these two phases and explain why they represent a challenge to the effectiveness of the system, especially for performance and energy consumption.

4.3.1 Parallelization

Parallelization is the process of partitioning the work into parallel tasks related by interdependencies which may be due to communication or synchronization. This phase involves finding and extracting parts of an application which can be executed concurrently, possibly engineering new algorithms to maximize the potential speedup of parallel execution. Basically, there are three sources of parallelism available in an application, and they are commonly known as task, data and pipeline parallelisms.

Task parallelism applies to separate computations that neither have control or data dependencies. As a simple analogy, making coffee and toasting bread are two task-parallel actions in the process of preparing a breakfast. Regarding software, it is the programmer's duty to find parallelism of that type. Indeed, because task-parallelism is inseparable from the algorithms used in the implementation, a machine cannot detect it automatically from a sequential program.

Data parallelism refers to the application of the same computation on different instances of data. As an analogy, consider what happens in a supermarket when several cashiers are serving different clients simultaneously. The function of a cashier, scanning items and collecting money, is the same for each. However the "instances" or customers they treat are different. In this simple example, the parallelization is quite easy. Indeed there are no dependency relationships between customers, and it is possible to serve them in any order. In the field of computer science, an application with this property is called embarassingly parallel, because any extra instance may be handled just by using an extra resource. In this case, the level of parallelism, and therefore the speedup, is only limited by the number of resources one is willing to invest. In other contexts such as video decoding, data parallelization is more complicated as there are *temporal* and *spatial* dependencies on the data set. Interestingly, data parallelism is surely a better candidate for automatic extraction than task parallelism is. A widespread example is the parallelization of loops inside compilers [DRV00].

Pipeline parallelism applies to a chain of actors working in parallel on different instances. Its classical illustration is an assembly line in a firm, where a system is assembled pieces after pieces, progressing through deeper stages of construction. At any instant, each actor on the line is working on a different instance of the system. In the software domain, multimedia and signal applications are naturally expressed as pipelines, because they may functionally be viewed as a chain of actors altering an input stream (a signal or a video).

The goal of parallelization is to express the application at the right level of *granularity*, with the appropriate amount of task, data and pipeline parallelism. This may be viewed as an algorithmic design phase in which the potential for parallelism is identified and exhibited. In this thesis we did not look at the ambitious task of automating the parallelization. Instead, we assume that the sequential application has been *manually* decomposed into a parallel specification consisting of a set of tasks and their interdependencies. Actually, although automatic parallelization is the holy grail of parallel computing, there are strong barriers which make it unapplicable for general programs and architectures. On the other hand, automatic or semi-automatic application deployment is a hot topic since mapping and scheduling decisions, which heavily impact the performance and energy consumption, are good candidates for analysis and optimization by tools.

Still, efficient parallelization is a crucial step for performance because the phase is feeding deployment tools with parallelism, which is their primary resource for working out an efficient implementation of the application on a multi-processor. Actually, as shown by the Amdahl's law [Amd67], the benefit of parallel execution can be severely limited by sequential fractions in the program. Supposing that 10 % of an application is not parallelized, its execution on ten processors brings a maximal speedup of five, not ten. This can be shown by a quick calculation: let T denotes the time for sequential execution, the parallel part takes at least $\frac{0.9}{10} \times T$ to execute (time is divided by the number of processors), and the sequential part takes $0.1 \times T$. The overall speedup is therefore $1/(\frac{0.9}{10} + 0.1) \approx 5.3$. In other words, compared to the ideal case where everything is "fluid" and parallel, the 10% of the work kept sequential has resulted in a theoretical speedup cut by half. The value computed that way is actually an *upper bound*, because in practice communication and synchronisation constraints may limit the acceleration of the parallel part even more. More concretely, the example shows that no matter how good the

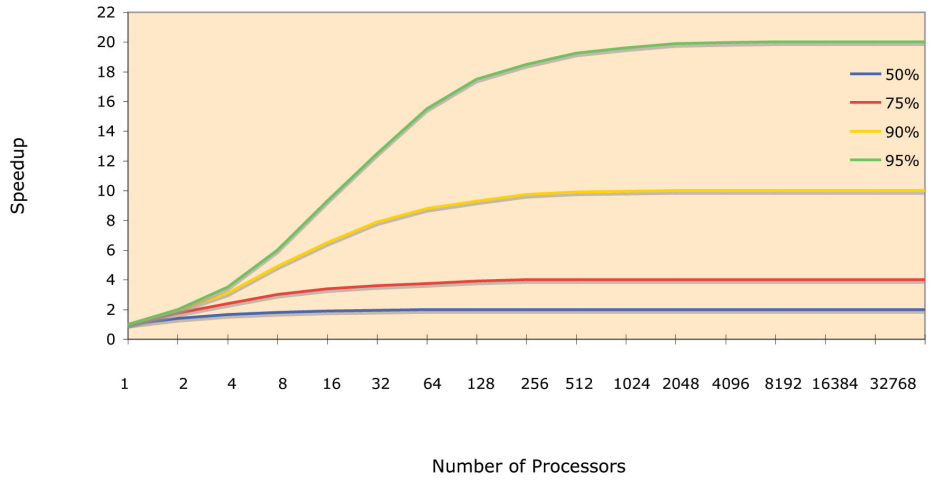


Figure 4.3.1: Illustration of Amdahl's law. The plots represent the speedup of parallel processing for different values of the parallel portion of an application.

mapping and scheduling decisions performed by tools may be, parallel execution speedup is *pre-eminently* conditioned to the amount of parallelism extracted from the application (Figure 4.3.1).

4.3.2 Mapping and Scheduling

Once an application has been parallelized, it remains to deploy it on the multi-processor architecture. This means choosing a processing unit for each task (mapping), and orchestrating the execution over time (scheduling).

Throughout this thesis, we assume that the parallel application is modeled as a *task-graph*, which is a DAG (Directed Acyclic Graph) with nodes representing computations, edges representing communication dependencies, and which is augmented with additionnal information relevant to deployment decisions, such as instruction count, communication volume, memory footprint, energy consumption or deadline. This is a high-level model of an application which makes parallelism explicit and supports different levels of granularity: a task may as well represent an instruction, a thread instance or a communication on a channel. In the same way, we suppose that computation resources are described by an abstraction of their major characteristics, like frequency for a processor, capacity for a memory or bandwidth for a communication channel. The choice of a model (granularity, parameters) for both the application and the platform actually depends on the particular problem to be handled, and on the trade-off between accuracy and tractability.

Constraints and goals widely differ from a system to another, and the optimization problem to solve for the application deployment may take various forms. Obviously the first question to ask is: what are the *static* decisions which can be made about the system? Typically the mapping is not constrained, and can thus be statically decided by formulating an optimization problem. On the other hand, it is not always possible to control task scheduling. For instance, a common practice is to avoid altogether scheduling on the processors by (manually) merging tasks, so that their number is exactly the number of

available processors. Sometimes scheduling is *dynamic*, meaning that decisions concerning the triggering-time of tasks are made on the fly, according to a given resource-sharing policy. Tasks are in this case executed in a cooperative preemption style, that is, data-triggered roundrobin. In the same way, fine-grained communications are generally not scheduled off-line, because the infrastructure (bus, NoC) manages them directly. Still, explicit data transfers such as those initiated by a DMA could well be subject to optimized scheduling. In a nutshell, there are many variants of mapping and scheduling problems, depending on constraints, degrees of freedom, and objectives. In the sequel we successively discuss mapping and scheduling, and show for these two optimization problems a simple formulation.

Mapping is the more liberal form of deployment. It prescribes which tasks are to be executed on which processor and sometimes how data is to be routed among tasks. Hence the decision variables for optimization are discrete variables indicating the placement of tasks and data transfers. The quality of a mapping is evaluated according to various criteria such as power, load balancing, communication traffic, number of resources used or throughput (if the application is a streaming application). Power estimates are often part of the cost funtions because the number and type of processors/links which are statically allocated impact the energy consumption of the system. Most mapping problems are NP-complete. For instance, the problem of minimizing the number of processors used, given a limit on the workload supported by each, reduces to a bin-packing problem [CJGJ96]. Similarly, a one-to-one mapping of a set of n communicating processes to a set of n processors, where the objective is to minimize the total communication cost, is a quadratic assignment problem [Tai91].

The mapping optimization problem can be formulated as described in Equation 4.3A, where a solution is represented as a 0-1 matrix A. Elements of the matrix define the assignment of tasks to processors (and communications to links), where A_{ij} equals one if task i is mapped on processor j, otherwise zero. The vector w_i represents the load of task i. When only the workload is modeled, w_i is a single integer representing a gross measure of the duration of the task. Otherwise, w_i may also have an additional coordinate representing memory utilization, or any other resource usage which should be restricted. Vector W_j is the corresponding capacity of processor j, and $C(A)$ is the multi-dimensional vector of objectives.

$$\begin{aligned} \underset{A}{\text{minimize}} \quad & C(A) \\ \text{subject to} \quad & \forall i \sum_j A_{ij} = 1 \qquad (assigment) \\ & \forall j \sum_i A_{ij} w_i \leq W_j \qquad (capacity) \end{aligned} \tag{4.3A}$$

Generally speaking, a scheduling problem occurs when different activities share a set of scarce resources. In this situation, there are some points in time where two or more activities are in *conflict* for the access to a resource. A schedule is a set of rules prescribing how to resolve the conflicts, by telling which activity gets the resource first.

In particular, the execution of a parallel application entails a scheduling problem, because there is a competition between tasks for the access to processors. In its most static form the scheduling specifies the exact times when a processor executes each of the tasks assigned to it, and in some cases (common in hard real time) it does the same with data transfers on a bus. More relaxed forms of scheduling can be adaptive (event-triggered)

to the actual execution. Scheduling is one of the most studied problem in combinatorial optimization and it has many ramifications associated with different fields of applications (project management, production planning, parallel software execution etc.). In this thesis we only consider task-graph scheduling problems because it is the appropriate model for the type of parallel application we are interested in. We also restrict ourselves to problems involving identical, related or unrelated machines. In general, we do not assume a task to be bound to a specific processor which means that the application scheduling also encompasses its mapping. Except in special cases where the number of processors is limited, task-graph scheduling is NP-complete. The computational hardness of the problem comes from the multiple ways of solving the resource conflicts.

The scheduling optimization problem can be encoded using an assignment matrix A (mapping), plus an additional vector of real variables s, where s_i is the start-time of task i. In the simple formulation we give (Equation 4.3B), the duration of a task does not depend on the processor where it is mapped, what amounts to say that processors are identical. Constraints are those of mapping, augmented with mutual exclusion and precedences induced by the task-graph.

$$
\begin{aligned}
\underset{A,s}{\text{minimize}} \quad & C(A,s) \\
\text{subject to} \quad & \forall i \sum_j A_{ij} = 1 \qquad (assigment) \\
& \forall j \sum_i A_{ij} w_i \leq W_j \qquad (capacity) \\
& \forall i \prec i' \; s_{i'} \geq s_i + d_i \qquad (precedence) \\
& \forall j, i, i' \; A_{ij} = 0 \vee A_{i'j} = 0 \\
& \vee s_{i'} \geq s_i + d_i \vee s_i \geq s_{i'} + d_{i'} \qquad (resource)
\end{aligned}
\tag{4.3B}
$$

4.4 Multi-Criteria Optimization for Deployment Decisions

As outlined in previous sections of this chapter, the multi-core revolution is underway. However, unlike the case of sequential programs for which there are high-level languages and compilers capable of optimizing the implementation, it remains difficult to generate efficient parallel programs automatically. In general, low-level deployment decisions such as mapping and scheduling which have a direct impact on the system performance are optimized manually. Given the complexity of deployment problems and the trend towards large multi-core systems, such ad hoc optimization is very likely to be ineffective.

In fact, optimizing deployment decisions often entails searching among a huge number of possible combinations. For instance there are m^n ways to map an application containing n parallel tasks on a multi-processor platform of size m, a number growing exponentially as n increases. Although it is generally possible to reduce the search space with simple considerations (symmetry breaking, memory limitations etc.), it is clearly impossible to evaluate every alternative by running the actual *code* on low-level instruction- or cycle-accurate simulators. There is a need to develop methods and tools for system-level *design space exploration* in order to identify few promising candidates which are kept for analysis at lower levels of abstraction.

In order to reason efficiently about deployment decisions at a high level, it is required to work with *abstract* models of the application and the multi-processor platform which are

amenable to manipulation by *analytical* methods. Obviously, what is gained in tractability from abstraction, is lost in faithfullness, and the potential of a solution may be wrongly estimated using these coarse models. But the point is, if we put together the complexity of applications, platforms and deployment problems, only methods based on abstract models may explore a significant part of the huge design space of upcoming multi-processor systems.

Given abstract application and platform descriptions, deployment decisions can be formulated as optimization problems. This requires defining a number of constraints (eg resources, precedence, memory) and objective functions. Most of the time it is necessary to take into account several conflicting optimization criteria, such as load balancing versus communication load (Section 1.1.1) or schedule latency versus energy consumption. As argued in Section 1.1.2, an appropriate methodology for decision making under conflicting goals is the multi-criteria approach: first optimize a few objectives simultaneously, with the goal of finding a representative set of the corresponding trade-offs, and then let the user analyse the information and choose among the solutions proposed.

However, it is important to clarify the scope of multi-criteria optimization methods. In fact, since they require users to make the final choice, the decision making performed with their help is only *semi-automatic*. The approach would consequently not be appropriate for optimizing the deployment decisions made by a compiler, which should enable fast testing and debugging of the application. Maybe a way to proceed in this case would be in two stages: first, the program is compiled quickly (without programmer's intervention) but softly optimized, relying on fast heuristics to decide on deployment. Then, once the development is completed, a more involved phase of aggressive program optimization is performed in order to refine deployment decisions and to improve performance, energy consumption, or other metrics which might be relevant for the final system. The multi-criteria approach would be applicable for this second phase.

In the embedded system domain, multi-criteria optimization is used for design space exploration. As mentioned in 4.2, embedded systems are generally specialized for a specific type of applications and there are multiple ways to configure the platform and the application. Design space exploration is the optimization phase during which diverse options for architectural parameters like hardware/software partitioning, memory mapping, processor frequency, or task mapping and scheduling are considered and evaluated.

In order to perform this exploration, the application specification must be clearly separated from the architecture model. This allows to repeatedly test different solutions without having to modify the application code. This design flow methodology is known under the name of Y-chart [BWH+03] and differs from the traditional model of harware/software codesign which has the drawback that performance-related decisions must be made much earlier in the flow. The concept of *system-level design* [KNRSV00] embodies the Y-chart principle of separating the application from the architecture specification, and additionally proposes to raise the level of abstraction at which the system is modeled, analysed and simulated. Performing design space exploration at a high level of abstraction enables to explore as widely as possible a space of configurations which is most of the time exponential in size. By doing so, dominated solutions are pruned before moving down in abstraction, and costly low-level simulations are only performed for the most promising candidates. This is where high-level analytic methods and multi-criteria optimization have their utility: they may help *pruning* the design space of embedded systems early in the flow when the number of solutions is too large for relying on simulations [Erb06]. As an

example, Sesame [EPTP07, ECEP06] and DOL [TBHH07] are two recent system-level design-space-exploration frameworks which feature a multi-objective optimization step based on evolutionary algorithms.

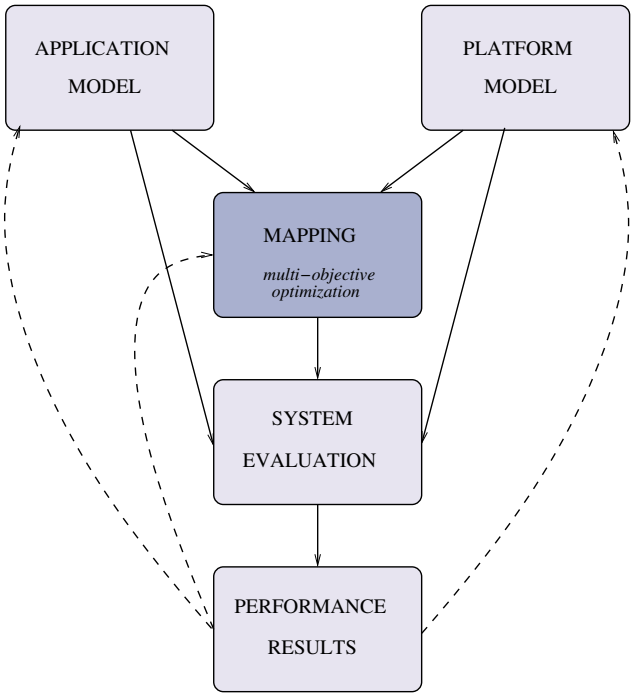


Figure 4.4.1: The Y-chart methodology separates the application and architecture descriptions for design space exploration. Multi-criteria optimization can be used during the mapping phase (application to architecture). After the mapping, the system is evaluated through performance analysis or simulation. The results are used to progressively refine the initial application and architecture models, and to improve the mapping.

Chapter 5

Energy-Aware Scheduling

Résumé : *Dans ce chapitre nous appliquons les méthodes d'optimisation proposées dans les chapitres 2 et 3 au problème d'ordonnancement sur architecture multi-processeur. Nous étudions différentes versions du problème en considérant deux objectifs : la latence de l'ordonnancement et la consommation d'énergie. Nous considérons d'abord une plateforme d'éxécution dans laquelle la fréquence des processeurs peut être configurée, et formulons le problème suivant : quel est la configuration optimale en termes de consommation d'énergie qui permet d'ordonnancer le graphe de tâches de l'application sans dépasser une latence maximale prédéfinie ? Ce problème a été traité en combinant un solveur SAT et l'algorithme développé dans le chapitre 2. Nous avons ensuite formulé deux extensions : l'ordonnancement des transferts de données sur les canaux de communications, et le cas où différentes instances du même graphe arrivent périodiquement dans le système. Dans le cas périodique, il est nécéssaire d'éxécuter plusieurs instances en parallèle (pipeline) afin d'exploiter au mieux les capacités de la plateforme. La taille des problèmes qui peuvent être résolus par cette méthode est relativement petite (entre 15 et 40 selon les variantes) mais la qualité de la ou les solutions est garantie. L'algorithme de recherche locale présenté au chapitre 3 a été testé sur un problème légèrement différent pour lequel la fréquence des processeurs est considérée fixe, mais qui intègre un modèle de la consommation énergétique plus réaliste. Cette méthode antagoniste à la première permet d'obtenir de bonnes solutions de manière très rapide (quelques secondes) mais sans garantie sur la qualité.*

5.1 Introduction

In this chapter, we apply the optimization techniques presented in chapters 2 and 3 to the problem of scheduling an application on a multi-processor system. We study different versions of the scheduling problem while considering objectives related to performance and energy consumption.

First, we define the abstract models used to represent the multi-processor architecture and the parallel application (Section 5.2). We then consider an execution platform in which the frequency of processing units can be configured, and formulate the problem of finding the *cheapest configuration* which meets the application demands. More formally we ask the following question: what is the least energy-demanding configuration of the architecture

on which a task-graph can be scheduled while meeting a strict deadline constraint? For this problem we use a coarse model of energy where we associate with each running processor a *static* cost related to its operating frequency. We report different results of applying the SMT-based optimization method presented in Chapter 2. Several single- and multi-criteria variants of this problem are treated, including the case where a stream of identical task-graphs arrives *periodically*.

In the second part we describe an implementation of the stochastic local search algorithm presented in Chapter 3 to solve a bi-criteria energy/deadline scheduling problem. In particular we detail how to represent a schedule and its neighborhood in order to efficiently perform a locally-optimal step. The algorithm we developed only supports a fixed multi-processor architecture where the speeds are pre-determined and are not part of the decision variables. On the other hand, the energy model used is more refined: each task induces a cost which depends on its duration and the speed of the processor where it executes. Moreover, the implementation offers the possibility to account for the energy spent on communications by associating a cost with each data transfer occuring on the network. This cost is based on the size of the data item and the length of the routing path.

Finally, we conclude this chapter by evaluating the results obtained by the two alternative multi-criteria optimization methods we have developed during this thesis. We mainly stress the advantages and drawbacks we could identify in both techniques, and discuss their applicability for solving real instances of mapping and scheduling problems.

5.2 Problem Specification

This section is dedicated to the definition of abstract models for the (configurable) multi-processor platform and the parallel application.

5.2.1 Execution Platforms

We assume a fixed set of speeds $V = \{v_0, v_1, \ldots, v_m\}$ with $v_0 = 0$ and $v_k < v_{k+1}$, each representing an amount of work (for example, number of instructions) that can be provided per unit of time. We propose the following model for a multi-processor system where speeds can be configured according to V.

Definition 5.2.1 (Execution Platform) *A configurable execution platform is a tuple* $E = (M, R, N, b, \rho)$ *where*

- $M \in \mathbb{N}$ *is a finite number of processors*
- $R \in V^M$ *is a vector assigning a speed to each machine called the platform configuration. We interpret* $R_i = 0$ *as saying that processor* i *is shut down in the configuration.*
- $N \subseteq \{0 \ldots M-1\}^2$ *is a set of (physical) communication channels all with the same bandwidth* b *(bits/second)*
- ρ *is a routing function* $\rho : \{0 \ldots M-1\}^2 \to N^*$ *which assigns to every pair* (m, m') *of processors an acyclic path* $(m, m_1), (m_1, m_2), \ldots (m_k, m')$ *in the network. Clearly* $\rho(m, m) = \epsilon$.

By abuse of notation we also refer to the path as a *set* of channels and say that $(m_1, m_2) \in \rho(m, m')$ if (m_1, m_2) appears in the path. Below we define some useful measures on platform configurations related to their work capacity.

- Number of active machines: $\mathcal{N}(R) = |i \in \{0 \dots M-1\}, R_i \neq 0|$;
- Speed of fastest machine: $\mathcal{F}(R) = \max_i R_i$;
- Work capacity: $\mathcal{S}(R) = \sum_{i=1}^{M-1} R_i$.

The last measure gives an upper bound on the quantity of work that the platform may produce over time when it is fully utilized.

5.2.2 Work Specification

The work to be done on a platform is specified by a variant of a *task graph*, where the size of a task is expressed in terms of *work* rather than *duration*.

Definition 5.2.2 (Task Graph) *A task graph is a tuple $G = (P, \prec, w, v)$ where $P = \{p_1, \dots p_n\}$ is a set of tasks, $\prec$ is a partial order relation on P with a unique*[1] *minimal element p_1 and a unique maximal element p_n. The function $w : P \to \mathbb{N}$ assigns a quantity of work to each task and $v : P \times P \to \mathbb{N}$ indicates the amount of data communicated between each pair of tasks (p, p') such that $p \prec p'$. When a task p is executed on a machine working in speed v, its execution time is $w(p)/v$.*

The following measures give an approximate characterization of what is needed in terms of time and work capacity in order to execute G.

- The width $\mathcal{W}(G)$ which is the maximal number of $\prec$-incomparable tasks, indicates the maximal parallelism that can potentially be useful;
- The length $\mathcal{L}(G)$ of the longest (critical) path in terms of work, which gives a lower bound on execution time;
- The total amount of work $\mathcal{T}(G) = \sum_i w(p_i)$ which together with number of machines gives another lower bound on execution time.

5.3 Satisfiability-Based Scheduling

In this section we solve the problem of finding platform configurations and associated energy-efficient schedules of the task-graph using SMT solvers. We first discuss briefly several approaches for handling mixed constraints in optimization (Section 5.3.1), and describe our encoding of the basic problem using a Boolean combination of linear constraints (Section 5.3.2).

We have first treated the problem with the single objective of minimizing the platform cost *given* a fixed deadline. We have developed a program which translates the problem into the Yices solver input language [DdM06], and perform the optimization using a one-dimensional version of Algorithm 2.4.1 which is enhanced by a simple strategy for handling non-terminating calls. We then improve the implementation to solve two orthogonal extensions of the problem. First we consider inter-task *communications* and pose the same question where the network channels are considered as additional resources occupied for durations proportional to the amount of transmitted data (Section 5.3.4). Then, we formulate a *periodic* extension where task-graph instances arrive every ω time units and must be finished within a relative deadline $\delta > \omega$ (Section 5.3.5). Efficient resource utilization for this problem involves *pipelined* execution, which may increase the number

1. This can always be achieved by adding fictitious minimal and maximal tasks with zero work.

of decision variables and complicate the constraints. We solve this infinite problem by reducing it to finding a schedule for a sufficiently-long finite unfolding.

We then move on to *bi-criteria* optimization. Rather than keeping the deadline constant and optimizing the cost, we use the Algorithm 2.4.1 presented in Chapter 2 to provide a good approximation of the trade-off curve between cost and deadline. We formulate the same problem including the scheduling of communications, but we also derive and experiment a more tractable model which only accounts for the latency of data transfers. For these developments we used the API of the SMT solver Z3 [DMB08] which can save information between the various calls.

5.3.1 Background

The problem of optimizing a linear function subject to linear constraints, also known as linear programming [Sch99], is one of the most studied optimization problems. When the set of feasible solutions is *convex*, that is, it is defined as a *conjunction* of linear inequalities, the problem is easy: there are polynomial algorithms and, even better, there is a simple worst-case exponential algorithm (simplex) that works well in practice. However, all these nice facts are not of much help in the case of scheduling under resource constraints. The mutual exclusion constraint, an instance of which appears in the problem formulation for every pair of unrelated tasks executing on the same machine, is of the form $[x, x'] \cap [y, y'] = \emptyset$, that is, a *disjunction* $(x' \leq y) \vee (y' \leq x)$ where each disjunct represents a distinct way to solve the potential resource conflict between the two tasks. As a result, the set of feasible solutions is decomposed into an exponential number of disjoint convex sets, a fact which renders the nature of the problem more combinatorial than numerical. Consequently, large scheduling problems cannot benefit from progress in relaxation-based methods for mixed integer-linear programming (MILP).

Techniques that are typically applied to scheduling problems [BLPN01] are those originating from the field known as constraint logic programming (CLP) [JM94]. These techniques are based on heuristic search (guessing variable valuations), constraint propagation (deducing the consequences of guessed assignments and reducing the domain of the remaining variables) and backtracking (when a contradiction is found). A great leap in performance has been achieved during the last decade for search-based methods for the generic discrete constraint satisfaction problem, the satisfiability of Boolean formulae given in CNF form (SAT). Modern SAT solvers [ZM02] based on improvements of the DPLL procedures [DLL62] can now solve problems comprising of hundreds of thousands of variables and clauses and are used extensively to solve design and verification problems in hardware and software.

Recently, efforts have been made to leverage this success to solve satisfiability problems for Boolean combinations of predicates, such as numerical inequalities, belonging to various "theories" (in the logical sense), hence the name *satisfiability modulo theories* (SMT) [GHN+04, BSST09]. SMT solvers combine techniques developed in the SAT context (search mechanism, unit resolution, non-chronological backtracking, learning, and more) with a theory-specific solver that checks the consistency of truth assignments to theory predicates and infers additional assignments. The relevant theory for standard scheduling problems is the theory of *difference constraints*, a sub theory of the theory of linear inequalities, but in order to cover costs and speeds we use the latter theory.

5.3.2 Constrained Optimization Formulation

We first formulate the scheduling problem without considering the communications. In this section we assume a task-graph $G = (P, \prec, w, v)$ where $v = 0$. This assumption implies that processors are symmetric entities, and the execution platform is fully represented by its configuration vector R which specifies the number of active processors and their respective speeds. Hence, we will just represent the execution platform by its configuration vector R for the remainder of this section.

We define a schedule for the pair (G, R) to be a function $s : P \to \mathbb{N} \times \mathbb{R}_+$ where $s(p) = (j, t)$ indicates that task p starts executing at time t on machine j. We will sometime decompose s into s_1 and s_2, the former indicating the machine and the latter, the start time. The duration of task p under schedule s is $d_s(p) = w(p)/R_{s_1(p)}$. Its execution interval (we assume no preemption) is $[s_2(p), s_2(p) + d_s(p)]$. A schedule is feasible if the execution intervals of tasks satisfy their respective precedence constraints and if they do not violate the resource constraints, which means that they are disjoint for all tasks that use the same machine.

Definition 5.3.1 (Feasible Schedule) *A schedule s is feasible for G on platform R if it satisfies the following conditions:*

1. Precedence*: If $p \prec p'$ then $s_2(p) + d_s(p) \leq s_2(p')$;*
2. Mutual exclusion*: If $s_1(p) = s_1(p')$ then $[s_2(p), s_2(p) + d_s(p)] \cap [s_2(p'), s_2(p') + d_s(p')] = \emptyset$. The total duration of a schedule is the termination time of the last task $\ell(s) = s_2(p_n) + d_s(p_n)$.*

To compare different platforms we use a *static* cost model that depends only on the membership of machines of various speeds in the platform and not on their actual utilization during execution.

Definition 5.3.2 (Platform Cost) *The cost of a platform R is $C(R) = \sum_{i=0}^{M-1} \alpha(R_i)$ where $\alpha : V \to \mathbb{N}$ associates a static cost to each speed.*

The singular[2] deadline scheduling predicate $\text{SDS}(G, R, \delta)$ holds if there is a feasible schedule s for G on R such that $\ell(s) \leq \delta$. The single-objective problem of finding the cheapest architecture R where, for a given δ, the task-graph is schedulable in time is

$$\min C(R)\ s.t\ .\text{SDS}(G, R, \delta) \tag{5.3A}$$

The harder part of the problem is to check whether, for a given execution platform R, $\text{SDS}(G, R, \delta)$ is satisfied when δ is close to the duration of the optimal schedule for G on R. Since we will be interested in approaching the cheapest platform we will often have to practically solve the optimal scheduling problem which is NP-hard.

Let us mention some observations that reduce the space of platforms that need to be considered. First, note that if $\text{SDS}(G, R, \delta)$ is solvable, then there is a solution on a platform R satisfying $\mathcal{N}(R) \leq \mathcal{W}(G)$, because adding processors beyond the potential parallelism in G does not help. Secondly, a feasible solution imposes two lower bounds on the capacity of the platform:

2. To distinguish it from the periodic problem defined in Section 5.3.5

1. The speed of the fastest machine should satisfy $\mathcal{F}(R) \geq \mathcal{L}(G)/\delta$, otherwise there is no way to execute the critical path before the deadline.
2. The total work capacity should satisfy $\mathcal{S}(R) \geq \mathcal{T}(G)/\delta$, otherwise even if we manage to keep the machines busy all the time we cannot finish the work on time

Solutions to the problem are assignments to decision variables $\{u_j\}$, $\{e_i\}$ and $\{x_i\}$ where variable u_j ranging over V, indicates the speed of machine j, integer variables e_i indicates the machine on which task p_i executes and variable x_i is its start time. Constants $\{v_k\}$ indicates the possible speeds, $\{w_i\}$ stand for the work in tasks and $\{c_k\}$ is the cost contributed by a machine running at speed v_k. The above-mentioned lower bounds on the speed of the fastest machine and on the platform capacity are b_1 and b_2, respectively. We use auxiliary (derived) variables $\{d_i\}$ for the durations of tasks based on the speed of the machine on which they execute and C_j for the cost of machine j in a given configuration. The following constraints define the problem.

– The speed of a machine determines its cost:

$$\bigwedge_j \bigwedge_k (u_j = v_k \Rightarrow C_j = c_k)$$

– Every task runs on a machine with a positive speed and this defines its duration:

$$\bigwedge_i \bigwedge_j ((e_i = j) \Rightarrow (u_j > 0 \wedge d_i = w_i/u_j))$$

– Precedence: $\bigwedge_i \bigwedge_{i':p_i \prec p_{i'}} x_i + d_i \leq x_{i'}$

– Mutual exclusion:

$$\bigwedge_i \bigwedge_{i' \neq i} ((e_i = e_{i'}) \Rightarrow ((x_i + d_i \leq x_{i'}) \vee (x_{i'} + d_{i'} \leq x_i))$$

– Deadline: $x_n + d_n \leq \delta$

– Total architecture cost: $C = \sum_j C_j$.

We use additional constraints that do not change the satisfiability of the problem but may reduce the space of the feasible solutions. They include the above mentioned lower bounds on architecture size and a “symmetry breaking" constraint which orders the machines according to speeds, avoiding searching among equivalent solutions that can be transformed into each other by permuting the indices of the machines.

– Machines are ordered: $\bigwedge_j (u_j \geq u_{j+1})$

– Lower bounds on fastest processor and capacity:

$$(u_1 > b_1) \wedge \sum_j u_j > b_2$$

5.3.3 Implementation and Experimental Results

We have implemented a prototype tool that takes a task graph as an input, performs preliminary preprocessing to compute width, critical path and quantity of work and then generates the constraint satisfaction problem in the Yices solver input language. Recall

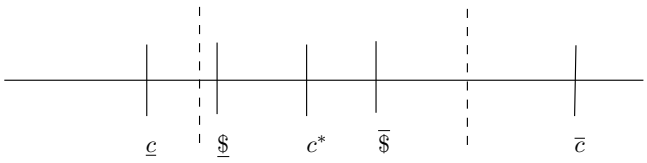


Figure 5.3.1: Searching for the optimum. The dashed line indicate the interval toward which the algorithm will converge, the best estimation of the optimum for time budget θ.

that the solver has no built-in optimization capabilities, and we can pose only queries of the form $\text{SDS}(G, R, \delta) \wedge C(R) \leq \mathbf{c}$ for some cost vector $\mathbf{c}$. We use $\psi(\mathbf{c})$ as a shorthand for this query.

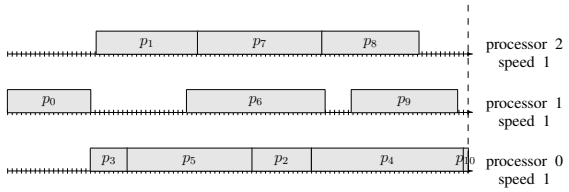
The quest for a cheap architecture is realized as a sequence of calls to the solver with various values of $\mathbf{c}$, which basically is a one dimensional version of Algorithm 2.4.1. Performing a search with an NP-hard problem in the inner loop is very tricky since, the closer we get to the optimum, the computation time becomes huge, both for finding a satisfying solution and for proving unsatisfiability (from our experience, the procedure may sometimes get stuck for hours near the optimum while it takes few seconds for slightly larger or smaller costs). We have implemented a simple mechanism to handle such non-terminating calls, which is based on the ideas presented in Section 2.6.1 [3]. We fix a time budget θ beyond which we do not wait for an answer (currently 5 minutes on a modest laptop). The outcome of the query $\psi(\mathbf{c})$ can be either

$$\begin{array}{ccl} (1, c) & : & \text{There is a solution with cost } c \leq \mathbf{c} \\ 0 & : & \text{There is no solution} \\ \$ & : & \text{The computation is too costly} \end{array}$$

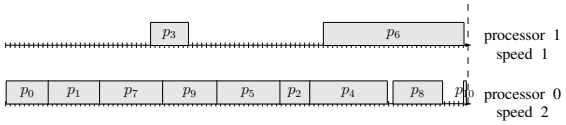
At every stage of the search we maintain 4 variables: $\underline{c}$ is the maximal value for which the query is not satisfiable, $\underline{\$}$ is the minimal value for which the answer is $\$$, $\overline{\$}$ is the maximal value for which the answer is $\$$, and $\overline{c}$ is the minimal solution found (see Figure 5.3.1). Assuming that computation time grows monotonically as one approaches the optimum from both sides, we are sure not to get answers if we ask $\psi(\mathbf{c})$ with $\mathbf{c} \in [\underline{\$}, \overline{\$}]$. So we ask further queries in the intervals $[\underline{c}, \underline{\$}]$ and $[\overline{\$}, \overline{c}]$ and each outcome reduces one of these intervals by finding a larger value of $\underline{c}$, a smaller value of $\underline{\$}$, a larger value of $\underline{\$}$ or a smaller value of $\overline{c}$. Whenever we stop we have a solution $\overline{c}$ whose distance from the real optimum is bounded by $\overline{c} - \underline{c}$. This scheme allows us to benefit from the advantages of binary search (logarithmic number of calls) with a bounded computation time.

We did experiments with this algorithm on a family of platforms with 3 available speeds $\{1, 2, 3\}$. The costs associated with the speeds are, respectively, 1, 8 and 27, reflecting the approximate cubic dependence of energy on speed. We have experimented with numerous graphs generated by TGFF tool [DRW98] and could easily find solutions for problems with 40-50 tasks. Figure 5.3.2 illustrates the influence of the deadline constraints on the platform and the schedule for a 10-task problem of width 4. With deadline 100 the problem can be scheduled on the platform $R = (0, 0, 3)$, that is, 3 slow processors, while when the deadline is reduced to 99, the more expensive platform $R = (0, 1, 1)$ is needed. In the next section we discuss how the model has been enriched to account for communications between tasks.

3. However this mechanism only works for the single criteria case that we treat here.



(a)



(b)

Figure 5.3.2: The effect of deadline: (a) a cheap architecture with deadline 100; (b) a more expensive architecture with deadline 99.

5.3.4 Adding Communications

The model described in the preceding section neglects communication costs. While this may be appropriate for some traditional applications of program parallelization, it is less so for modern streaming applications that have to pass large amounts of data among tasks and there is a significant variation in communication time depending on the *distance* between the processors on which two communicating tasks execute. From now on, we consider a task-graph G with $v \neq 0$, and extend the constrained optimization formulation to account for communications. We assume that a task waits for the arrival of *all* its data before starting execution, and that it transmits them *after* termination.

We propose to schedule communications on network links in the same way that we schedule tasks on processors. Hence when a task p and its successor p' are mapped to m and m' respectively, the termination of p should be followed by the execution of additional communication tasks that have to be scheduled successively on the channels on the path from m to m'. Task p' should not start executing before the last of those communication tasks has finished (Figure 5.3.3). In order to model the problem in the SMT solver we introduce new real variables for the start-time of each communication task. Each edge z of the task-graph is associated with η real variables $\{o_{z-k}\}_{k=1}^{\eta}$, where η is the maximal length of any path returned by the routing function, and o_{z-k} is the start-time of the k-th communication task corresponding to edge z. When the communication due to edge z is forwarded through a path with $n < \eta$ hops, all o_{z-k} variables for $k \geq n$ are irrelevant. In the sequel we informally explain the additionnal constraints induced by this model, but spare the reader from the exact formulation which is quite involved.

The precedence constraints are obviously modified with the introduction of communications, and in particular they become dependent on the task-to-processor assignment. Consider an edge z which links two tasks $i \prec i'$. When i and i' are respectively mapped on processors j and j' with $|\rho(j, j')| = n$, the following constraint is implied

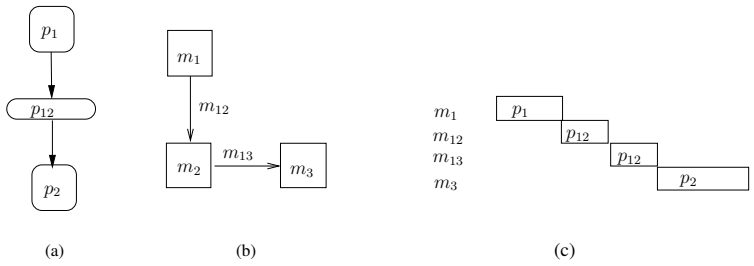


Figure 5.3.3: (a) a part of a task-graph with data transmission from p_1 to p_2 denoted by p_{12}; (b) a part of an architecture where communication from m_1 to m_3 is routed via channels m_{12} and m_{23}; (c) a part of a schedule, including communication, where p_1 executes on m_1 and p_2 on m_3.

$$\begin{array}{lllllllll} o_{z-0} & \geq x_i + d_i & \wedge & o_{z-1} & \geq & o_{z-0} + v(i,i') & & & \\ & & \wedge & \ldots & & & & & \\ & & \wedge & o_{z-n} & \geq & o_{z-n-1} + v(i,i') & \wedge & x'_i & \geq & o_{z-n} + v(i,i') \end{array} \tag{5.3B}$$

The left part of Equation 5.3B forces the start-time of the first communication to occur after the termination of the predecessor task. The right part reciprocally imposes the start-time of the successor to be greater than the termination of the last communication task, and the constraints in the middle prescribe that each router along the path should not begin transmission before it receives the data. Additionally the conflicts on the network are encoded by similar resource constraints as for tasks. When the k-th link on the path used for the communication due to an edge $z = (i_1, i_2)$ is the same than the k'-st link used for communication due to edge $z' = (i_3, i_4)$, the implied constraint is

$$o_{z-k} \quad \geq \quad o_{z'-k'} + v(i_1, i_2) \quad \vee \quad o_{z'-k'} \quad \geq \quad o_{z-k} + v(i_3, i_4) \tag{5.3C}$$

We have coded this problem and ran experiments with TGFF-generated graphs to find cheap and deadline-satisfying configurations of an architecture with up to 8 processors, $\{0, 1, \ldots, 7\}$ equipped with a *Spidergon* network topology developed in STMicroelectronics [CLM+04]. A spidergon network has links of the form $(i, i+1 \; mod \; 8)$, $(i, i-1 \; mod \; 8)$ and $(i, i+4 \; mod \; 8)$ for every i and there is a path of length at most 2 between any pair of processors (Figure 5.3.4). In this setting we could solve easily problems with 15-20 tasks. Figures 5.3.5 shows the effect of the deadline on the architecture for a task-graph with 16 tasks. For a deadline of 25 the task graph can be scheduled on an architecture with 3 machines while for deadline 24, 4 machines are needed. Note that the schedule for the first case is very tight and is unlikely to be found by heuristics such as list scheduling.

5.3.5 Periodic Scheduling

In this section we extend the problem to deal with a *stream* of instances of G that arrive periodically [4].

4. In order to keep the problem tractable, we add the periodic extension to the original problem where we ignore communications.

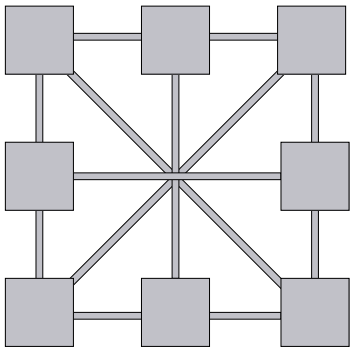


Figure 5.3.4: The spidergon network.

Definition 5.3.3 (Periodic Scheduling Problem) *Let ω (arrival period) and δ be positive integers. The periodic deadline scheduling problem* PDS(G, R, ω, δ) *refers to scheduling an infinite stream $G[0], G[1], \ldots$ of instances of G such that each $G[h]$ becomes available for execution at $h\omega$ and has to be completed within a (relative) deadline of δ, that is, not later than $h\omega + \delta$.*

Let us make some simple observations concerning the relation between SDS and PDS for fixed G and R.

1. PDS$(\omega, \delta) \Rightarrow$ SDS(δ). This is trivial because if you cannot schedule one instance of G in isolation you cannot schedule it when it may need to share resources with tasks of other instances.
2. If $\delta \leq \omega$ then PDS$(\omega, \delta) \Leftrightarrow$ SDS(δ) because in this case each instance should terminate *before* the arrival of the next instance and hence in any interval $[h\omega, (h + 1)\omega]$ one has to solve one instance of SDS(δ). Thus we consider from now on that $\omega < \delta$.
3. Problem PDS(ω, δ) is solvable only if $W(G) \leq \omega\mathcal{S}(R)$. The quantity of work demanded by an instance should not exceed the amount of work that the platform can supply within a period. Otherwise, backlog will be accumulated indefinitely and no finite deadline can be met.
4. When SDS(ω) is not solvable, PDS(ω, δ) can only be solved via *pipelined* execution, that is, executing tasks belonging to successive instances simultaneously.

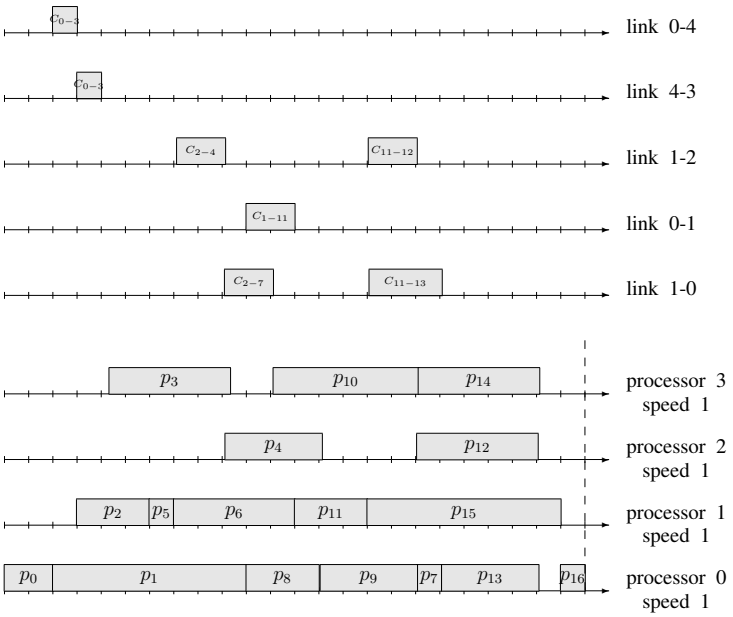
We first encode the problem using infinitely many copies of the task related decision variables where $x_i[h]$ and $e_i[h]$, denotes, respectively, the start time of instance h of task p_i and the machine on which it executes, which together with the speed of that machine determines its duration $d_i[h]$. The major constraints that need to be added or modified are:

– The whole task graph has to be executed between its arrival and its relative deadline:

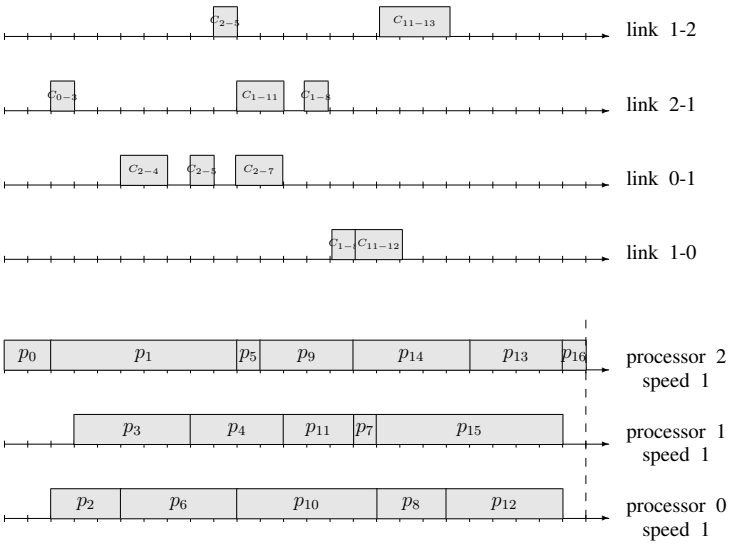
$$\bigwedge_{h \in \mathbb{N}} x_1[h] \geq h\omega \ \wedge \ x_n[h] + d_n[h] \leq h + \delta$$

– Precedence:

$$\bigwedge_{h \in \mathbb{N}} \bigwedge_{i} \bigwedge_{i' : p_i \prec p_{i'}} x_i[h] + d_i[h] \leq x_{i'}[h]$$



(a)



(b)

Figure 5.3.5: Scheduling with communication: (a) $\delta = 24$; (b) $\delta = 25$.

– Mutual exclusion: execution intervals of two *distinct* task instances that run on the same machine do not intersect.

$$\bigwedge_{j}\bigwedge_{i,i'}\bigwedge_{h,h'}(i \neq i' \vee h \neq h') \wedge e_i[h] = e_{i'}[h'] \Rightarrow$$
$$(x_i[h] + d_i[h] < x_{i'}[h']) \vee (x_{i'}[h'] + d_{i'}[h'] < x_i[h])$$

Note that the first two constraints treat every instance *separately* while the third is machine centered and may involve several instances due to pipelining.

While any satisfying assignment to the infinitely-many decision variables is a solution to the problem, we are of course interested in solutions that can be expressed with a finite (and small) number of variables, solutions which are *periodic* in some sense or another, that is, there is an integer constant β such that for every h, the solution for instance $h + \beta$ is the solution for instance h shifted in time by $\beta\omega$.

Definition 5.3.4 (Periodic Schedules)

– *A schedule is β-machine-periodic if for every h and i, $e_i[h + \beta] = e_i[h]$;*
– *A schedule is β-time-periodic if for every h and i, $x_i[h + \beta] = x_i[h] + \beta\omega$;*
– *A schedule is (β_1, β_2)-periodic if it is β_1-machine-periodic and β_2-time-periodic.*

We say that β-periodic solutions are *dominant* for a class of problems if the existence of a solution implies the existence of a β-periodic solution. It is not hard to see that there is some β for which β-periodic solutions are dominant: the deadlines are bounded, all the constants are integers (or can be normalized into integers), hence we can focus on solutions with integer start times. Combining with the fact that overtaking (executing an instance of a task before an older instance of *the same* task) can be avoided, we end up with a finite-state system where any infinite behavior admits a cycle which can be repeated indefinitely. However, the upper bound on dominant periodicity derived via this argument is too big to be useful.

It is straightforward to build counter-examples to dominance of β-machine-periodic schedules for any β by admitting a task of duration $d > \beta\omega$ and letting $\delta = d$. Each instance of this task has to be scheduled immediately upon arrival and since each of the preceding β instances will occupy a machine, the new instance will need a different machine. However, from a practical standpoint, for reasons such as code size which are not captured in the current model, it might be preferable to execute all instances of the same task on the same processor and restrict the solutions to be 1-machine-periodic. For time periodicity there are no such constraints unless one wants to use a very primitive runtime environment.

We encode the restriction to (β_1, β_2)-periodic schedules as an additional constraint.

– Schedule periodicity:

$$\bigwedge_{h\in\mathbb{N}}\bigwedge_{i} e_i[h + \beta_1] = e_i[h] \wedge x_i[h + \beta_2] = x_i[h] + \beta_2\omega$$

We denote the infinite formula combining feasibility and (β_1, β_2)-periodicity by $\Phi(\beta_1, \beta_2)$ and show that it is equi-satisfiable with a finite formula $\Phi(\beta_1, \beta_2, \gamma)$ in which h ranges over the finite set $\{0, 1, \ldots \gamma - 1\}$. In other words, we show that it is sufficient to find a feasible schedule to the finite problem involving the *first* γ instances of G.

Claim 1 (Finite Prefix)
Problems $\Phi(\beta_1, \beta_2)$ and $\Phi(\beta_1, \beta_2, \gamma)$ are equivalent when $\gamma \geq \beta + \left\lceil \frac{\delta}{\omega} \right\rceil - 1$. where $\beta = lcm(\beta_1, \beta_2)$.

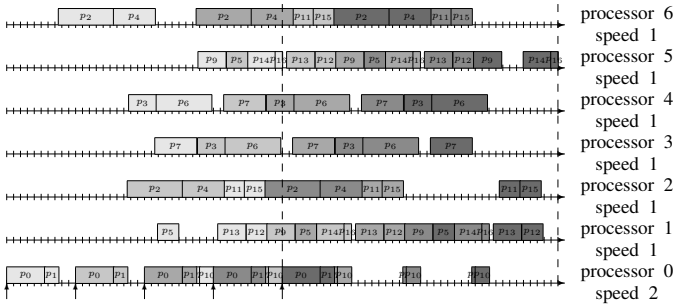


Figure 5.3.6: A $(2, 1)$-periodic schedule for a task graph with 16 tasks.

Proof: Intuitively, the prefix should be sufficiently *long* to demonstrate repeatability of a segment where β instances are scheduled, and sufficiently *late* to demonstrate steady-state behavior where resource constraints are satisfied by the maximal number of instances whose tasks may co-exist simultaneously without violating the deadline.

Suppose we scheduled $\gamma = \beta + \lceil \frac{\delta}{\omega} \rceil - 1$ instances successfully. Since $\gamma \geq \beta$ we can extract a pattern that can be repeated to obtain an infinite schedule that clearly satisfies β-periodicity and precedence constraints. We now prove that this periodic schedule satisfies resource constraints. Suppose on the contrary that instance h of a task i is in conflict with instance h' of a task i', $h \leq h'$ and let $h = (k\beta + k'),\ k' \in \{0 \dots \beta - 1\}$. The deadline constraint, together with the fact that task i and i' overlap in time, limits the set of possible values for h' to be of the form

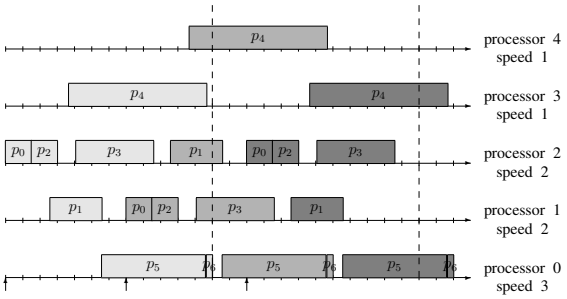
$$h' = h + \Delta = k\beta + k' + \Delta$$

with $\Delta \in \{0 \dots \lceil \frac{\delta}{\omega} \rceil - 1\}$. Because of β-periodicity we can conclude that task i and i' also experience a conflict in the respective instances k' and $k' + \Delta$. Since $k' < \beta$ and $\Delta \leq \lceil \frac{\delta}{\omega} \rceil - 1$ we have

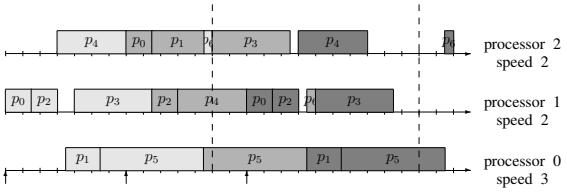
$$k' + \Delta < \beta + \left\lceil \frac{\delta}{\omega} \right\rceil - 1$$

which contradicts our assumption. ∎

Hence the problem can be reduced to $\Phi(\beta_1, \beta_2, \gamma)$ with γ copies of the task-related decision variables. Note however that in fact there are only β "free" copies of these variables and the values of the other variables are tightly implied by those. The unfolding is required only to add additional resource constraints, not decision variables. Table 5.1 shows performance results on several graphs with different values of δ/ω, β_1 and β_2. In general we can treat problems where the number of unfolded tasks is around 100. Figure 5.3.6 shows a $(2, 1)$-periodic schedule for a task graph 1 of Table 5.1 with 17 tasks and $\delta/\omega = 4$ which requires 5 unfoldings. Although there are 85 tasks the problem is solved quickly in 3 minutes as there are only 34 decision variables. Figure 5.3.7 shows an example where making β_2 larger improves the solution. We consider a task-graph of 5 tasks, $\omega = 7$ and $\delta = 12$ with machine periodicity $\beta_1 = 2$. When we impose 1-time-periodicity we get a solution with 5 machines and cost 45, while when we move to 2-time-periodicity we can do with 3 machines and cost 43.



(a)



(b)

Figure 5.3.7: The effect of time periodicity: (a) A $(2, 1)$-periodic solution; (b) A cheaper $(2, 2)$-periodic schedule. Tasks p_1 and p_3 are not executed 1-periodically.

5.3.6 Bi-Criteria Optimization

In this section we treat the bi-criteria problem of finding trade-offs between architecture cost and deadline, which is formulated as:

$$\min(C(E), \delta) \; s.t \; .\text{SDS}(G, E, \delta) \tag{5.3D}$$

We encoded this problem using the C API of SMT solver Z3 [5]. Our program approximates the Pareto front using Algorithm 2.4.1 and returns a set of trade-offs with the associated platform-configurations and schedules. We also chose to implement a simpler model of communications, which we describe in the sequel.

In this model, which is best relevant for architectures where communications are handled directly by the network, we assume no control on the scheduling of communications, but we extend the formulation to account for the latency of data transfers. Obviously, the delay of a communication increases with the size of the data and the length [6] of the routing path between the source and the destination processors. Hence we propose to modify the precedence constraints to force a task p' such that $p \prec p'$ and $v(p, p') \neq 0$, to wait for a time proportional to the quantity $v(p, p')|\rho(a_p, a_{p'})|$. When two tasks are running on the same machine we assume the communication time to be negligible. This model ignores the resource conflicts which can occur between data transfers on the network, a fact which simplifies the constraints and renders the problem more tractable. This abstraction would be appropriate for systems where the network is dimensioned to prevent the communication traffic from becoming too much of a bottleneck. A possible way to improve the

5. An advantage we found in using Z3 rather than Yices is that the API of the former is more advanced. In particular it provides a safe way to terminate a call.

6. In the execution platform model (Definition 5.2.1) we assume that all links are identical, which means that only the length of the routing path matters for the latency of communications.

	tasks	work	cp	ω	δ	$\frac{\delta}{\omega}$	β_2	β_1	platform/cost	time
0	10	78	49	8	24	3	1	2	(1,2,5)/48	1'
							2	2	(1,2,4)/47	2'
							1	3	(1,2,4)/47	2'
					32	4	1	2	(0,3,5)/29	1'
					32	4	2	2	(0,3,4)/28	4'
1	17	77	48	10	30	3	1	2	(0,2,4)/20	4'
							2	2	(0,2,4)/20	5'
					40	4	1	2	(0,1,6)/14	3'
2	21	136	65	20	40	2	1	1	(0,2,3)/19	3'
					60	3			(0,1,5)/13	5'
								2	(0,1,5)/13	6'
3	29	199	89	25	50	2	1	1	(1,2,2)/45	4'
								2	(1,2,2)/45	8'
4	35	187	63	40	80	2	1	1	(0,0,5)/5	6'
				30	60	2			(0,1,5)/13	5'
								2	?	$\perp$
5	40	210	56	35	70	2	1	1	(0,4,4)/(34,36)	11'
6	45	230	45	70	140	2	1	1	(0,0,4)/4	18'

Table 5.1: Results for the periodic deadline scheduling problem. The columns stand for: number of tasks in G, the quantity of work, the length of the critical path, the graph input period, the deadline, the maximum pipelining degree, the time and machine periodicities, the platform found and its cost and execution time. Platforms are written as $(R(1), R(2), \ldots)$. A pair (c_1, c_2) stands for a solution with cost c_2 whose optimality has not been proved, but for which the strict lower bound c_1 was found. The symbol $\perp$ means that no solution was ever found, i.e. a timeout occurred at the first call to the solver near the upper bound.

accuracy would be to introduce a parameter representing the global communication load (for instance the total amount of data which transists during the execution), and assume that data transfers are increasingly delayed as the network is heavily loaded.

In our experiments we were capable of approximating the Pareto front of up to 25-tasks graphs on the 8-spidergon architecture with reasonably small error (less than 2% for each objective) in a total computational time of a few minutes. Interestingly there was not a big overhead involved in moving to a bi-objective version of the problem. This may be due to the *learning* capability of modern satisfiability solvers which allow them to keep information about the problem through successive calls. Figure 5.3.8 shows the approximation obtained for a 20-tasks TGFF-generated graph on the 8-spidergon architecture. As one can see, the unsatisfiability information gives a good guarantee on the quality of the approximation returned.

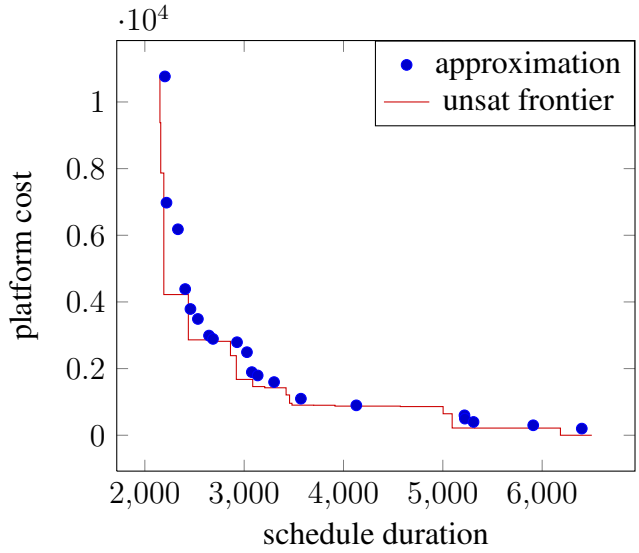


Figure 5.3.8: A 2% approximation of the Pareto front for a 20-tasks TGFF-generated task-graph (objectives functions have been rescaled to range in the same interval).

5.4 Scheduling with Stochastic Local Search

5.4.1 Introduction

We have used the stochastic local search algorithm presented in Chapter 3 to solve bi-criteria energy and deadline scheduling problems and we report in this section the implementation details and results of different experiments. We consider a multi-processor platform similar to Definition 5.2.1 but we assume that the speed of each processor is *fixed*.[7] In this context we employ a more refined measure of energy consumption which is defined by the sum of the contributions of each task and data transfer. The energy cost of a task $p \in G$ is computed from the speed of the processor on which it executes and its duration as:

$$C_1(p) = \alpha(s)\frac{w(p)}{s} \tag{5.4E}$$

where $s \in V$ is the speed of the processor where task p is mapped, $\alpha(s)$ is the unit cost of execution on a processor of speed s and $w(p)$ is the task workload. The cost of a data transfer depends on the length of the routing path and on the size of the data. Given two tasks $p \prec p'$ the energy cost $C_d(p, p')$ of their communication is

$$C_2(p, p') = n(b \cdot v(p, p') + l) \tag{5.4F}$$

where n is the number of hops between the processors on which tasks p and p' are mapped, b and l are respectively the bandwith and latency of links, and $v(p, p')$ is the size of the data. The bi-criteria optimization problem is then formulated as follows

$$\min(C, \delta)\ s.t.\ \textsc{sds}(G, E, \delta) \tag{5.4G}$$

7. A model where the speed of each processor can vary necessitates additional algorithmic development which could be done in future work.

with

$$C = \Sigma_p C_1(p) + \Sigma_{p,p'} C_2(p, p') \tag{5.4H}$$

Below we describe an SLS implementation for this problem.

5.4.2 Implementation

The critical part of the implementation of the local search method proposed in Section 3.5.1 is the function which steps from a schedule to one of its neighbors. This function must be optimized for the algorithm to perform more steps in a given amount of time. We start by defining the neighborhood and go on with a description of an algorithm for updating the schedule makespan incrementally.

As outlined in Definition 5.3.1, a schedule consists of two distinct parts: the mapping of tasks to processors and the start-times for execution. It is not feasible to perform local search directly on start-times which are real numbers. Hence, we move to an alternative representation of the schedule based on a *priority* relation $<$ defined on the tasks. The relation, which must be consistent with the partial order of the original task-graph G, specifies how to resolve resource conflicts happening on processors. A complete schedule is derived from this representation by starting the execution of each task as soon as all its predecessors (in terms of the original precedence) have terminated and not before its $<$ predecessor that execute on the same machine, as shown in Figure 5.4.1. Internally we store the priority relation in the form of a *permutation* implemented by a particular variant of a binary tree.

The local search algorithm proceeds by performing a topological sort of G to generate an initial permutation. A step then consists in changing the place of a task in the permutation while taking care that it remains consistent with the precedence relation of the graph. Let $\underline{p} < p < \overline{p}$ be three tasks such that $\underline{p}$ is the maximal element which precedes p in G and $\overline{p}$ is the minimal element which succeeds p in G. Because of precedence constraints, task p may only be moved somewhere between $\underline{p}$ and $\overline{p}$. For this reason, although an optimal step

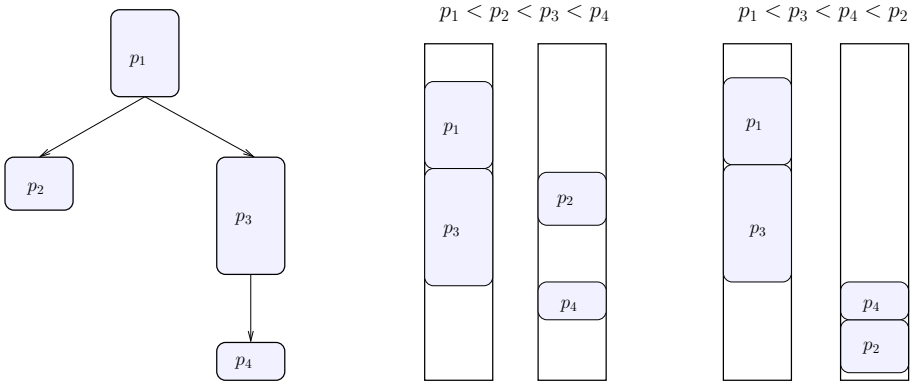


Figure 5.4.1: The picture shows two schedules of the same task-graph which are derived from two priority relations. We assume that tasks p_1 and p_3 are mapped to the first processor while p_2 and p_4 are mapped to the second. Because of the graph precedences, p_1 is necessarily executed before p_3. On the other hand there is a conflict between p_2 and p_4 for the second processor and this is resolved differently in both cases.

in this neighborhood is quadratic in the worst case, it generally requires less operations. Moves in the permutation are combined with changes in processor assignments. There are two options here: either take the union of the respective step types (in each step either change the permutation or change an assignment) or the cross-product (allow steps that change both simultaneously). We have chosen the latter option despite the fact that it increases the neighborhood size because it allows to move faster in the solution space.

A stochastic local search algorithm basically spends its time on evaluating neighboring solutions. Therefore, in order to achieve good performance, it is important that the cost functions are recomputed *incrementally* after each move. While this is easy for the energy cost (it is performed in constant time), we had to develop an incremental algorithm for updating the schedule makespan. This algorithm makes use of a graph G' equivalent to the task-graph G, but augmented with edges which assert the order imposed by the global task permutation on each processor. Each node in G' maintains the critical path towards the sink of the graph. In particular the source node holds the critical path of G', whose length is the schedule makespan. When a task p is moved in the permutation and/or in processor assignment, the critical path attributes must be updated. Three nodes of G' are directly affected by the change: the node of task p, the predecessor of p on the processor where it is currently mapped and the new predecessor, which may be on the same or another processor. The algorithm starts by updating the critical path of these nodes and then *recursively* propagates the changes through G' in reverse topological order. If the critical path of the current node has increased, the algorithm checks for each predecessor whether the path through task p has become critical. If it has decreased, it only updates the predecessors for which the critical path was going through p.

5.4.3 Experiments

We did experiments on a set of 8 TGFF-generated task-graphs of increasing sizes. The multi-processor platform on which graphs are scheduled has 8 processors and a communication network with a spidergon topology. The communication model is the same as in Section 5.3.6, which means that it only accounts for the latency of data transfers. As a measure of performance for our implementation, Table 5.2 reports the times needed by the program to carry out a locally optimal step for each graph when running on a Xeon 3.2GHz processor.

# task	10	15	20	25	30	35	40	45
step time (ms)	0.18	0.39	0.55	0.77	0.8	1.1	1.35	1.38

Table 5.2: Approximate processor time (in milliseconds) needed for the local search program to execute one optimal step.

The optimization program was run on each input graph and the set of trade-offs found saved at different points in time. The approximation quality is evaluated similar to experiments of Section 3.5.2 where we measure the dominated volume and compare it to a reference set of all the points obtained in any run. Results are found in Table 5.3 which shows the percentage of volume achieved after different numbers of steps of local search.

The results show that the local search algorithm is good at producing a decent approximation in quite a short time. All volume indicators are over 90 percent after 50,000 steps,

		# steps						
		1000	10,000	50,000	100,000	200,000	500,000	1,000,000
# tasks	10	0.93	0.98	1	1	1	1	1
	15	0.82	0.94	0.98	0.98	0.98	0.99	1
	20	0.88	0.95	0.97	0.97	0.98	0.99	1
	25	0.74	0.91	0.94	0.96	0.97	0.99	0.99
	30	0.67	0.79	0.93	0.94	0.97	0.98	0.99
	35	0.59	0.75	0.93	0.95	0.96	0.97	0.98
	40	0.7	0.87	0.9	0.92	0.93	0.95	0.97
	45	0.55	0.58	0.93	0.94	0.95	0.96	0.97

Table 5.3: Percentage of dominated volume. The reference set used for computing these values is the non-dominated set of points obtained by running the algorithm for 5,000,000 steps.

which takes only 1 minute of computation for the biggest graph of 45 tasks. Unlike the optimization method using SMT, there is no guarantee of closeness to the actual Pareto front. However we can say that the method rapidly approaches *its* best results: running the algorithm for 5 millions steps only leads to little improvement, especially when compared to the time invested for the search (Table 5.3).

Additionally, it seems that the algorithmic scheme for balancing across weight vectors we developed in Chapter 3 is efficient for the scheduling problem as well. This is illustrated on Figure 5.4.2, which shows approximations obtained for the graphs of 20 and 45 tasks. As one can see, the set of solutions obtained through these experiments are quite evenly spread over the cost space.

5.5 Discussion

We have formulated several design questions, involving cost optimization, scheduling, communication and pipelining, inspired by real problems in multi-core embedded systems. The methods we used to tackle these problems are really different in nature: the SMT-based optimization is exact but has limited scale, the stochastic local search algorithm is a fast heuristic with no garantee on the quality of the output. In the sequel we expose our conclusions on the use and applicability of these techniques.

Through our first experiments (Section 5.3), we demonstrated how modern solvers for constraint satisfaction can handle decent-size instances of multi-objective problems. However, the scalability appears quite poor for an industrial context, not to mention the fact that we used quite simple energy and communication models. Still, we think that our method could solve larger instances of the scheduling problem since there is much room for improvement in the implementation. So far we have considered the solver as a black-box and focused on the development of efficient search methods on top of it, but we did not look at the *inside*. Actually, because of their ability to efficiently solve large hardware and software verification problems, there is a growing interest in using SAT/SMT solvers for optimization, including the mapping and scheduling of applications on multi-processors [ZSJ10, RGH+10, LYH+08]. Because SAT solvers were initially designed for tackling *decision* problems, there is some room for tuning their internal mechanisms

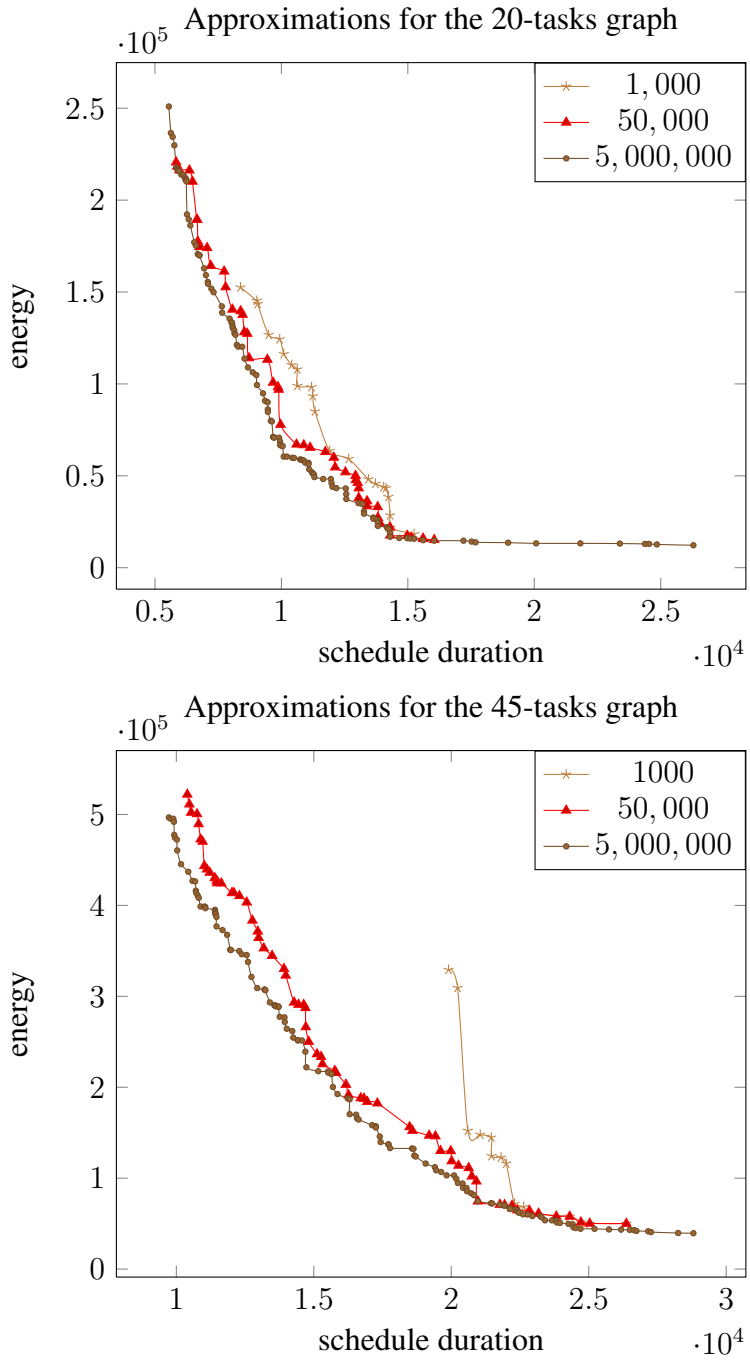


Figure 5.4.2: Approximations of the Pareto front for the graphs with 20 tasks and 45 tasks. The curves show the results of letting the algorithm run for 1,000, 50,000 and 5,000,000 steps.

towards better efficiency in performing optimization. There has been some recent work in this direction: Pseudo Boolean (PB) [ES06] or Max-SAT [HLO08] solvers natively support specific optimization problems defined as extensions of SAT. As another example, a recent work deals with the definition and implementation of a decision procedure for a "theory of costs" [CFG+10]. It investigates how the SMT solver may specifically handle the predicates encoding upper and lower bounds on boolean cost functions, rather than using the theory of reals as we do. Combining such developments with our multi-dimensional binary search algorithm could probably increase the size of scheduling problems we can treat.

On the other hand, we experimented stochastic local search which is a method of a really different flavor. We have designed and implemented an algorithm for tackling bi-criteria scheduling problems. Our experiments demonstrate that it is efficient in producing "good-looking" sets of non-dominated solutions in a really short time, but without a proven bound in quality. Thanks to the restart strategy for varying the weight vector that we have proposed in Chapter 3, we can provide an evenly-spread approximation in cost space. An important advantage of local search over the SMT-based method is that it scales well: this was confirmed experimentally on quadratic assignment and scheduling problems, since we could solve instances of significant size. The difficulties associated with SLS are essentially found in the design, implementation and tuning of an algorithm to step to a new solution while recomputing the costs efficiently. Efficient algorithms exist for some classical problems in combinatorial optimization such as traveling salesman, quadratic assignment and scheduling, but as we argued in Section 3.6, things get more complicated when the cost functions are multiple and when the decision variables are of different nature.

Conclusions

In this thesis we have developed, implemented and tested new computational techniques for solving hard multi-criteria optimization problems. The first class of techniques was motivated by the significant progress made in recent years in SAT and SMT solvers which provide powerful engines for solving hard mixed combinatorial-numerical constraint satisfaction problems. Starting from the straightforward idea of using such solvers for optimization by search, and being motivated by a practical energy cost vs. performance problem, we have discovered the world of multi-criteria optimization which resulted in our first major contribution, a search algorithm that directs the solver toward parts of the cost space that can improve the (bound on the) distance between the set of solutions and the actual Pareto front. We expect these techniques to improve in the future as solvers will be tuned for the needs of optimization.

The concepts, algorithms and data-structure developed in this work can be used in contexts other than multi-criteria optimization. Any bounded subset of $\mathbb{R}^n$ which is monotonic (upward closed or downward closed in each of the dimensions) admits a boundary between itself and its complement which verifies the properties of a Pareto front. Our algorithm can be used to approximate such surfaces using membership queries that need not be as hard as calls to an SMT solver. Such techniques can be useful, for example, for parameter-space exploration for dynamical systems where the oracle for a point p in the parameter space is a simulator that generates a trace that corresponds to p and checks whether or not it satisfies a property.

While the first contribution used solver technology as a given black box, the second contribution was more ambitious and produced a stand-alone solver based on stochastic local search. The idea of using the restart sequence $\mathcal{L}$ of [LSZ93] to sample evenly the space of scalarizations of a multi-dimensional cost function, first proposed to us by Scott Cotton, turned out to work well in practice, at least on the numerous examples to which it was applied. We believe this approach can find a respectable niche among the powerful heuristic techniques for solving multi-criteria optimization problems. More experience is needed in order to derive methodological guidelines for efficient encoding and exploration of local neighborhoods so that it becomes a routine not requiring a PhD to be performed.

These two techniques have been applied to a family of problems inspired by energy-aware scheduling. The experimental results give an idea on the size of such problems that can be solved by current technologies. Although size matters in optimization, performance is not sufficient (and not always necessary) for the usability of these techniques in the motivating domain, namely, deployment of parallel programs. Let us reflect in the remaining paragraphs about the gaps to be bridged between the abstract models that we have used, both for application programs and execution platforms, and the current practice.

Currently many applications are not described as clean task graphs but it looks like developers will have no choice but to convert them, this way or another, to more com-

ponentized descriptions in data-flow style if they want to run them on parallel machines. Profiling and past experience can be used to populate these models with the appropriate execution time figures. Applications that do not fit into the acyclic task-graph framework will need more effort to be modeled and be subject for optimization. On the architecture side, execution platforms are much more complex than the simple distributed memory *spidergon* architecture that we used in our experiments. They may involve a memory hierarchy, DMA engines and other features that increase the number of ways an application can be deployed on them. Moreover, the scheduling framework that we used for communication channels may not be realizable in NoC environments and a different type of modeling will be needed. In general, there is uncertainty in all aspects of the system performance including execution time, task arrival patterns, or communication load, which is not well handled by optimization methods. Other issues to think about are methodological: at what phases of the design will optimization methods intervene? Should optimal deployment be recomputed at every compilation step? Will it be postponed to the final phases or will it be too late then? How will it integrate in design-space exploration, before the architecture is fully realized. We leave some of these problems for future research.

Conclusions (French)

Dans cette thèse nous avons développé, implémenté et testé de nouvelles techniques pour résoudre des problèmes difficiles d'optimisation multi-critère. La première classe de techniques a été motivé par les progrès importants réalisés ces dernières années dans le domaine du SAT, avec des solveurs SMT qui sont maintenant capables de résoudre les problèmes de satisfaction de contraintes mixtes combinatoire-numérique. Partant de l'idée simple d'utiliser ces solveurs pour l'optimisation, et étant motivé par un problème pratique d'ordonnancement énergie/performance, nous avons découvert le monde de l'optimisation multi-critère, ce qui a abouti à notre première contribution : un algorithme de recherche qui dirige le solveur vers les parties de l'espace de coûts où l'on peut améliorer la distance entre l'ensemble des solutions trouvées et le vrai front de Pareto. Nous pensons que ces techniques fourniront des résultats encore meilleurs à l'avenir, car les solveurs SMT pourraient être spécialisés pour le traitement de problèmes d'optimisation.

Les concepts, les algorithmes et structures de données développés dans ce travail pourraient être utilisés dans d'autres contextes que l'optimisation multi-critère. Tout ensemble borné de $\mathbb{R}^n$ qui est monotone admet une limite entre lui-même et son complément qui vérifie les propriétés d'un front de Pareto. Notre algorithme peut être utilisé pour approximer de telles surfaces en utilisant des requêtes d'appartenance à l'ensemble qui ne sont pas obligées d'être aussi dures à résoudre que les appels à un solveur SMT. Ces techniques peuvent être utiles, par exemple, pour l'exploration de l'espace des paramètres d'un système dynamique, où l'oracle est un simulateur qui génère la trace obtenue avec certains paramètres et vérifie si une propriété est vérifiée.

Alors que notre première contribution est une technique de résolution qui utilise un solveur SMT comme boite noire, la deuxième contribution est plus ambitieuse et consiste en un solveur à part entière basé sur la recherche stochastique locale. L'idée d'utiliser la séquence de redémarrage $\mathcal{L}$ de [LSZ93] pour échantillonner de manière régulière l'espace des scalarizations d'une fonction de coût multi-dimensionnelle (suggéré par Scott Cotton), s'est montrée efficace en pratique, au moins sur les nombreux exemples que nous avons testé. Nous pensons que cette technique peut trouver une niche raisonnable parmis les nombreuses heuristiques d'optimisation multi-critère. Il y a toutefois un besoin de développer des méthodes et directives permettant de facilement encoder un algorithme de recherche locale, afin que cela ne soit pas réservé à une personne aguérie.

Les deux techniques proposées ont été appliquées à une famille de problèmes d'ordonnancement. Les résultats expérimentaux donnent une idée de la taille des problèmes qui peuvent être résolus par les technologies actuelles. Bien que la taille des problèmes que l'on peut traiter compte pour l'optimisation, ceci n'est pas suffisant pour l'utilisation de ces techniques dans le domaine d'application visé, à savoir le déploiement de programmes parallèles sur architecture multi-processeur. Nous mentionnons dans le paragraphe suivant les lacunes à combler entre les modèles abstraits que nous avons utilisés, tant pour les

applications que pour les plateformes d'exécution, et la pratique actuelle.

Actuellement de nombreuses applications ne sont pas décrites par des graphes de tâches, mais il semble que les développeurs n'auront d'autre choix que de les convertir, de cette façon ou d'une autre, en des descriptions plus modulaire si ils veulent les faire fonctionner sur des machines parallèles. Le profiling de l'application et l'expérience du programmeur peuvent servir pour alimenter ces modèles avec des temps d'exécution. En ce qui concerne l'architecture, les plateformes d'exécution sont beaucoup plus complexes que l'architecture simple à mémoire distribué *Spidergon* que nous avons utilisé dans nos expériences. Elles peuvent contenir une hiérarchie mémoire, des composants DMA, et avoir d'autres caractéristiques qui augmentent le nombre de façons de deployer une application. Par ailleurs le modèle de communication que nous avons choisit (ordonnancement des transferts de données sur les canaux) n'est pas forcément réaliste pour les systèmes basés sur un NoC. En général il y a aussi de l'incertitude sur tous les aspects du système (temps d'exécution et de communication, arrivée des tâches dans le système etc.), ce qui n'est pas bien traité par des méthodes d'optimisation. Enfin il faut aussi réfléchir à quelques problèmes de méthodologie : durant quelles phases de la conception les méthodes d'optimisation peuvent-elles intervenir? Le déploiement optimal doit-il être recalculé à chaque étape de compilation? Comment l'intégrer dans l'étape d'exploration du design de l'architecture? Ces problèmes sont autant de pistes pour des recherches futures.

Bibliography

[A+97] R.T. Azuma et al. A survey of augmented reality. *Presence-Teleoperators and Virtual Environments*, 6(4):355–385, 1997.

[ABC+06] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, et al. The landscape of parallel computing research: A view from berkeley. Technical report, Citeseer, 2006.

[ABD+09] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, et al. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[AC91] M. Dorigo et V. Maniezzo A. Colorni. Distributed optimization by ant colonies. In *First European Conference on Artificial Life*, pages 134–142. Elsevier, 1991.

[AGK+04] V. Arya, N. Garg, R. Khandekar, A. Meyerson, K. Munagala, and V. Pandit. Local search heuristics for k-median and facility location problems. *Siam Journal of Computing*, 33(3), 2004.

[Amd67] G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *spring joint computer conference*, pages 483–485. ACM, 1967.

[Art05] A. Artieri. Nomadik: an MPSoC solution for advanced multimedia. In *Application-Specific Multi-Processor SoC*, 2005.

[AVV08] J.E.C. Arroyo, P.S. Vieira, and D.S. Vianna. A GRASP algorithm for the multi-criteria minimum spanning tree problem. *Annals of Operations Research*, 159(1):125–133, 2008.

[BBTZ04] N.K. Bambha, S.S. Bhattacharyya, J. Teich, and E. Zitzler. Systematic integration of parameterized local search techniques in evolutionary algorithms. In *GECCO (2)*, pages 383–384, 2004.

[BDM02] L. Benini and G. De Micheli. Networks on chips: A new SoC paradigm. *IEEE Computer*, 35(1):70–78, 2002.

[BDZ10] J. Bader, K. Deb, and E. Zitzler. Faster hypervolume-based search using monte carlo sampling. *Multiple Criteria Decision Making for Sustainable Energy and Transportation Systems*, pages 313–326, 2010.

[BF09] K. Bringmann and T. Friedrich. Approximating the least hypervolume contributor: NP-Hard in general, but fast in practice. In *EMO*, pages 6–20, 2009.

[BF10] K. Bringmann and T. Friedrich. Approximating the volume of unions and intersections of high-dimensional geometric objects. *Comput. Geom.*, 43(6-7):601–610, 2010.

[BFLI+09] N. Beume, C.M. Fonseca, M. López-Ibáñez, L. Paquete, and J. Vahrenhold. On the complexity of computing the hypervolume indicator. *IEEE Trans. Evolutionary Computation*, 13(5):1075–1082, 2009.

[BJM+05] D. Bertozzi, A. Jalabert, S. Murali, R. Tamhankar, S. Stergiou, L. Benini, and G. De Micheli. Noc synthesis flow for customized domain specific multiprocessor systems-on-chip. *IEEE Trans. Parallel Distrib. Syst.*, 16(2):113–129, 2005.

[BKR91] R.E. Burkard, S. Karisch, and F. Rendl. QAPLIB-A quadratic assignment problem library. *European Journal of Operational Research*, 55(1):115–119, 1991.

[BLPN01] P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-based scheduling: Applying constraint programming to scheduling problems*. Springer, 2001.

[Bor07] S. Borkar. Thousand core chips: A technology perspective. In *DAC*, pages 746–749, 2007.

[BSMD08] S. Bandyopadhyay, S. Saha, U. Maulik, and K. Deb. A simulated annealing-based multiobjective optimization algorithm: AMOSA. *IEEE Trans. Evolutionary Computation*, 12(3):269–283, 2008.

[BSST09] C.W. Barrett, R. Sebastiani, S.A. Seshia, and C. Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 825–885. 2009.

[BWH+03] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *IEEE Computer*, 36(4):45–52, 2003.

[BZ11] J. Bader and E. Zitzler. Hype: An algorithm for fast hypervolume-based many-objective optimization. *Evolutionary Computation*, 19(1):45–76, 2011.

[CBL+10] F. Clermidy, C. Bernard, R. Lemaire, J. Martin, I. Miro-Panades, Y. Thonnart, P. Vivet, and N. Wehn. MAGALI: A network-on-chip based multi-core system-on-chip for MIMO 4G SDR. In *IEEE IC Design and Technology (ICICDT)*, pages 74–77, 2010.

[CFG+10] A. Cimatti, A. Franzén, A. Griggio, R. Sebastiani, and C. Stenico. Satisfiability modulo the theory of costs: Foundations and applications. *TACAS*, pages 99–113, 2010.

[CJ98] P. Czyzak and A. Jaszkiewicz. Pareto simulated annealing-a metaheuristic technique for multiple-objective combinatorial optimization. *Journal of Multi-Criteria Decision Analysis*, 7(1):34–47, 1998.

[CJGJ96] E.G. Coffman Jr, M.R. Garey, and D.S. Johnson. Approximation algorithms for bin packing: A survey. In *Approximation algorithms for NP-hard problems*, pages 46–93. PWS Publishing Co., 1996.

[CL98] F. Y. Cheng and D. Li. Genetic algorithm development for multiobjective optimization of structures. *AIAA Journal*, 36(6):1105–1112, 1998.

[CLM+04] M. Coppola, R. Locatelli, G. Maruccia, L. Pieralisi, and A. Scandurra. Spidergon: A novel on-chip communication network. In *IEEE Symposium on System-on-Chip, 2004.*, page 15, 2004.

[Coh85] J.L. Cohon. *Multicriteria programming: Brief review and application*. New York, NY, Academic Press, 1985.

[DD98] I. Das and J.E. Dennis. Normal-boundary intersection: A new method for generating the pareto surface in nonlinear multicriteria optimization problems. *SIAM Journal on Optimization*, 8:631, 1998.

[DdM06] B. Dutertre and L.M. de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV*, pages 81–94, 2006.

[Deb01] K. Deb. *Multi-objective optimization using evolutionary algorithms*. Wiley, 2001.

[DJR01] S. Dutta, R. Jensen, and A. Rieckmann. Viper: A multiprocessor SoC for advanced set-top box and digital TV systems. *IEEE Design & Test of Computers*, 18(5):21–31, 2001.

[DLL62] M. Davis, G. Logemann, and D.W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[DMB08] L.M. De Moura and N. Bjorner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[DRV00] A. Darte, Y. Robert, and F. Vivien. *Scheduling and automatic parallelization*. Birkhäuser, 2000.

[DRW98] R.P. Dick, D.L. Rhodes, and W. Wolf. TGFF: Task graphs for free. In *CODES*, pages 97–101, 1998.

[ECEP06] C. Erbas, S. Cerav-Erbas, and A.D. Pimentel. Multiobjective optimization and evolutionary algorithms for the application mapping problem in multiprocessor system-on-chip design. *IEEE Trans. Evolutionary Computation*, 10(3):358–374, 2006.

[EG00] M. Ehrgott and X. Gandibleux. A survey and annotated bibliography of multiobjective combinatorial optimization. *OR Spectrum*, 22(4):425–460, 2000.

[Ehr05] M. Ehrgott. *Multicriteria optimization*. Springer Verlag, 2005.

[EPTP07] C. Erbas, A.D. Pimentel, M. Thompson, and S. Polstra. A framework for system-level modeling and simulation of embedded systems architectures. *EURASIP Journal on Embedded Systems*, 2007(1):2–2, 2007.

[Erb06] C. Erbas. *System-level modelling and design space exploration for multiprocessor embedded system-on-chip architectures*. Amsterdam University Press, 2006.

[ES06] N. Eén and N. Sörensson. Translating pseudo-boolean constraints into SAT. *JSAT*, 2(1-4):1–26, 2006.

[Fet09] B.A. Fette. *Cognitive radio technology*. Academic Press, 2009.

[FF96] C.M. Fonseca and P.J. Fleming. On the performance assessment and comparison of stochastic multiobjective optimizers. In *PPSN*, pages 584–593, 1996.

[FGE05] J. Figueira, S. Greco, and M. Ehrgott. *Multiple criteria decision analysis: state of the art surveys*. Springer Verlag, 2005.

[Fle03] M. Fleischer. The measure of pareto optima. In *EMO*, pages 519–533, 2003.

[FR95] T.A. Feo and M.G.C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6(2):109–133, 1995.

[FWB07] X. Fan, W.D. Weber, and L.A. Barroso. Power provisioning for a warehouse-sized computer. In *ISCA*, pages 13–23, 2007.

[GHN+04] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *CAV*, pages 293–295. Springer, 2004.

[GM06] F. Glover and R. Marti. Tabu search. *Metaheuristic Procedures for Training Neutral Networks*, pages 53–69, 2006.

[GMF97] X. Gandibleux, N. Mezdaoui, and A. Fréville. A tabu search procedure to solve multiobjective combinatorial optimization problem. *Lecture notes in economics and mathematical systems*, pages 291–300, 1997.

[GMNR06] S. Gochman, A. Mendelson, A. Naveh, and E. Rotem. Introduction to Intel core duo processor architecture. *Intel Technology Journal*, 10(2):89–97, 2006.

[Han97] M.P. Hansen. Tabu search for multiobjective optimization: MOTS. In *MCDM*, pages 574–586, 1997.

[Hel00] K. Helsgaun. An effective implementation of the Lin-Kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000.

[HLO08] F. Heras, J. Larrosa, and A. Oliveras. Minimaxsat: An efficient weighted max-sat solver. *Journal of Artificial Intelligence Research*, 31(1):1–32, 2008.

[HLW71] YY Haimes, LS Lasdon, and DA Wismer. On a bicriterion formulation of the problems of integrated system identification and system optimization. *IEEE Transactions on Systems, Man, and Cybernetics*, 1(3):296–297, 1971.

[Hoo99] H.H. Hoos. On the run-time behaviour of stochastic local search algorithms for SAT. In *AAAI/IAAI*, pages 661–666, 1999.

[HS04] H. Hoos and T. Stützle. *Stochastic Local Search Foundations and Applications*. Morgan Kaufmann / Elsevier, 2004.

[JM94] J. Jaffar and M.J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19:503–581, 1994.

[JTW05] A.A. Jerraya, H. Tenhunen, and W. Wolf. Guest editors' introduction: Multiprocessor systems-on-chips. *IEEE Computer*, 38(7):36–40, 2005.

[KB57] T.C. Koopmans and M. Beckman. Assignment problems and the location of economic activities. *Econometric*, 25:53–76, 1957.

[KC00] J.D. Knowles and D. Corne. Approximating the nondominated front using the pareto archived evolution strategy. *Evolutionary Computation*, 8(2):149–172, 2000.

[KC03] J.D. Knowles and D. Corne. Instance generators and test suites for the multiobjective quadratic assignment problem. In *EMO*, pages 295–310, 2003.

[KC05] J.D. Knowles and D. Corne. Memetic algorithms for multiobjective optimization: issues, methods and prospects. *Recent advances in memetic algorithms*, pages 313–352, 2005.

[KDH+05] J.A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM journal of Research and Development*, 49(4.5):589–604, 2005.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, New Series 220 (4598):671–680, 1983.

[Kno05] J.D. Knowles. A summary-attainment-surface plotting method for visualizing the performance of stochastic multiobjective optimizers. In *ISDA*, pages 552–557, 2005.

[KNRSV00] K. Keutzer, A.R. Newton, J.M. Rabaey, and A.L. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 19(12):1523–1543, 2000.

[LLGCM10] J. Legriel, C. Le Guernic, S. Cotton, and O. Maler. Approximating the pareto front of multi-criteria optimization problems. In *TACAS*, pages 69–83, 2010.

[LM10] J. Legriel and O. Maler. Meeting deadlines cheaply. Technical Report 2010-1, VERIMAG, January 2010.

[LMS03] H. Lourenco, O. Martin, and T. Stützle. Iterated local search. *Handbook of metaheuristics*, pages 320–353, 2003.

[LSZ93] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of las vegas algorithms. In *ISTCS*, pages 128–133, 1993.

[LYH+08] W. Liu, M. Yuan, X. He, Z. Gu, and X. Liu. Efficient SAT-based mapping and scheduling of homogeneous synchronous dataflow graphs for throughput optimization. In *RTSS*, pages 492–504, 2008.

[Mat65] J. Matyas. Random optimization. *Automation and Remote Control*, 2(26):246–253, 1965.

[Meu08] H.W. Meuer. The top500 project: Looking back over 15 years of supercomputing experience. *Informatik-Spektrum*, 31(3):203–222, 2008.

[Mit98] M. Mitchell. *An introduction to genetic algorithms*. The MIT press, 1998.

[MOP+09] R. Marculescu, U.Y. Ogras, L.S. Peh, N.E Jerger, and Y.V. Hoskote. Outstanding research problems in noc design: System, microarchitecture, and circuit perspectives. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 28(1):3–21, 2009.

[NO06] R. Nieuwenhuis and A. Oliveras. On SAT modulo theories and optimization problems. In *SAT*, pages 156–169, 2006.

[Par12] V. Pareto. Manuel d'économie politique. *Bull. Amer. Math. Soc.*, 18:462–474, 1912.

[PD07] K. Pipatsrisawat and A. Darwiche. A Lightweight Component Caching Scheme for Satisfiability Solvers. In *SAT-07*, 2007.

[PS03] L. Paquete and T. Stützle. A two-phase local search for the biobjective traveling salesman problem. In *EMO*, pages 479–493, 2003.

[PS06a] L. Paquete and T. Stützle. Stochastic local search algorithms for multiobjective combinatorial optimization: A review. Technical Report TR/IRIDIA/2006-001, IRIDIA, 2006.

[PS06b] L. Paquete and T. Stützle. A study of stochastic local search algorithms for the biobjective QAP with correlated flow matrices. *European Journal of Operational Research*, 169(3):943 – 959, 2006.

[PY00] C.H. Papadimitriou and M. Yannakakis. On the approximability of trade-offs and optimal access of web sources. In *FOCS*, pages 86–92, 2000.

[RGH+10] F. Reimann, M. Glaß, C. Haubelt, M. Eberl, and J. Teich. Improving platform-based system synthesis by satisfiability modulo theories solving. In *CODES+ISSS*, pages 135–144, 2010.

[Sch99] Alexander Schrijver. *Theory of linear and integer programming*. Wiley-Interscience series in discrete mathematics and optimization. Wiley, 1999.

[SG76] S. Sahni and T. Gonzalez. P-complete approximation problems. *Journal of the ACM*, 23:555–565, 1976.

[SKC93] B. Selman, H. Kautz, and B. Cohen. Local Search Strategies for Satisfiability Testing. In *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, 1993.

[SKC94] B. Selman, H.A. Kautz, and B. Cohen. Noise strategies for improving local search. In *AAAI*, pages 337–343, 1994.

[Ste86] R.E. Steuer. *Multiple criteria optimization: Theory, computation, and application*. John Wiley & Sons, 1986.

[Tai91] E. Taillard. Robust taboo search for the quadratic assignment problem. *Parallel computing*, 17(4-5):443–455, 1991.

[TBHH07] L. Thiele, I. Bacivarov, W. Haid, and K. Huang. Mapping applications to tiled multiprocessor embedded systems. In *ACSD*, pages 29–40, 2007.

[TBS+10] L. Torres, P. Benoit, G. Sassatelli, M. Robert, F. Clermidy, and D. Puschini. An introduction to multi–core system on chip–trends and challenges. *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, page 1, 2010.

[TH04] D.D. Tompkins and H.H. Hoos. UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT & Max-SAT. In *SAT*, 2004.

[UTFT99] E.L. Ulungu, J. Teghem, P.H. Fortemps, and D. Tuyttens. MOSA method: a tool for solving multiobjective combinatorial optimization problems. *Journal of Multi-Criteria Decision Analysis*, 8(4):221–236, 1999.

[VA04] D.S. Vianna and J.E.C. Arroyo. A GRASP algorithm for the multi-objective knapsack problem. In *SCCC*, pages 69–75, 2004.

[WJM08] W. Wolf, A.A. Jerraya, and G. Martin. Multiprocessor system-on-chip (MPSoC) technology. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(10):1701–1713, 2008.

[WM95] W.A. Wulf and S.A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23:20–20, 1995.

[Wol07] W. Wolf. *High-performance embedded computing: architectures, applications, and methodologies*. Morgan Kaufmann, 2007.

[ZKT08] E. Zitzler, J.D. Knowles, and L. Thiele. Quality assessment of Pareto set approximations. In *Multiobjective Optimization*, pages 373–404, 2008.

[ZM02] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *CAV*, pages 17–36, 2002.

[ZSJ10] J. Zhu, I. Sander, and A. Jantsch. Constrained global scheduling of streaming applications on mpsocs. In *ASP-DAC*, pages 223–228, 2010.

[ZT98] E. Zitzler and L. Thiele. Multiobjective optimization using evolutionary algorithms - a comparative case study. In *PPSN*, pages 292–304, 1998.

[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. *IEEE Trans. Evolutionary Computation*, 7(2):117–132, 2003.

TITLE

Multi-Criteria Optimization and its Application to Multi-Processor Embedded Systems

ISBN : □□□□□□□□□□□□□□□