Architectures reconfigurables et traitement de problèmes NP-difficiles : un nouveau domaine d'application

Youssef Hamadi[*] — **David Merceron**[**]

[*] *LIRMM, UMR 5506 CNRS/Université Montpellier II*
161, Rue Ada, 34392 Montpellier Cedex 5
hamadi@lirmm.fr

[**] *EURIWARE, 12-14 rue du fort de St-Cyr*
78067 St Quentin-en-Yvelines Cedex
damercer@euriware.fr

RÉSUMÉ. *L'algorithme GSAT est un algorithme de recherche locale. Cette méthode recherche la première instanciation satisfaisable de formules logiques de forme normale conjonctive. Bien que de nature incomplète son exploration fine de l'espace de recherche associée à l'utilisation d'heuristiques puissantes lui a permis de résoudre des problèmes encore inaccessibles en recherche exhaustive. De plus, le caractère générique de ses formules d'entrées lui a ouvert de larges domaines d'applications (synthèse et test de circuits, planification de tâches, ordonnancement, vision, etc. . .). Dans cette étude, nous présentons une implémentation de cette méthode sur architecture reconfigurable de type FPGA [XIL 91]. Nous poursuivons par là le double but de permettre le traitement rapide de très gros problèmes SAT et d'autoriser un traitement de type temps réel pour les instances de tailles plus réduites. Dans ce travail, la flexibilité de ce type d'architecture est donc utilisée pour résoudre efficacement des problèmes SAT.*

ABSTRACT. *GSAT is a greedy local search procedure. It searches for satisfiable instanciations of formulas under conjunctive normal form. Intrinsically incomplete, this algorithm has shown its ability to deal with formulas of large size that are not yet accessible to exhaustive methods. Many problems such as circuits synthesis and test, planning, scheduling, vision can efficiently be solved by using the GSAT algorithm. In this study, we give an implementation of GSAT on Field Programmable Gate Arrays (FPGA) [XIL 91] in order to speed-up the resolution of SAT problems. By this implementation, our aim is to reach very large SAT problems and to enable real-time resolution for current size problems. The FPGA technology allows users to adapt a generic logic chip to different tasks. In the framework of SAT problems we show how to quickly adapt our chips to efficiently solve satisfiability problems.*

MOTS-CLÉS : *Architectures reconfigurables, FPGA, Problèmes SAT, Méthodes de recherche locale*

KEY WORDS : *Reconfigurable hardware, FPGA, Problems SAT, Local search procedures*

1. Introduction

La méthode de recherche locale GSAT a été présentée par B. Selman [SEL 92]. Elle permet d'atteindre un état satisfaisant l'ensemble ou un sous-ensemble des contraintes d'un problème en appliquant des réparations successives à une instanciation de départ des variables. Cette méthode a démontré son efficacité par le traitement de problèmes de grandes tailles qui restent généralement inaccessibles aux méthodes classiques de type exploration arborescente. De plus, confronté à d'autres procédures de recherche locale, GSAT a souvent obtenu les meilleurs résultats [BEE 94], [HOO 96]. Les performances de cet algorithme et la variété de ses domaines d'application ont suscité notre intérêt pour son adaptation sur architecture reconfigurable à base de FPGA (Field Programmable Gate Array) [XIL 91]. L'implémentation d'une méthode de recherche locale tel que GSAT sur FPGA doit permettre d'une part un traitement rapide d'instances de plus grandes tailles, d'autre part d'offrir une échelle de résolution temps réel aux problèmes de tailles plus modestes. Ces derniers problèmes ont souvent un caractère réactif dans l'exécution (exemple : systèmes de vision artificielle pour robot mobile [WAL 75]) justifiant un traitement rapide. Le changement d'échelle dans les problèmes considérés peut bénéficier à l'immense champ d'application de la procédure : problèmes de satisfaction (SAT) et de façon plus générale aux problèmes d'optimisation (MAX-SAT). Ces problèmes interviennent dans différents sous-domaines de l'intelligence artificielle et concernent notamment certains problèmes clefs tels que la synthèse de circuits [KAM 92], le test [LAR 90] la planification [STE 81], l'ordonnancement, la vision [WAL 75], etc. . . Pour faire face à la complexité de tels problèmes, la solution la plus courante consiste à distribuer et/ou paralléliser l'exploration [RAO 93] [HAM 98] [HAM 96] [HAM 99] ou de façon plus marginale à employer simultanément diverses méthodes dans un cadre coopératif [HOG 93].

L'idée d'utiliser une implémentation matérielle pour accélérer la résolution de problèmes NP-difficiles est assez neuve. Dans le cadre de la satisfaction de contraintes (CSP), P.R. Cooper et al. ont utilisé la puissance du matériel [SWA 88], [COO 92]. Cependant leur approche est très spécifique et consiste en la réalisation d'un circuit (ASIC) représentant une classe de problèmes. De plus, ils ne recherchent pas de solution mais se contentent d'une réduction de l'espace de recherche. Ce travail a été étendu par H.W. Guesguen [GUE 91] mais ne permet toujours pas la recherche de solution. La première implémentation sur architecture reconfigurable d'une méthode de résolution de problème de satisfaction a été présentée par M. Yokoo et al. [YOK 96]. Cependant ce travail s'intéresse à une procédure de recherche complète dont la complexité structurelle ruine les performances en cas d'implémentation physique. Il apparaît donc que l'application de la puissance du matériel à la résolution de problèmes de type SAT et MAX-SAT centraux en IA suscite un intérêt croissant. Cela est généralement dû au fait que le champ d'application des techniques d'IA se voyant élargi aux problèmes réels, les implémentations sur architecture classique butent très vite sur la complexité des espaces explorés. Nous offrons ici une solution à ce problème en adaptant sur architecture reconfigurable une méthode performante d'optimisation combinatoire. Nous verrons comment le caractère flexible de ces architectures nous permet d'exploiter efficace-

ment la structure des problèmes traités. Les FPGAs contrairement aux ASICs offrent la reconfiguration nécessaire à la mise en place de diverses heuristiques. Nous verrons ainsi que la reconfiguration de l'algorithme matériel est simplement réalisable par la mise en place d'heuristiques developpées pour la version logicielle de GSAT.

Après une brève présentation de la recherche locale, nous détaillons l'algorithme GSAT. Nous introduisons ensuite les architectures matérielles reconfigurables. S'en suit une présentation détaillée des différents composants de notre implantation physique de GSAT. Nous montrons ensuite comment augmenter sensiblement l'efficacité de notre design par l'ajout de certaines heuristiques développées spécifiquement pour la méthode logicielle. Finalement nous analysons en détail les performances attendues de notre implantation sur différentes instances de problèmes difficiles et nous discutons son architecture.

2. La recherche locale

Dans de nombreuses applications, la vitesse de résolution des problèmes domine d'autres facteurs tels que l'exactitude des solutions générées. Généralement ces applications associent dynamicité et grands espaces de recherche. La dynamicité fait que les contraintes du problème évoluent très vite rendant toute solution optimum vite obsolète, la taille des espaces rencontrés fait que la résolution par méthodes classiques quand elle s'avère possible est souvent trop lente. Les méthodes de recherche locale qui sacrifient complétude contre rapidité sont donc idéales pour ces problèmes. Nous présentons ici brièvement cette catégorie de méthodes en mettant à jour leurs caractéristiques communes.

La définition d'une méthode de recherche locale nécessite la définition de trois fonctions. La première associe une qualité (resp. un coût) à chaque point de l'espace de recherche $\mathcal{E}$, la seconde associe chaque point de cet espace à un ensemble de points voisins, enfin la dernière fonction autorise un choix dans cet ensemble de voisins. L'on parle respectivement de fonction d'évaluation (parfois appelée fonction de qualité (resp. coût)), de fonction de voisinage ou d'adjacence et de fonction de transition.

Selon les problèmes traités la qualité pourra se définir en considérant le degré de satisfaction qu'offre chaque point de l'espace de recherche. Par exemple, nombre de contraintes satisfaites dans le cas de problèmes utilisant le formalisme CSP, nombre de clauses satisfaites pour des problèmes SAT/MAX-SAT. Une solution est un point de l'espace de qualité (resp. coût) maximum (resp. minimum). Le voisinage d'un point constitue le futur de la recherche, il sera évalué par la fonction de transition. De sa taille dépendra à la fois la complexité d'exécution d'une transition et la qualité du nouvel état. La fonction de transition implique la définition d'un critère de choix entre un point et ses voisins de l'espace de recherche. Ce choix nécessite au minimum l'évaluation de chaque point du voisinage. Cependant des affinements sont possibles tels que le maintien d'un certain historique destiné à éviter la redécouverte des mêmes états (voir [GLO 89]). Lors d'une transition plusieurs voisins peuvent fournir la même qualité (resp. coût), ces égalités sont alors 'cassées' de manière aléatoire.

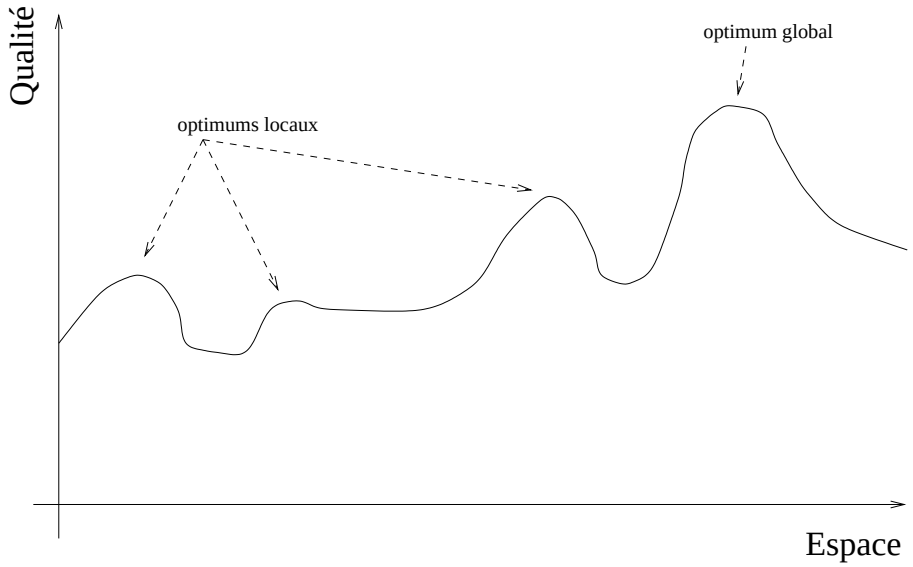


Figure 1. *Fonction de qualité et recherche locale*

Une méthode de recherche locale commence par générer (de façon plus ou moins aléatoire) un point dans l'espace $\mathcal{E}$, le voisinage de ce point est ensuite évalué grâce à la fonction de transition, décidant ainsi du nouvel état. Naturellement une transition s'effectue si le voisinage contient un point de qualité (resp. coût) supérieure (resp. inférieur). Un tel point peut ne pas exister : la méthode a alors atteint un optimum.

Si cet optimum est global, nous sommes alors en présence d'une solution au problème (voir figure 1). Dans le cas d'un optimum local, différents mécanismes sont utilisés pour faire sortir la recherche de cette impasse. Ces mécanismes accentuent encore les différences entre algorithmes. Les plus courants consistent à relancer plusieurs fois la recherche à partir d'un nouveau point de l'espace ou encore à tolérer une dégradation de la fonction d'évaluation (c'est-à-dire acceptation de pertes (resp. hausses) de qualité (resp. coût)).

3. La méthode GSAT

Présentée en 1992 par B. Selman, GSAT est une méthode de recherche locale (voir algo. 1). Partant du principe qu'une affectation totale des variables du problème à résoudre est plus informatif qu'une affectation partielle successivement étendue, l'algorithme démarre en générant une instanciation arbitraire des variables et essaie ensuite par une série de réparations d'atteindre un état offrant le plus haut degré de satisfaction. C'est-à-dire soit une solution au problème identifiable par la satisfaction de l'ensemble des contraintes de départ, soit une satisfaction partielle dont la qualité est estimable par la mesure des contraintes satisfaites.

Le format d'entrée de la méthode (voir algorithme 1) utilise la logique des propositions. Une formule sous forme normale conjonctive [1] (CNF) est utilisée pour coder le problème à résoudre. Cette forme logique peut être obtenue à partir d'une représentation de haut niveau par le biais de traducteurs automatiques [BOT 92].

Les réparations de l'affectation initiale s'obtiennent donc en changeant la valeur d'une des variables représentant le problème. Les variables logiques sont ici des variables booléennes (ordre 0). Le changement d'instanciation d'une variable consistera donc en sa complémentation et constituera une réparation de l'affectation. Par souci de cohérence avec la littérature classique sur GSAT, nous parlerons alors de flip [2] de variable pour désigner l'opération de complémenter. Le voisinage d'un état est donc constitué de l'ensemble des points différents dans l'instanciation d'une variable propositionnelle.

Algorithme 1: La méthode de recherche locale GSAT

```
GSAT(A, Max-essais, Max-flips)
Entrée : A : formule logique sous forme normale conjonctive
         Max − essais, Max − flips : Entier
Sortie : L'instanciation des variables de A offrant le degré maximum de satisfaction
début
    pour (i = 1 à Max-essais) faire
        V ← instanciation initiale des variables
        pour (j = 1 à Max-flips) faire
            si A satisfaisable par V alors
                retourner V
            sinon
                p ← la variable dont le flip offre la plus forte
                augmentation du nombre de clauses satisfaites
                V ← V avec p flippée
    retourner La meilleure instanciation rencontrée
fin
```

Naturellement, le flip incontrôlé des variables du problème ne pourrait suffire à assurer la convergence de la méthode. Cette convergence est contrôlée par l'affectation d'un score (i.e. qualité) à l'instanciation de chaque variable propositionnelle. Cette valeur représente le nombre de clauses satisfaites par cette instanciation. Lors d'une réparation (i.e. transition), l'algorithme recherche la variable dont le changement d'instanciation (flip) offrirait la plus forte hausse de score.

1. Conjonction de clauses, une clause étant une disjonction de littéraux, un littéral étant constitué par une variable ou par sa négation.

2. Le terme de perturbation sera considéré comme équivalent par la suite.

Cette recherche nécessite de complémenter les variables une à une en relevant l'effet de ce changement sur la satisfaction globale des clauses. Le changement d'instanciation de la variable offrant la plus forte hausse de satisfaction, permet d'atteindre une nouvelle affectation. Si la satisfaction n'est pas optimale (il existe une clause insatisfaite), cette affectation sert de base à une nouvelle réparation.

Les égalités pouvant survenir dans la hausse de satisfaction autorisée par différentes variables sont "cassées" de façon aléatoire. Cette manière de procéder est commune à l'ensemble des méthodes de recherche locale et cela même lorsque différents niveaux d'heuristiques sont utilisés. D'une façon générale, cette part d'indéterminisme est de plus en plus recherchée dans ce type de méthodes [SEL 93a], [SEL 94].

L'exécution est controlée par deux paramètres fournis par l'utilisateur :

— Max-essais : cet entier représente le nombre maximum de tentatives que l'on s'autorise pour parvenir à la satisfaction globale du problème. Chaque tentative démarre avec un nouvelle affectation aléatoire des variables.

— Max-flips : un nombre de réparations est possible à partir de chaque affectation initiale. Cet entier représente le nombre de ces réparations. Un petit multiple du nombre de variables du problème (dans sa représentation normale conjonctive) est généralement utilisé.

La multiplication des deux valeurs précédentes représente donc le nombre d'états que la méthode est en droit d'explorer à la recherche d'un modèle des variables (affectation satisfaisant l'ensemble des clauses). Cela correspond à une valeur limite de l'exploration. Cependant, si la recherche d'un modèle se révèle infructueuse, le meilleur sous-modèle (i.e. la meilleure interprétation), c'est-à-dire l'instanciation des variables de meilleur score cumulé, est retourné par le programme.

Les définitions suivantes seront utilisées tout au long de ce travail pour faciliter le rapprochement entre l'implantation matérielle et logicielle de l'algorithme.

DÉFINITION 3.1 (ESSAI)
Un essai est constitué par l'exécution de la boucle externe de l'algorithme 1. Cela correspond à une tentative de résolution à partir d'une configuration de départ. Ce mécanisme d'essais successifs constitue en fait un système d'extraction des minimums locaux. □

DÉFINITION 3.2 (FLIP)
Un flip correspond à l'exécution de la boucle interne de la procédure GSAT. Cela correspond donc à la recherche de la variable offrant la plus forte hausse de score et à sa complémentation. □

L'utilisateur a donc la possibilité de borner le nombre total de perturbations. Une implémentation incrémentale génère au démarrage le score associé à chaque variable, chaque perturbation décidant ensuite du flip d'une variable propositionnelle p. Cette variable ainsi que l'ensemble des variables présentes dans les clauses où intervient p voient leurs scores reconsidérés. La perturbation d'une affectation des variables nécessite donc uniquement la mise à jour du score d'une partie des variables. Sur la plupart des problèmes aléatoires servant de base à l'évaluation de ce type de méthodes

[SEL 93b], le voisinage d'une variable, c'est-à-dire l'ensemble des variables intervenant dans les mêmes clauses, est relativement restreint. Cette taille réduite autorise les excellentes performances de la méthode qui effectue des perturbations en temps constant.

Nous entrevoyons donc ici une partie de la difficulté de notre tâche : réaliser une implémentation physique offrant de meilleures performances qu'un algorithme incrémental. Nous verrons que l'incrémentalité, si elle se justifie dans le cas d'une implémentation sur machine classique, est souvent trop coûteuse dans le cas d'une implantation physique.

4. Matériel reconfigurable

Afin d'accélérer les processus nécessaires à notre approche, nous nous sommes intéressés à l'utilisation de co-processeurs reconfigurables. Un co-processeur a pour but d'accélérer des tâches spécifiques pour lesquelles le processeur principal, du fait de sa généricité, est peu performant. Pratiquement, on adjoint un composant particulier, dédié le plus souvent à une classe de fonctions, et qui communique de manière privilégiée avec le processeur. Cette approche comporte cependant une limite résidant essentiellement dans le nombre de co-processeurs qu'il est techniquement possible d'adjoindre au processeur, et dans la difficulté que l'on peut avoir à trouver un composant répondant exactement au problème que l'on veut résoudre. La logique reconfigurable offre une alternative attrayante : elle se base sur la programmation dynamique de portes logiques, permettant ainsi de construire une infinité d'architectures sur un même support physique. Le couplage d'un tel composant avec un processeur standard permet à ce dernier de disposer d'un co-processeur adapté à la fonction à accélérer. Depuis l'introduction du concept de logique reconfigurable, plusieurs générations de composants se sont succédés, se répartissant en diverses familles afin de couvrir différentes demandes.

Nous introduisons ici les Field Programmable Gate Array à mémoire statique (mémoire SRAM) basés sur des tables de vérités (LUT pour Look-Up Table). La programmation de ce type de cartes consiste en une implantation des différentes parties opératoires, cette implantation devant tenir compte des interconnexions nécessaires entre ces parties. La technologie doit donc permettre une infinité de reconfigurations avec un grain de configuration situé au niveau de la porte logique.

Nous explicitons principalement ici les FPGA de la famille Xilinx [XIL 91], hormis la série Xc6000. Pour l'utilisateur, l'architecture globale des FPGA Xilinx est composée de trois éléments de base :

— CLB : Bloc logique configurable. Composé de 2 tables de vérités de 2^4 bits chacun (LUT). Un LUT permet d'implémenter n'importe quelle fonction booléenne avec 4 variables en entrée. Ces deux LUT peuvent être interconnectés afin de ne former qu'un seul LUT à 5 entrées. De plus, chaque CLB dispose de deux flip-flops [3], permettant de mémoriser des signaux logiques (1 bit chaque).

3. Un flip-flop est une mémoire qui stocke un bit jusqu'au coup d'horloge suivant. Ceci permet de synchroniser différentes parties du design.

— IOBs : Blocs d'entrées sorties. Ils permettent la communication entre le FPGA et le reste de la carte (son voisinage).

— Ressources d'interconnexion : connexions directionnelles locales et globales.

Chaque type de FPGA a un nombre de CLB spécifique. Par exemple, un Xc3090 dispose de 320 CLBs, tandis qu'un Xc4062 comporte 2304 CLBs. Afin d'augmenter le nombre de CLB, il est possible de coupler plusieurs FPGA sur la même carte.

5. Reconfiguration dynamique

La reconfiguration dynamique d'un FPGA augmente la densité et/ou la vitesse d'un design. Cette technique est utile lorsque l'on veut rapidement adapter le design au problème en cours de résolution. Une reconfiguration dynamique efficace, tire partie du flot de données et du type des données que le système est en train de traiter. Ainsi, le composant est en adéquation avec le problème à traiter, et la résolution en est d'autant plus efficace.

La reconfiguration dynamique peut être effectuée de trois manières différentes :

— Alterner entre plusieurs designs compilés par avance [ELD 94]. Une fois la compilation effectuée, la substitution d'un design par un autre ne prend que quelques millisecondes.

— Modifier directement les équations logiques dans un CLB à partir d'informations connues uniquement lors de l'exécution. Nous modifions directement le fichier bitstream et le rechargeons sur le FPGA. Ceci est une des manières les plus rapides d'adapter le matériel au problème à résoudre [LEM 95a], [LEM 95b].

— Régénérer le design à la volée à partir des fichiers du type Xilinx Netlist File (le langage bas niveau pour les FPGAs Xilinx) avec des informations connues en cours d'exécution du programme. Cette forme de reconfiguration augmente la densité du design.

6. Amélioration des performances de GSAT par l'utilisation de FPGAs

Nous souhaitons accélérer l'exécution de l'algorithme GSAT en implémentant les parties les plus coûteuses sur matériel reconfigurable. Dans les sections suivantes, nous présentons l'implémentation générale de GSAT sur FPGA. Dans cette réalisation, la partie logicielle gère les contrôles de flux et l'initialisation de chaque essai.

6.1. *Implémentation générale*

Nous présentons ici la méthode utilisée pour effectuer un flip sur la carte reconfigurable.

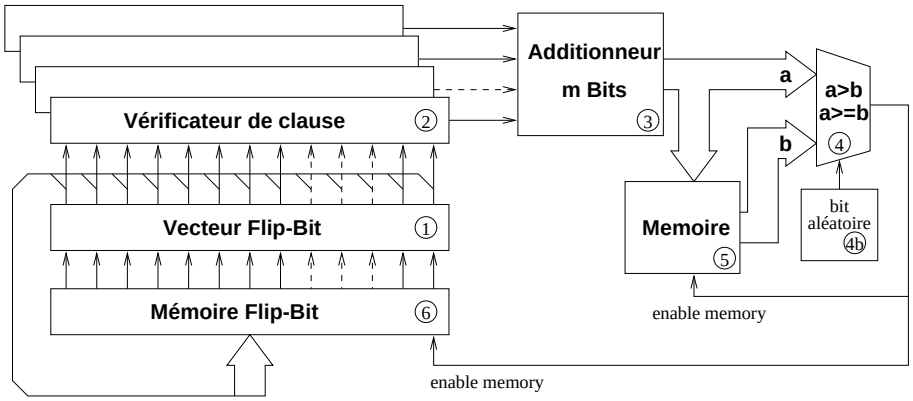


Figure 2. *L'implémentation générale de GSAT sur FPGA*

1. Initialisation du vecteur de variables : le vecteur de variables est initialisé par une valeur aléatoire qui est fournie par la partie logicielle. Un générateur de bit aléatoire serait implémentable sur FPGA, mais le coût d'un tel générateur est d'au moins 7 LUTs, ce qui signifie 3 CLB et demi. Pour le vecteur initial de variables, tel que nous sommes en train de le positionner ici, il est nécessaire que toutes les variables, avec un bit par variable, soient initialisées de façon aléatoire. Ceci implique une dépense de $7 \times n$ LUTs ou $7 \times n/2$ CLBs, n étant le nombre de variables booléennes de la forme conjonctive. La génération du vecteur aléatoire sur FPGA peut également être imaginée en n'utilisant qu'un unique générateur d'un bit et en décalant ce bit ainsi généré dans le vecteur. Cette dernière solution engendre le développement de contrôles supplémentaires concernant le décalage pour l'initialisation. Ceci est coûteux en flip-flops, alors que nous sommes déjà gros consommateurs de flip-flops dans notre implémentation (voir figure 3). Pour les raisons de coût inhérent aux deux solutions que nous venons d'invoquer, nous avons donc choisi de traiter l'initialisation du vecteur sur la partie logicielle de notre implémentation. Un autre avantage du choix d'une implémentation logicielle est de permettre l'implantation aisée de diverses heuristiques destinées à améliorer la recherche, ainsi que nous le verrons dans la section 7. Une fois le vecteur de variables initialisé, chaque variable est amenée à être flippée l'une après l'autre, et est remise à sa valeur initiale, lors du coup d'horloge suivant. Cette partie de l'implémentation coûte n coups d'horloges, n représentant le nombre de variables du problème.

2. Vérification des clauses : A chaque coup d'horloge une variable est flippée dans l'étape (1). A partir de l'instanciation des variables donnée en (1), nous vérifions chacune des clauses en parallèle. Ceci signifie que le temps passé à la vérification des clauses pour un vecteur de variables donné est indépendant du nombre de clauses. La résolution de cette étape coûte un coup d'horloge.

3. Nous comptons le nombre de clauses satisfaites lors de l'étape (2) à l'aide d'un additionneur m bits, où m est le nombre de clauses de la forme normale conjonctive. Cette partie de l'algorithme se fait en un coup d'horloge, ce qui signifie que le comptage

des clauses est effectué en parallèle. Il est évident que l'implémentation d'un additionneur m bits implique une cascade d'additionneurs de type half-adder et full-adder. Il va donc sans dire que cette cascade d'additionneurs entraîne un retard p entre l'entrée de l'additionneur et sa sortie. Dans notre problème, cela signifie que toutes les clauses sont bel et bien traitées en parallèle. À chaque coup d'horloge nous calculons le nombre de clauses satisfaites, mais, entre le moment où le vecteur de variables est présenté aux vérificateurs de clause et le moment ou l'on connaît le nombre de clauses satisfaites, il s'est écoulé $O(log_2(m)) + 1$ coups d'horloges.
Contrairement à l'implémentation logicielle incrémentale de GSAT, qui lors de chaque flip recalcule le score d'une partie des variables, notre design par la parallélisation du contrôle des clauses calcule successivement le score de chaque variable du problème.

4. Le nombre de clauses satisfaites est comparé à la valeur maximale du nombre de clauses satisfaites score-max stockée en (5). Chacune de ces deux valeurs utilise $O(log_2(m))$ bits. La comparaison effectuée est soit du type *plus grand que* ou *plus grand ou égal* selon la valeur d'un bit aléatoire donnée par un générateur implémenté sur le composant (4b). Comme cela a été explicité dans la section 3, cela permet de sortir d'un optimum local. Le score-max est initialisé à 0 dans (5) avant le premier flip de variable. Cette partie de l'implémentation est considérée (comme pour l'additionneur précédemment présenté) comme coûtant un coup d'horloge auquel il faut rajouter un retard $O(log_2(log_2(m)))$.

5. Si le nombre de clauses satisfaites est plus grand que (ou plus grand ou égal, selon le bit aléatoire donné en (4b)) le score-max stocké, on autorise le stockage du résultat de l'additionneur dans la mémoire (5). Ceci signifie que le meilleur score-max obtenu est conservé en permanence.

6. Si le nombre de clauses satisfaites est plus grand que (ou plus grand ou égal, selon le bit aléatoire donné en (4b)) le précédent score-max stocké, alors on autorise également le stockage du vecteur de variables qui a généré le nouveau score.

7. Après que chaque variable ait été flippée, le vecteur de variables stocké en (6) est utilisé comme nouveau vecteur initial. Il est donc transféré en (1).

Dans notre implémentation, la complexité totale d'un flip est n coups d'horloges, tandis que dans une implémentation logicielle, cette complexité est fonction à la fois du nombre moyen de clauses où intervient une variable ainsi que de la taille de ces clauses. La complexité constante de l'implémentation logicielle semble plus efficace, mais elle nécessite une mise à jour de plusieurs structures de données complexes pour chaque flip de variable. La section 8 montre combien l'implémentation matérielle, avec ses n opérations réellement élémentaires est plus efficace.

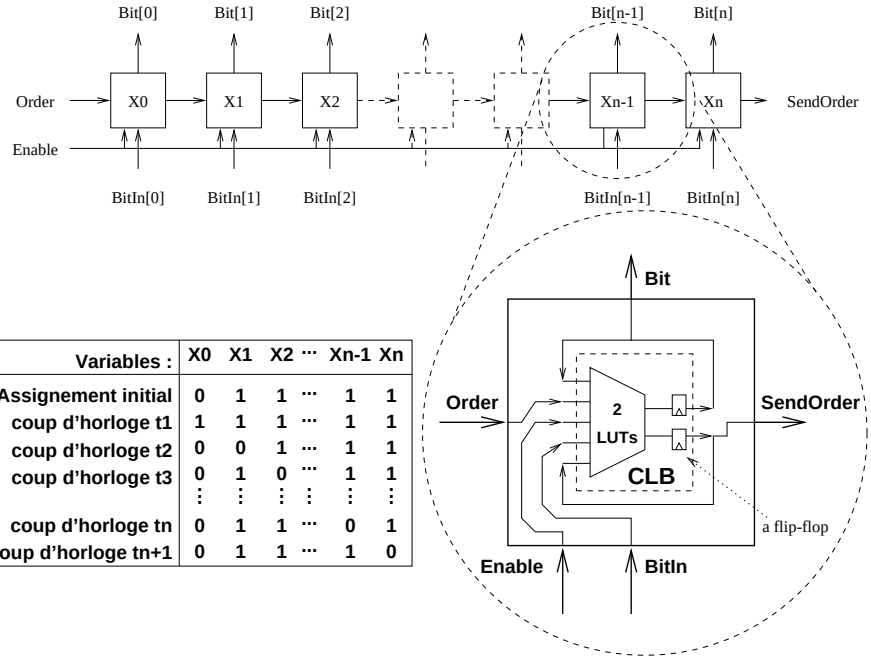
6.2. *Le vecteur flip-bit*

Nous nommons vecteur Flip-bit le vecteur des variables du problème SAT à traiter, il est composé de n Flip-bit. Ce vecteur est initialisé avec une valeur par défaut définie par la partie logicielle de notre système. Chaque variable est flippée tour à tour.

Cela signifie qu'une variable voit sa valeur complémentée si sa voisine de gauche a été flippée au coup d'horloge précédent. Elle reprend sa valeur initiale au coup d'horloge suivant.

Dans son implémentation sur FPGA, chaque Flip-bit est une variable qui flippe sa valeur lorsqu'elle en reçoit l'ordre, envoie l'ordre à sa voisine de droite, puis reprend sa valeur initiale.

La valeur initiale d'un Flip-bit est positionnée par l'utilisation du bit d'entrée $BitIn$ et en autorisant le stockage de cette valeur ($Enable$=1). Le Flip-bit reçoit l'ordre de se flipper grâce à l'entrée $Order$ puis renvoie cet ordre au Flip-bit de droite par sa sortie $SendOrder$ au coup d'horloge suivant. Chaque Flip-bit communique sa valeur sur la sortie nommée Bit (voir la figure 3).



Variables :	X0	X1	X2	···	Xn-1	Xn
Assignement initial	0	1	1	···	1	1
coup d'horloge t1	1	1	1	···	1	1
coup d'horloge t2	0	0	1	···	1	1
coup d'horloge t3	0	1	0	···	1	1
	⋮	⋮	⋮	⋮	⋮	⋮
coup d'horloge tn	0	1	1	···	0	1
coup d'horloge tn+1	0	1	1	···	1	0

Figure 3. *Un vecteur de Flip-bit avec zoom sur un Flip-bit particulier*

initiale, à chaque coup d'horloge une variable est flippée et retrouve sa valeur le coup d'horloge d'après

Nous présentons ci-dessous le comportement interne d'un Flip-bit :

1. A la première étape, chaque Flip-bit doit présenter sur sa sortie la valeur qu'on lui attribue initialement. (Bit = $BitIn$ AND $Enable$). Le Flip-bit mémorise alors sa valeur lorsqu'il y est autorisé (Enable=1).

2. Si un Flip-bit reçoit l'ordre de se flipper ($Order$=1), alors la valeur présentée sur sa sortie Bit doit s'inverser (Bit = $\overline{Bit}$ AND $Order$). Au même moment, il envoie l'ordre à son voisin de droite de se flipper ($SendOrder$=1).

3. Si un Flip-bit vient d'inverser sa valeur en sortie (et qu'il a commandé son voisin de droite), il doit reprendre sa valeur initiale (Bit = $\overline{Bit}$ AND $SendOrder$).

4. Dans tous les autres cas, le Flip-bit doit envoyer sa valeur initiale (Bit = Bit AND $\overline{Enable}$ AND $\overline{Order}$ AND $\overline{SendOrder}$).

Il est donc possible de définir la sortie Bit d'un Flip-bit de la manière suivante :

Bit = (($BitIn$ AND $Enable$) OR ($\overline{Bit}$ AND $Order$) OR ($\overline{Bit}$ AND $SendOrder$) OR (Bit AND $\overline{Enable}$ AND $\overline{Order}$ AND $\overline{SendOrder}$).

Lorsqu'un Flip-bit reçoit l'ordre d'inverser sa valeur en sortie, il envoie le même ordre à son voisin de droite le coup d'horloge suivant. D'un point de vue purement logique la sortie $SendOrder$ peut être définie comme suit :

$SendOrder$ = $Order$

Ainsi, si l'horloge interne du système fonctionne à la fréquence de 60MHz (ce qui est envisageable dans les conditions d'une bonne densité de design et d'un bon routage), un coup d'horloge prend $16.7ns$. Pour un vecteur de 50 variables, un flip coûte donc $50 \times 16.7 = 833.33ns$.

6.3. *Vérificateur de clauses*

Cette partie définit les vérificateurs de clauses. Un vérificateur de clauses est une unité de calcul qui prend en entrée les variables nécessaires à la résolution de la clause à résoudre.

Toutes les clauses sont traitées en parallèle, ce qui veut dire au même coup d'horloge. Cela prend donc le même délai pour vérifier une, deux ou m clauses.

Deux sortes d'implémentations sont possibles pour améliorer l'efficacité des vérificateurs de clause.

La première possibilité (voir figure 4) est de fournir à chaque clause un accès à toutes les variables (ce qui permet au passage de résoudre tout problème de type n-SAT), et lors de l'exécution de choisir quelle variable doit être utilisée dans telle ou telle clause. Ceci implique qu'à l'exécution, une clause ne doit prendre en compte que les variables qui lui sont nécessaires et faire abstraction des autres. Cette solution a ceci d'intéressant, qu'une fois le design compilé, nous pouvons très rapidement instancier sur ce design n'importe quel problème n-SAT de m clauses et avec au plus n variables. Cependant cette solution est coûteuse en CLB.

La deuxième implémentation consiste à construire un design spécifique dans lequel seules les variables utiles sont acheminées aux clauses (voir figure 5). Évidemment, cette solution demande plus de temps pour la compilation du design que la solution précédemment évoquée dans laquelle un design générique est construit et où le bitstream est modifié à la volée. Cette solution permet également d'économiser le nombre de CLB utilisés. Un seul CLB est nécessaire à la construction d'une clause 3-SAT. La figure 5 donne un exemple de clauses de type 3-SAT. Dans cette exemple, les clauses sont définies comme suit : $C1 = (x1 \vee \overline{x4} \vee \overline{x7})$, $C2 = (x2 \vee \overline{x3} \vee x6)$ and $C3 = (\overline{x4} \vee x5 \vee x8)$

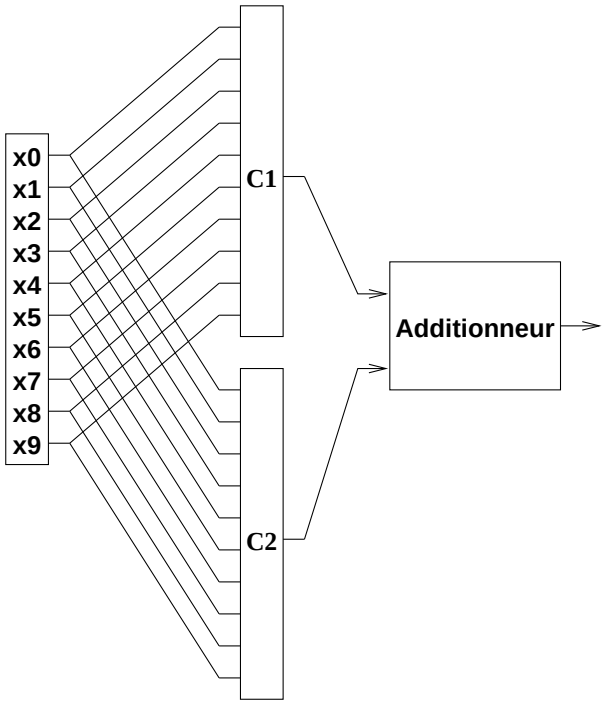


Figure 4. *Un exemple de vérificateur de clauses 3-SAT*

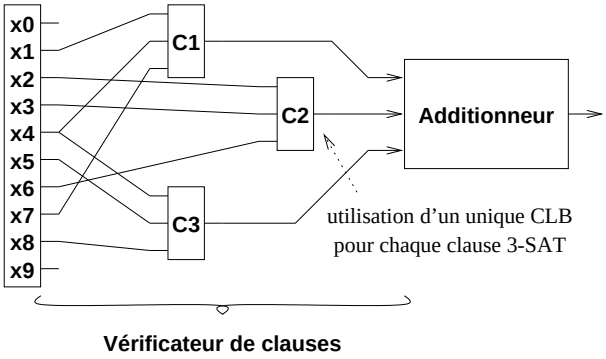


Figure 5. *Un exemple de vérificateur de clauses 3-SAT*

Chaque clause de type 3-SAT a ses équations définies dans une table de vérité (LUT), la négation n'est pas représentée sur les figures 4 et 5, elle fait partie des LUTs.

6.4. *Additionneurs*

Chacun des CLBs que nous utilisons possède 2 bits en sortie. Ce qui signifie que nous sommes contraints, pour la construction d'un additionneur de m-bits en entrée, d'utiliser des additionneurs ayant 2 ou 3 bits en entrée (respectivement appelés Half-Adder et Full-Adder). Nous programmons dynamiquement sur FPGA la construction de l'additionneur grâce à une méthodologie définie par L. Dadda. Présentée en 1965 cette méthodologie permet la construction optimisée d'un additionneur m bits à partir de Half-Adders et de Full-Adders [DAD 65]. Comme cela a été montré dans [LEM 95b], il est possible d'implémenter dynamiquement sur FPGA des additionneurs à partir de cette méthode.

6.5. *Mémoires*

Il est trivial de construire une mémoire k-bits sur un FPGA. A la sortie de chaque CLB, 2 flip-flops permettent de mémoriser 2 bits jusqu'au coup d'horloge suivant. Afin de stocker ce bit aussi longtemps que nécessaire, il est possible d'inhiber l'arrivée du signal d'horloge utilisant l'entrée "enable clock".

6.6. *Comparateurs*

Un CLB dispose de 5 bits en entrée. Nous souhaitons comparer 2 valeurs dans ce comparateur. Nous sommes donc contraints d'affecter au mieux 2 fois 2 bits en entrée. Nous devons donc comparer les bits de même poids et cascader ces comparateurs afin de construire un comparateur de k-bits.

Comme il a été dit dans la section 3, l'indéterminisme de GSAT dans le cas d'égalité de scores est essentiel à son fonctionnement optimal. Afin de garder cette caractéristique, nous avons, dans notre approche, ajouté au design la génération d'un bit aléatoire qui commande le fonctionnement du comparateur. Ce dernier peut se comporter comme un comparateur de type supérieur, ou supérieur ou égal, selon le bit aléatoire. Cela signifie que, dans le cas de deux scores égaux, le comparateur choisit aléatoirement de garder l'une ou l'autre des configurations.

7. Heuristiques

La simplicité de la procédure GSAT lui confère rapidité et efficacité dans l'exploration. Cependant certains problèmes fortement structurés résistent au mécanisme de

choix de la procédure. Offrir des possibilités de résolutions de tels problèmes est primordial puisque pour la plupart, les problèmes réels sont fortement structurés.

Pour une méthode de recherche complète (de type backtracking), un ordre sur les variables a plusieurs implications sur la recherche. Il affecte l'espace élagué par les méthodes de type avant (ex. forward checking [HAR 80]) et influe aussi sur la détection rapide des échecs prônée par ces méthodes. Un tel ordre influence également le nombre de backtracks et la profondeur de ces retours arrière il est donc à considérer lors de l'emploi de schémas de retour "intelligent" (ex. conflict-directed backjumping [PRO 93]).

GSAT et de façon générale une méthode de recherche locale, n'étend pas une instanciation partielle des variables, elle considère à chaque fois pour chaque état atteint, l'ensemble des états atteignables par une perturbation simple de l'état courant. Les variables ne sont pas ordonnées, les heuristiques classiquement utilisées en exploration complètes sont inapplicables.

Différentes heuristiques ont cependant été proposées par le passé pour modifier le comportement de base de la méthode [SEL 93a]. Par leur ajout l'algorithme est à même de traiter des instances particulièrement difficiles (notamment des instances structurées). Nous nous proposons ici de montrer comment augmenter notre design pour implémenter efficacement deux heuristiques mises au point pour la procédure. La première associe un poids à l'insatisfaction de chaque clause. La seconde utilise les échecs précédents pour rediriger la recherche vers des voies plus prometteuses.

7.1. *Heuristique des poids*

Cette technique résulte de l'observation que, sur certains problèmes, différentes tentatives de résolution procurent le même état final. La série de max-flips perturbations conduit donc à une affectation rendant insatisfaites les mêmes clauses. Cela signifie que certaines clauses sont plus difficilement satisfaisables que d'autres. Cela signifie aussi que ces clauses interviennent plus fortement dans le problème puisqu'elles sont plus difficiles à satisfaire. L'heuristique des poids permet d'une part de détecter ces parties du problèmes, d'autre part d'accroître progressivement leur importance dans le calcul des scores.

Un entier figurant le poids est associé à chaque clause, lors du calcul du score de chaque variable l'insatisfaction d'une clause est pondérée par son poids. Ainsi une clause "dure" verra son influence accrue dans le calcul de la satisfaction globale. De plus, le poids de chaque clause peut être automatiquement extrait durant la recherche en exploitant les états finaux des tentatives précédentes. Les deux étapes suivantes illustrent cette extraction :

1. Initialisation du poids de chaque clause à 1
2. A la fin de chaque essai infructueux augmenter (typiquement de 1) le poids de chaque clause insatisfaite

Dans notre implémentation FPGA ce mécanisme peut être rajouté en modifiant le poids de chaque clause insatisfaite dans les entrées de l'additionneur (voir section 6.4). La détermination des clauses insatisfaites et l'incrément de leurs poids est facilement implantable puisqu'elle a lieu sur la partie logicielle de l'architecture au démarrage de chaque essai. Cette heuristique est assimilable à une reconfiguration dynamique de l'algorithme.

7.2. *Moyenne des états précédents*

Après chaque tentative infructueuse l'algorithme repart avec une affectation des variables totalement nouvelle. Cependant le mécanisme de convergence de la méthode fait que les essais infructueux permettent souvent d'atteindre la satisfaction de larges parties du problème, c'est-à-dire de nombreuses clauses. Cette heuristique s'offre de réutiliser la meilleure partie de ces affectations porteuses de satisfaction partielle dans la génération de nouvelles instanciations de départ des variables. Elle considère les deux affectations précédentes pour en générer une troisième.

Lors de l'exécution l'instanciation de départ de la i^{eme} tentative est générée en utilisant la moyenne des bits [4] des deux tentatives précédentes. Plutôt que l'état final des essais $i-2$ et $i-1$ l'heuristique utilise les deux meilleurs états rencontrés lors des deux essais précédents. Même si le processus de convergence fait que bien souvent état final et meilleur état rencontré par le passé diffèrent peu. Une génération classique est utilisée pour la génération des deux premiers états de départ. Cette génération de nouveaux états par réutilisation d'états intermédiaires est d'ailleurs à la base des méthodes de recherche de type génétique [VOS 92]. La génération des affectations de départ étant assurée par la partie logicielle, l'incorporation de cette heuristique dans notre design est une fois de plus aisée. Nous considérons, dans notre design, le meilleur état généré dans chaque tentative comme étant l'état final obtenu à la fin de chaque tentative.

8. Discussion

8.1. *Taille de problème*

La série de FPGAs nommée Virtex, dernièrement mise au point par Xilinx, est la première à dépasser le million de portes logiques. Elle permet l'implantation de méthodes de résolution de grande taille sur le même composant, réduisant ainsi les problèmes liés à la gestion des entrées/sorties que l'on rencontre lors d'une résolution distribuée sur plusieurs composants. Le XCV1000 est un des composants de cette série. Il comporte 27648 cellules logiques similaires aux CLBs présentés en section 4. Les mesures faites par Xilinx sur ce composant établissent qu'il permet, dans une grande majorité des cas, un fonctionnement du design à une fréquence située entre 100Mhz et

4. Les bits de même ordre sont considérés deux à deux, les bits identiques représentants l'instanciation des mêmes variables sont réutilisés les autres sont générés aléatoirement

200Mhz. Par exemple, ces mesures montrent qu'un additionneur de 64 bits nécessite $7.2ns$ (soit une fréquence d'horloge admissible de 130Mhz). Le transfert de données jusqu'à ce composant est possible via un bus PCI cadencé à 66Mhz.

Nous proposons donc dans nos estimations, un calcul des gains attendus basé sur l'utilisation d'un tel composant cadencé à 60Mhz.

Sur un composant du type XCV1000, le nombre de cellules logiques disponibles doit aisément permettre l'implémentation de problèmes 3-SAT de plus de 1000 variables et 4300 clauses.

8.2. *Performances*

Le tableau suivant (voir tableau 1) résume les résultats présentés par B. Selman dans [SEL 93a] (programme C, CPU MIPS R6000). Ces résultats présentent les performances de GSAT lors du traitement de six instances de problèmes aléatoires difficiles ($p1$ à $p6$). Ces problèmes sont de type SAT, chaque clause étant constituée d'exactement 3 littéraux, on parle alors de problèmes 3-SAT. Ils sont qualifiés de difficiles car ils se situent tous au point de difficulté maximum de la zone de transition de phase [5]. Pour ce type d'instances (3-SAT aléatoires) l'appartenance à la zone de difficulté maximale a été identifiée par un ratio $n/m \simeq 4.3$. Ces résultats sont comparés avec les temps supposés pour notre implémentation matérielle en effectuant la même série de perturbations aléatoire. Il s'agit donc là de temps issus d'une simulation.

Plusieurs commentaires sont possibles sur l'algorithme logiciel. Premièrement, le nombre d'essais [6] nécessaires pour atteindre un état satisfaisable est fonction de la taille du problème (voir 5^{eme} col.). En effet plus le problème est de taille importante, plus la méthode nécessite de tentatives pour atteindre l'optimum. Ces différentes tentatives nécessitent toutes la définition de nouvelles affectations de départ consommatrices en ressources de calcul. Enfin des facteurs technologiques interviennent, tels que les accès cache/mémoire plus fréquents pour les problèmes importants. C'est ce qui explique la baisse du ratio flips/secondes calculé à partir du temps global. Cela ne remet pas en cause notre analyse sur le caractère incrémental de la méthode mais révèle simplement que la complexité effective d'une réparation est fortement dépendante de la taille du problème.

Pour les instances les plus importantes ($p5$ et $p6$) ce ralentissement rend la version logicielle inutilisable dans le cas d'applications interagissant avec un agent humain. L'échelle des temps de réponse autorisés pour de telles interactions a été définie notamment par A. Newell [NEW 94]. L'auteur propose la définition d'une human cognitive

5. La zone de transition de phase correspond à l'endroit où les problèmes passent de l'état peu contraint à l'état très contraint. Cette zone s'observe en faisant varier la dureté des problèmes. Dans cette zone, le point de dureté particulier où environ la moitié des instances possèdent une solution correspond au degré de difficulté maximum pour une méthode de recherche (réparative ou complète). Pour une étude sur le sujet se reporter à [CHE 91], [WIL 93] [HOG 94], [SMI 94].

6. En multipliant le nombre d'essais par le paramètre mflips, l'on obtient le nombre exact de perturbations effectuées.

band (10^-1 à 10 secondes) caractérisant l'essentiel des tâches cognitives humaines. Un système prétendant interagir efficacement avec l'utilisateur doit permettre des temps de réponses en adéquation avec cet intervalle.

formules			logiciel incrémental				matériel à 60MHz		accélération
pbs	vars	clauses	$mflips$	essais	temps	flips/s	temps	flips/s	
$p1$	50	215	250	6.4	0.4s	4000	1.3ms	12.10^5	300
$p2$	100	430	500	42.5	6s	3542	35.4ms	60.10^4	169
$p3$	140	602	700	52.6	14s	2630	85.5ms	42.10^4	163
$p4$	150	645	1500	100.5	45s	3350	0.3s	40.10^4	120
$p5$	300	1275	6000	231.8	12mn	1932	6.95s	20.10^4	103
$p6$	500	2150	10000	995.8	1.6h	1729	83s	12.10^4	70

Figure 5. *Comparaison des performances : logiciel/approche mixte*

Avec notre système, le ratio flips/secondes (9^{eme} col.) est uniquement fonction du nombre de variables du problème. La parallélisation du contrôle de satisfaction des clauses autorise notre design à s'affranchir de leur nombre. Cependant cette valeur intervient toujours dans la complexité spatiale du design. La dernière colonne du tableau présente l'accélération permise par notre méthode. Elle varie entre 300 et 70, cette dégradation est fonction du nombre de variables incluses dans la codification normale conjonctive des problèmes. Nous voyons donc que même confronté à la version incrémentale notre algorithme offre des performances largement meilleures. Considérons les problèmes les plus durs ($p5$ et $p6$), les temps fournis (6.95s contre 12mn et 83s contre 1.6h) laissent entrevoir l'usage de notre méthode dans le cadre d'applications interactives utilisant des problèmes de tailles comparables.

L'initialisation de chaque tentative de résolution (essai) est faite par la partie logicielle de notre architecture. Cela autorise une très grande flexibilité entre les essais et permet notamment l'usage de toutes sortes d'heuristiques (actuelles ou futures) tirant parti des échecs précédents (voir section 7) [SEL 93a], [FRA 96], [FRA 97]. Or, les temps présentés ici concernent uniquement la partie matérielle et ne reflètent donc pas les temps passés dans la génération plus ou moins fine d'affectations de départ. Cependant puisque logiciel et matériel peuvent opérer simultanément la génération d'états de départ peut être masquée et ne pas intervenir dans la complexité temporelle globale [7].

GSAT a été plus récemment opposée à diverses méthodes (complètes/incomplètes) [SEL 96]. Ces résultats peuvent servir de base pour une nouvelle extrapolation des temps de notre design. Les accélérations sont alors du même ordre que celles présentées ici (plusieurs ordre de grandeur).

7. A titre de référence, la mise à jour d'une carte reconfigurable de type DecPerle-1 nécessite un temps fixe d'environ $100ms$ (bitstreams pré-compilés).

8.3. *Retours d'expériences*

Dans cette étude, le partition de l'algorithme GSAT entre les entités logicielle et matérielle n'a pas fait l'objet d'une méthodologie de partitionnement matériel/logiciel particulière (CoDesign). Le passage d'un problème résolu jusqu'à présent de manière logicielle en une résolution concurrente matérielle/logicielle doit être motivé par l'espérance d'un gain de performances. Le gain en performance que nous avons espéré sur ce travail est un gain en durée d'exécution. Dans notre approche, l'accélération matérielle est permise par une parallélisation de la résolution. Cette programmation "spatiale" nous a permis d'accélérer fortement les transitions dans l'espace d'état du problème.

Le choix d'un partitionnement matériel/logiciel exige de manière générale l'étude de nombreux critères. Nous citerons entre autre :

— La facilité de décomposition du problème initial en sous-problèmes (dont l'implémentation se fera soit sur le matériel, soit sur le logiciel)

— Le gain en performance escompté (durée d'exécution, changement d'échelle des données traitées, etc. . .)

— L'expérience des développeurs dans le passage d'un sous-problème sur la partie matérielle (exemple : additionneur, comparateur)

— Les problèmes liés à la dépendance des sous-problèmes (communication, taille du bus de données, etc.).

Dans notre cas, l'étude du partionnement de GSAT entre le logiciel et le matériel n'a pu se faire que parce que nous avions une forte expérience de l'algorithme ainsi qu'une expertise importante de l'implémentation sur FPGA [LEM 95b]. Nous passons d'une solution totalement logicielle à une résolution concurrente matérielle/logicielle. Les optimisations mises en oeuvre en logiciel impliquent souvent des structures de données complexes, et de ce fait sont assez rarement implémentables sur FPGA. Ceci a été le cas dans notre étude comme dans nos travaux précédents où la distribution efficace de GSAT oblige à une réappropriation totale de la méthode [HAM 96].

L'implémentation d'un algorithme en développement concurrent matériel/logiciel nécessite souvent de revenir sur une version antérieure du logiciel, version souvent qualifiée de "basique". Le développeur ne doit donc pas hésiter à revenir à une version algorithmiquement moins performante, pour espérer ensuite obtenir un gain significatif sur une implémentation mixte matérielle/logicielle. D'une façon générale, c'est ce que l'on constate en algorithmique parallèle/distribuée, où les méthodes séquentielles les plus performantes sont souvent les moins distribuables/parallélisables [MAC 85].

9. Conclusion et perspectives

Nous avons initié, dans ce travail, une nouvelle utilisation des architectures reconfigurables, dont la souplesse et la montée en capacité doivent pouvoir permettre à terme une utilisation dans divers domaines d'applications particulièrement gourmands en cal-

culs. L'intelligence artificielle est un de ces domaines. Les espaces explorés y sont de taille exponentielle et disposent souvent d'heuristiques puissantes. La combinaison d'un matériel reconfigurable guidant une méthode logicielle "futée" y serait, nous le pensons, particulièrement à propos. C'est dans cette optique que ce travail a été réalisé. Nous pensons avoir en quelque sorte défriché ce nouveau domaine d'application.

La procédure GSAT obtient d'excellents résultats dans le traitement des problèmes de satisfaction (SAT) et de satisfaction partielle (MAX-SAT). Cette efficacité reconnue [BEE 94] a suscité notre intérêt pour la méthode. GSAT est une méthode de recherche locale. En tant que telle, elle possède une faible complexité structurelle [8] (pas de retour arrière). Les explorations de l'espace d'états sont systématiques (examen du voisinage puis choix d'un état dans ce voisinage). Ces deux raisons font de ces méthodes les parfaites candidates pour une implantation matérielle. Finalement, le format d'entrée de GSAT (CNF) est un autre argument pour son adaptation au matériel.

Une implantation de notre design peut être imaginée sur un ASIC. Cependant, les FPGAs offrent la reconfigurabilité nécessaire à la mise en oeuvre de diverses heuristiques existantes ou futures (voir section 7). Il nous semble illusoire de figer matériellement un résolveur générique de problèmes à contraintes. A l'inverse, imaginer différentes configurations relatives à des classes de problèmes nous semble réalisable et efficace. Les FPGAs peuvent être abordés comme des co-processeurs génériques qui à terme pourraient être intégrés au sein même des microprocesseurs conventionnels [LEM 95a]. C'est dans ce cadre que notre étude a été basée.

Dans ce travail nous avons proposé une implantation sur architecture reconfigurable de l'algorithme. Les buts recherchés étant d'une part le traitement de problèmes de très grande taille, d'autre part un traitement temps-réel de problèmes de taille plus réduite. L'implémentation FPGA en atteignant ces deux objectifs permet un élargissement du champ d'application de la procédure. Ces buts ont été atteints grâce à la flexibilité et à la variabilité du matériel reconfigurable. En dehors de l'accélération brute autorisée par le matériel, la programmation spatiale utilisée pour le contrôle de satisfaction des clauses du problème autorise de bonnes performances. Cette parallélisation nous affranchit du nombre de clauses, la complexité d'exécution dépendant alors uniquement du nombre de variables. L'exploration de l'espace de recherche est donc très rapide, par exemple un matériel tournant à 60Mhz permet de considérer 60.10^6 configurations par seconde.

L'architecture choisie est ouverte, elle se compose d'une partie matérielle pilotée par une partie logicielle. Cette utilisation du logiciel nous offre une grande latitude dans le contrôle de la recherche et permet notamment l'emploi d'heuristiques efficaces. De plus, cette bipolarité peut être la base d'applications interactives puisque le matériel permet d'atteindre des temps de résolutions réalistes dans un tel cadre. Le formalisme SAT permettant de coder de nombreux problèmes d'IA, l'accélération permise par une implantation matérielle est susceptible à terme d'élargir le champ d'ap-

8. Cette complexité structurelle est à la base des faibles performances du design présenté dans [YOK 96] qui adaptent sur architecture reconfigurable l'algorithme de Davis et Putnam.

plication des technologies issues de l'intelligence artificielle. Comme futurs développements nous envisageons d'abord une réflexion sur l'implantation d'heuristiques présentées plus récemment [FRA 96], [FRA 97]. Ensuite, les méthodes de recherche locale exigeant toutes la définition de trois grands critères (voir section 2) destinés à diriger la recherche, le mécanisme d'exploration étant lui très similaire d'une méthode à une autre. Cela signifie que notre architecture de travail doit pouvoir être généralisable à d'autres méthodes de ce type, nous envisageons d'étudier cette possibilité de développement/généralisation.

Puisque la partie logicielle est inactive pendant chaque tentative (essai) exécutée par le matériel, nous envisageons de profiter de cette capacité inutilisée de calcul pour exécuter parallèlement une autre méthode de résolution, logicielle celle-là. Naturellement cette partie du travail nécessite la mise au point d'un cadre de coopération efficace (voir [HOG 93] pour diverses évaluations du cadre coopératif). Ce type d'approche mixte et coopérante a été remis en vogue récemment [MAZ 96]. Les deux algorithmes doivent alors être capables d'échanger des informations utiles sur l'instance en cours de traitement. Comme par exemple le partage de zones de l'espace de recherche détectées comme ne faisant pas partie d'une solution finale, *nogood* [DEC 90]. Ce dernier type d'informations peut être rapidement produit par le matériel et étendu par une méthode d'exploration arborescente complète [MIN 92].

10. Bibliographie

[BEE 94] BEERINGER A., ASCHEMANN G., HOOS H. H., METZGER M. et WEISS A., « GSAT versus Simulated Annealing ». In COHN A. G., Ed., *Proceedings of the Eleventh European Conference on Artificial Intelligence*, p. 130–134, Chichester, Aug 1994. John Wiley and Sons.

[BOT 92] BOTTEN L. C. et CADEN M. J., AMPL - a Mathematical Programming Language. In HOUSTIS E. N. et RICE J. R., Eds., *Artificial Intelligence, Expert Systems and Symbolic Computing: Selected and Revised Papers from the IMACS 13th World Congress, Dublin, Eire*, p. 436–445. North-Holland, Amsterdam, Netherlands, 1992.

[CHE 91] CHEESEMAN P., KANEFSKY B. et TAYLOR W. M., « Where the really hard problems are ». In MYLOPOULOS J. et REITER R., Eds., *Proceedings of IJCAI-91*, p. 331–337. Morgan Kaufmann, 1991.

[COO 92] COOPER P. R. et SWAIN M. J., « Arc consistency: Parallelism and Domain Dependence ». *AI*, vol. 58, n° 1–3, p. 207–235, Dec 1992.

[DAD 65] DADDA L., « *Some schemes for parallel multipliers* », vol. 19, p. 349–356. Alta Frequenza, 1965.

[DEC 90] DECHTER R., « Enhancements schemes for constraint processing: backjumping, learning and cutset decomposition ». *AI*, vol. 41, n° 3, p. 273–312, 1990.

[ELD 94] ELDREDGE J. G. et HUTCHINGS B. L., « RRANN: the run time reconfiguration artificial neural network ». In *In Proc of Custom Integrated Circuits Conference*, p. 77–80, 1994.

[FRA 96] FRANK J., « Weighting for Godot: Learning Heuristics for GSAT ». In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Inno-*

vative Applications of Artificial Intelligence Conference, p. 338–343, Menlo Park, August4–8 1996. AAAI Press / MIT Press.

[FRA 97] FRANK J., « Learning Short-Term Weights for GSAT ». In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, p. 384–391, San Francisco, August23–29 1997. Morgan Kaufmann Publishers.

[GLO 89] GLOVER F., « Tabu search: 1 ». *ORSA Journal on Computing*, vol. 1, n° 3, p. 190–206, Summer 1989.

[GUE 91] GUESGUEN H., « Connectionist Networks for Constraint Satisfaction ». In *AAAI Spring Symposium on Constraint-based Reasoning*, p. 182–190, Mar 1991.

[HAM 96] HAMADI Y., « Distribution de GSAT ». In HERMES, Ed., *Journées Francophones d'Intelligence Artificielle Distribuée et de Systèmes Multi-Agents*, p. 189–199, avr 1996.

[HAM 98] HAMADI Y., BESSIÈRE C. et QUINQUETON J., « Backtracking in Distributed Constraint Networks ». In *ECAI*, p. 219–223, Aug 1998.

[HAM 99] HAMADI Y., « Optimal Distributed Arc-Consistency ». In *Fifth International Conference on Principles and Practice of Constraint Programming (CP'99)*, p. 219–233, 1999.

[HAR 80] HARALICK R. M. et ELLIOTT G. L., « Increasing tree search efficiency for constraint satisfaction problems ». *AI*, vol. 14, p. 263–314, 1980.

[HOG 93] HOGG T. et HUBERMAN B. A., Better Than the Best: The Power of Cooperation. In NADEL L. et STEIN D., Eds., *1992 Lectures in Complex Systems*, vol. V de *SFI Studies in the Sciences of Complexity*, p. 165–184. Addison-Wesley, Reading, MA, 1993.

[HOG 94] HOGG T., « Statistical Mechanics of Combinatorial Search ». In *Proc. of the Workshop on Physics and Computation, PhysComp94*, p. 196–202, Los Alamitos, CA, 1994. IEEE Press.

[HOO 96] HOOS H. H., « Solving Hard Combinatorial Problems with GSAT ». *Lecture Notes in Computer Science*, vol. 1137, p. 107–117, 1996.

[KAM 92] KAMATH A. P., KARMARKAR N. K., RAMAKRISHNAN K. G. et RESENDE M. G. C., « A continuous approach to inductive inference ». *Mathematical Programming*, vol. 57, p. 215–238, 1992.

[LAR 90] LARRABEE T., « Efficient Generation of Test Patterns Using Boolean Satisfiability ». Rapport technique RR90/2, Digital research western lab., 1990.

[LEM 95a] LEMOINE E., « *Architecture reconfigurable dynamique. Une application à un système dédié à la recherche rapide dans les bases de données de séquences génétiques* ». PhD thesis, Université Montpellier II, 5 Juillet 1995.

[LEM 95b] LEMOINE E. et MERCERON D., « Run Time Reconfiguration of FPGA for Scanning Genomic DataBases ». In BUELL D. A. et POCEK K. L., Eds., *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, p. 90–98, Napa, CA, April 1995.

[MAC 85] MACKWORTH A. K. et FREUDER E. C., « The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problem ». *Artificial Intelligence*, vol. 25, n° 1, p. 65–74, 1985.

[MAZ 96] MAZURE B., SAIS L. et GREGOIRE E., « Boosting complete techniques thanks to local search methods ». In *Proc. Math and Artif. Intell.*, 1996.

[MIN 92] MINTON S., JOHNSTON M. D., PHILIPS A. B. et LAIRD P., « Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. ». *Artificial Intelligence*, vol. 58, p. 161–205, 1992.

[NEW 94] NEWELL A., Unified Theory of Cognition. Harvard University Press, 1994.

[PRO 93] PROSSER P., « Hybrid Algorithms for the constraint satisfaction problem ». *Computational Intelligence*, vol. 9, n° 3, p. 268–299, 1993.

[RAO 93] RAO V. N. et KUMAR V., « On the Efficiency of Parallel Backtracking ». *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, n° 4, p. 427–437, Apr 1993.

[SEL 92] SELMAN B., LEVESQUE H. et MITCHELL D., « A New Method for Solving Hard Satisfiability Problems ». In PRESS/MIT A., Ed., *Proceedings of AAAI*, p. 440–446, Jun 1992.

[SEL 93a] SELMAN B. et KAUTZ H., « Domain-Independant Extensions to GSAT : Solving Large Structured Satisfiability Problems ». In *IJCAI*, 1993.

[SEL 93b] SELMAN B. et KAUTZ H. A., « An Empirical Study of Greedy Local Search for Satisfiability Testing ». In *Proceedings of the 11th National Conference on Artificial Intelligence*, p. 46–53, Menlo Park, CA, USA, July 1993. AAAI Press.

[SEL 94] SELMAN B. et KAUTZ H., « Noise Strategies for Improving Local Search by Bart Selman and Henry Kautz ». In PRESS/MIT A., Ed., *Proceedings of AAAI*, p. 440–446, Jun 1994.

[SEL 96] SELMAN B., KAUTZ H. et COHEN B., « Coloring, and Satisfiability: Second DIMACS Implementation Challenge ». In JOHNSON D. S. et TRICK M. A., Eds., *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 26, 1996.

[SMI 94] SMITH B., « The Phase Transition in Constraint Satisfaction Problems: A Closer Look at the Mushy Region ». In *ECAI*, p. 100–104, 1994.

[STE 81] STEFIK M., « Planning With Constraints ». *AI*, vol. 16, n° 2, p. 111–139, 1981.

[SWA 88] SWAIN M. J. et COOPER P. R., « Parallel Hardware for Constraint Satisfaction ». In *National Conference on Artificial Intelligence*, p. 682–686, Los Altos, CA, 1988. Morgan Kaufmann.

[VOS 92] VOSE M., « Modeling simple genetic algorithms ». In WHITLEY D., Ed., *Foundations of Genetic Algorithms Workshop (FOGA-92)*, Vail, Colorado, July 1992.

[WAL 75] WALTZ D. L., Understanding line drawings of scenes with shadows. In WINSTON P. H., Ed., *The Psychology of Computer Vision*, p. 19–91. McGraw-Hill, 1975.

[WIL 93] WILLIAMS C. P. et HOGG T., « The Typicality of Phase Transitions in Search ». *Computational Intelligence*, vol. 9, n° 3, p. 221–238, 1993.

[XIL 91] XILINX, *The Programmable Gate Array Data Book*. Product Briefs, 1991.

[YOK 96] YOKOO M., SUYAMA T. et SAWADA H., « Solving Satisfiability Problems Using Field Programmable Gate Arrays: First Results ». In *Principles and Practice of Constraint Programming*, p. 497–509, 1996.

Article reçu le 9 novembre 1998.
Version révisée le 20 octobre 1999.
Rédacteur responsable : Dominique Lavenier

Youssef HAMADI *a obtenu un doctorat d'informatique au Laboratoire d'Informatique de Robotique et de Micro électronique de Montpellier (LIRMM). Son travail de thèse porte sur le traitement distribué de problèmes de satisfaction de contraintes. Ses principaux centre d'intérêt comprennent les traitements distribués et parallèles de problèmes représentables par le formalisme contrainte.*

David MERCERON *prépare un doctorat d'informatique au Laboratoire d'Informatique de Robotique et de Micro électronique de Montpellier (LIRMM). Son travail de thèse porte sur les principes d'acquisition et de construction de connaissances basés sur les interactions entre un utilisateur et son agent rationnel. Les architectures matériellement reconfigurables ont été étudiées dans cette thèse pour réduire les temps de réaction de l'agent rationnel. Par ailleurs, David MERCERON travaille comme consultant au Centre de Compétences Objet de la société EURIWARE.*