HAL
open science

Métaheuristiques pour l'allocation de mémoire dans les systèmes embarqués

Maria Soto, André Rossi, Marc Sevaux

To cite this version:

Maria Soto, André Rossi, Marc Sevaux. Métaheuristiques pour l'allocation de mémoire dans les systèmes embarqués. Actes du 11ème congrés national de la socitété française de recherche opérationnelle, ROADEF'2010, Feb 2010, Toulouse, France. pp.35-43. hal-00490232

HAL Id: hal-00490232

https://hal.science/hal-00490232v1

Submitted on 8 Jun 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Métaheuristiques pour l'allocation de mémoire dans les systèmes embarqués

María Soto, André Rossi, Marc Sevaux

Université de Bretagne-Sud. Lab-STICC, CNRS UMR 3192
Centre de Recherche B.P. 92116
F-56321 Lorient Cedex FRANCE
{maria.soto, andre.rossi, marc.sevaux}@univ-ubs.fr

Résumé :

La gestion de la mémoire cache a un impact significatif sur les performances et sur la consommation énergétique des systèmes embarqués. Cet article traite de l'allocation de mémoire des structures de données d'une application à la mémoire cache de manière à optimiser les performances du système. Les concepteurs de circuits souhaitent trouver un compromis entre le coût de l'architecture (le nombre de bancs mémoire à embarquer) et la consommation électrique. Le problème abordé consiste à allouer un banc mémoire à toute structure de données de manière à minimiser les conflits d'accès aux données. Le modèle proposé pour ce problème est le k-weighted graph coloring problem. Une formulation par PLNE et deux métaheuristiques basées respectivement sur une recherche taboue et sur un algorithme hybride à base de population sont comparées sur un ensemble d'instances. Les résultats obtenus sont encourageants et suggèrent que l'utilisation de méthodes issues de la coloration de graphes est une piste prometteuse pour l'allocation de mémoire dans les systèmes embarqués.

Mots-Clés : *Allocation de mémoire, métaheuristiques, coloration de graphes*

1 Introduction

Les progrès continus des technologies en microélectronique ont rendu possible le développement de puces miniatures permettant d'étendre considérablement les fonctionnalités des appareils multimédia portatifs (la téléphonie mobile en est un exemple spectaculaire). Alors que la technologie ouvre des perspectives et des marchés toujours plus vastes, la conception des circuits intégrés devient, elle, de plus en plus complexe. D'autant qu'une innovation permanente sur des produits à bas coût s'accompagne naturellement d'une obsolescence rapide des produits fabriqués, qui contraint les concepteurs de circuits à minimiser le temps de conception.

Traditionnellement, les circuits intégrés étaient conçus à la main par des spécialistes qui maîtrisaient la technologie et la complexité du produit dans son ensemble. Une telle façon de faire n'est plus possible aujourd'hui, et des outils de conception assistée par ordinateur comme Gaut [3] ont été introduits pour générer le design (ou l'architecture) d'un circuit intégré à partir de ses spécifications. Bien que les solutions produites par ce biais puissent être générées très rapidement, elles souffrent généralement d'un manque d'optimisation qui se traduit notamment par une consommation énergétique importante, ce qui est particulièrement préjudiciable pour les systèmes embarqués.

Cet article aborde la problématique de l'optimisation de l'allocation de mémoire cache des processeurs embarqués. En effet, la gestion de la mémoire cache a un impact significatif sur les performances et sur la consommation énergétique comme l'ont montré Wuytack *et al.* dans [15]. Lorsque l'on conçoit un circuit intégré pour un système embarqué (typiquement un décodeur MPEG, un filtre numérique ou tout autre algorithme de traitement du signal et de l'image), l'application à synthétiser est fournie sous la forme d'un code source écrit en langage C, et les structures de données manipulées doivent être placées dans des bancs mémoire (il s'agit de la mémoire cache du processeur qui exécute l'application).

Pour des raisons technologiques comme pour des raisons de coût, le nombre de bancs mémoire est limité. On fait également l'hypothèse que le processeur est capable d'accéder à tous ses bancs mémoire simultanément, ce qui autorise plusieurs chargements en temps masqué. Ainsi, les structures de données (ou plus simplement les variables) `a` et `b` peuvent être chargées en même temps lorsqu'on souhaite effectuer l'opération `a+b`, à condition que `a` et `b` se trouvent dans deux bancs mémoire différents. Si ces variables partagent le même banc mémoire, le processeur ne peut y accéder que séquentiellement, ce qui nécessite deux fois plus de temps qu'un accès en parallèle. On dira que `a` et `b` sont en conflit si elles apparaissent dans la même opération. Tout conflit a un coût, qui est proportionnel au nombre de fois où l'opération `a+b` est effectuée dans l'application (ce coût peut être un réel, dans le cas où le code a été "profilé" par un outil logiciel [7, 10] basé sur une estimation stochastique de la probabilité de branchement des instructions conditionnelles). Le recours à une estimation du nombre de fois où une opération est effectuée se présente lorsque l'opération apparaît dans une boucle `while` par exemple.

Un conflit entre deux structures de données sera dit *ouvert* si les deux structures de données partagent le même banc mémoire, il est dit *fermé* si les deux structures de données ont été affectées à deux bancs mémoire différents.

Ainsi, les concepteurs de circuits souhaitent trouver un compromis entre le coût de l'architecture (le nombre de bancs mémoires à embarquer) et la consommation énergétique de l'ensemble [1]. En effet, les électroniciens considèrent généralement que dans une certaine mesure, minimiser la consommation est équivalent à minimiser le temps d'exécution de l'application [2]. De plus, la consommation d'une architecture donnée peut également être estimée à l'aide d'un modèle empirique comme dans [8]. Ainsi, on a tout intérêt à paralléliser l'accès aux données de manière à minimiser le temps total d'exécution de l'application.

Le problème de l'allocation de mémoire peut donc être formulé ainsi : il convient d'allouer un banc mémoire à toute structure de données de l'application, de manière à minimiser le temps total d'accès aux données pour le processeur. Dans la section suivante, on propose un programme linéaire en nombres entiers pour résoudre ce problème de manière exacte. La section 3 présente deux métaheuristiques, une recherche taboue et une méthode à population, enfin la section 4 compare les résultats et discute les approches proposées.

2 Modélisation du problème par PLNE

Soit n le nombre de structures de données de l'application considérée. On désigne par a_i la structure de données i pour tout i dans $\{1, \ldots, n\}$. Le nombre de bancs mémoire disponibles est noté m. Par ailleurs, on note o le nombre de conflits entre deux structures de données. La paire $p_k = (a_{k1}, a_{k2})$ a le coût d_k pour tout k dans $\{1, \ldots, o\}$, ce coût s'exprime en unités de temps (typiquement la milliseconde). Le statut du conflit k dépend de l'allocation des structures de données a_{k1} et a_{k2}. Plus précisément, tout conflit se trouve nécessairement dans l'un des 2 statuts suivants :

– Statut 1 : a_{k1} et a_{k2} sont dans deux bancs mémoire différents. Le conflit est fermé, il n'engendre aucun coût,
– Statut 2 : a_{k1} et a_{k2} sont dans le même banc mémoire. Le conflit est ouvert, il engendre le coût d_k.

Il faut noter qu'une structure de données peut être en conflit avec elle-même. Le cas se produit typiquement lorsque la structure de données est un tableau et que l'application traduit une relation de récurrence comme `a[i] = a[i+1] ;`.

Toutes ces informations sont considérées comme des données fournies par le concepteur (ou plus exactement par un outil de *profiling* du code de l'application). L'entier n désigne le nombre de structures de données n, et m est le nombre de conflits entre ces structures de données. La liste des conflits (a_{k1}, a_{k2}) et leur coût d_k pour tout k dans $\{1, \ldots, o\}$ est une information qui caractérise à la fois l'architecture et l'application.

Les variables de décision du problème d'affectation des structures de données aux bancs mémoire sont :

– La matrice X, où $x_{i,j}$ indique que la structure de données a_i est affectée au banc mémoire b_j pour tout i dans $\{1, \ldots, n\}$ et pour tout j dans $\{1, \ldots, m\}$ ($x_{i,j} = 0$ sinon).
– Le vecteur Y représentant le statut des conflits. Pour tout k dans $\{1, \ldots, o\}$, l'élément y_k est égal à 1 seulement si le conflit k est ouvert, il vaut zéro sinon.

Le coût d'une solution (c'est à dire d'une affectation des structures de données aux bancs mémoire) est égal à la somme des coûts des conflits ouverts. Le coût d'une solution s'exprime comme suit :

$$\text{Minimiser} \sum_{k=1}^{o} y_k d_k \tag{1}$$

Les contraintes suivantes doivent être satisfaites pour que la solution soit réalisable.

Premièrement, toute structure de données est affectée à un banc mémoire :

$$\sum_{j=1}^{m} x_{i,j} = 1 \quad \forall i \in [1, n] \tag{2}$$

Deuxièmement, pour tout conflit, la variable y_k doit avoir la valeur numérique appropriée 0 ou 1. Cela est assuré à l'aide de la contrainte ci-dessous :

$$x_{a_{k1},j} + x_{a_{k2},j} \leq 1 + y_k, \qquad \forall k \in [1, o], \forall j \in [1, m] \tag{3}$$

Le problème peut alors être formulé comme un programme linéaire en nombres entiers comprenant la fonction objectif et l'ensemble des contraintes :

$$\begin{aligned}
&\text{Minimiser} && \sum_{k=1}^{o} y_k d_k && && (1)\\
&\text{sous les contraintes} && \sum_{j=1}^{m} x_{i,j} = 1 && \forall i \in [1, n] && (2)\\
& && x_{a_{k1},j} + x_{a_{k2},j} \leq 1 + y_k && \forall k \in [1, o],\ \forall j \in [1, m] && (3)
\end{aligned}$$

$$x_{i,j} \in \{0,1\} \qquad \forall i \in [1,n],\ \forall j \in [1,m] \tag{4}$$
$$y_k \in \{0,1\} \qquad \forall k \in [1,o] \tag{5}$$

Il faut noter que ce problème est identique au *k-weighted graph coloring problem* [14] : il s'agit de colorer un graphe non orienté dont les sommets représentent les structures de données et où chaque couleur représente un banc mémoire, de manière à minimiser la somme des coûts des arêtes dont les deux extrémités ont la même couleur. Une arête représente un conflit entre deux structures de données. Un problème proche consiste à déterminer le nombre minimum de bancs mémoire pour lequel aucun conflit n'est ouvert, il s'agit là d'un problème de coloration de graphes [4].

3 Solution du problème d'allocation mémoire

3.1 Méthode de résolution exacte

Une solution optimale peut être obtenue à l'aide d'un solveur, comme GLPK [6] ou Xpress-MP [16]. Mais comme le montrent les tests numériques de la section 4, une solution optimale ne peut pas être obtenue en temps raisonnable pour des instances moyennes. C'est la raison pour laquelle deux métaheuristiques sont proposées dans cet article.

3.2 Métaheuristiques pour le problème d'allocation mémoire

La première métaheuristique est une recherche taboue implémentée à partir d'une méthode taboue pour la coloration de graphes qui s'appelle *TabuCol* et décrit dans le chapitre 10 de [9]. L'autre métaheuristiques est un algorithme hybride à base de population qui s'appuie sur l'algorithme *Evocol (Evolutionary Hybrid Algorithm for Graph Coloring)* décrit dans [12]. Les sections suivantes présentent ces méthodes de manière détaillée.

3.2.1 Méthode taboue

La méthode taboue appartient aux métaheuristiques à base de voisinages qui s'appuient sur un même principe : l'algorithme passe de manière itérative de la solution courante à une solution voisine [13] . Le voisinage $\mathcal{N}(x)$ de la solution courante x dépend du problème traité. Généralement, les opérateurs de recherche locale s'arrêtent quand une solution localement optimale est trouvée. Il est alors nécessaire de sortir des minima locaux pour espérer trouver une solution de meilleure qualité. La méthode taboue y parvient en permettant que la solution courante soit temporairement dégradée. Pour ce faire, certaines modifications sur la solution courantes sont temporairement interdites (ou taboues), ce qui est à l'origine du nom de cette métaheuristique. On note NT le nombre maximal de mouvements interdits, il s'agit de la taille de la liste taboue.

La structure générale de la recherche taboue utilisée pour résoudre le problème est décrite dans l'algorithme 1, et commentée ci-après.

Les données de TabuMemexBasic sont les n structures de données, les m bancs mémoire, le coût des conflits entre les paires de structures de données concernées, deux paramètres de calibrage $Niter$ le nombre d'itérations de l'algorithme et NT la taille de la liste taboue. Le résultat de l'algorithme est la meilleure affectation mémoire trouvée, contenue dans $Best$.

Algorithme 1 : **TabuMemexBasic.**

Données : n structures de données, m bancs mémoire, le coût des conflits, $Niter$ nombre maximum d'itérations et NT la taille de la liste taboue.
Résultat : $Best$: la meilleure allocation trouvée.
`Initialisation`
Choisir une solution initiale s;
$Best \leftarrow s$;
$Iter = 0$;
`Phase itérative`
pour $((Iter \leq Niter)$ *ou* $(cost(s) > 0))$ **faire**
  Générer $s' \in \mathcal{N}(s)$, à partir de s en affectant la structure de données a_i au banc mémoire b_j de façon à ce que $cost(s') < cost(s''), \forall s'' \in N(s)$;
  si (*la paire* (a_i, b_j) *n'est pas dans la liste taboue*) **alors**
    $s \leftarrow s'$;
    Mettre à jour la liste taboue;
    si $cost(s) < cost(Best)$ **alors**
      $Best \leftarrow s$;
    fin
  fin
  Recalculer la taille de la liste taboue NT;
  $Iter \leftarrow Iter + 1$.
fin

La solution initiale est choisie au hasard, c'est-à-dire que les structures de données sont affectées aux bancs mémoire de façon aléatoire. Toute solution initiale générée de cette façon est nécessairement admissible car la capacité des bancs mémoire n'est pas prise en compte.

Dans la phase itérative, l'algorithme effectue la recherche de la meilleure solution au voisinage de la solution courante pendant un nombre maximal d'itérations, $Niter$, mais la recherche peut s'interrompre avant si une solution sans conflit ouvert est trouvée (une telle solution est nécessairement optimale). Le voisinage considéré est constitué de toutes les solutions obtenues en changeant l'affectation d'une structure de données quelconque.

A chaque itération, l'algorithme cherche la solution voisine de coût minimum, même si la solution trouvée est de plus mauvaise qualité que la solution courante. La recherche taboue interdit à la structure de données a_i d'être affectée à son ancien banc mémoire b_j pendant NT itérations, c'est-à-dire qu'une nouvelle solution sera acceptée si la paire (a_i, b_j) n'est pas dans la liste taboue. Chaque fois que la solution courante change, la liste taboue est mise à jour sur le principe du FIFO (First In First Out), et $Best$ est mise à jour si la solution courante est de coût strictement inférieur au coût de $Best$.

La taille de la liste taboue n'est pas constante, elle est ajustée à chaque itération, comme dans les travaux de Porumbel, Hao et Kuntz sur la coloration de graphes [12].

La section 4 présente l'implémentation de TabuMemexBasic de manière détaillée.

3.2.2 Algorithme mémétique

Cette sous-section présente la dernière métaheuristique intitulé EvoMemexBasic qui est décrite dans l'algorithme 2. EvoMemexBasic prend en entrée les n structures de données, le nombre de

bancs mémoire m et deux paramètres liés à l'algorithme : r nombre des parents impliqués dans le croisement et p le nombre d'enfants produits à chaque itération. Il retourne *Best*, la meilleure solution trouvée.

Algorithme 2 : **EvoMemexBasic.**

Données : n structures de donnée, m bancs mémoire, le coûts des conflits, le nombre de parents, r, impliqués dans chaque itération, et le nombre d'enfants, p, produits pour chaque itération.
Résultat : *Best* : la meilleure allocation trouvée.
`Initialisation`
Choisir une population aléatoire de solutions;
`Phase itérative`
répéter
  répéter
    répéter
      Sélectionner r parents ($r > 2$);
      Croiser les parents pour produire un nouvel enfant s;
      Appliquer la méthode TabuMemexBasic à l'enfant s;
      Accepter ou pas l'enfant s;
    jusqu'à *un enfant est accepté* ;
  jusqu'à *p enfants produits* ;
  Actualiser la population de parents avec les p enfants.
jusqu'à *un critère d'arrêt est satisfait* ;

L'initialisation consiste à générer un ensemble de solutions aléatoirement, ce qui est aisé étant donné que toute allocation des structures de données aux bancs mémoire est admissible.

Le principe général de l'algorithme EvoMemexBasic consiste à obtenir p nouveaux enfants en croisant r parents différents, c'est à dire r éléments de la population courante. Le croisement des r parents consiste à choisir les meilleures affectations des structures de données aux bancs mémoire de chaque parent pour former l'enfant comme indiqué dans le prochain paragraphe. Chaque enfant (nouvelle solution) ainsi produit est lui-même amélioré en l'utilisant comme solution initiale de la recherche taboue TabuMemexBasic.

L'algorithme 3 décrit ce croisement, il prend pour chaque banc mémoire les meilleures affectations des structures de données de façon à minimiser le *taux de conflit*, c'est à dire le nombre de conflits du banc mémoire considéré divisé par le nombre de structures affectées à ce banc mémoire. Il choisit les meilleures affectations pour l'enfant et efface de chaque parent les structures déjà assignées. Finalement, afin d'assigner les structures de données qui ne le sont pas encore, le banc mémoire choisi est celui qui donne la plus petite somme des coûts des conflits ouverts.

EvoMemexBasic considère que la population s'améliore si le coût des conflits des solutions décroît à chaque itération, et si la population conserve une diversité suffisante pour assurer une couverture satisfaisante de l'espace des solutions. De ce fait, un enfant n'est accepté dans la population que si la distance à ses parents est supérieure à un certain seuil. La distance entre deux solutions s_a et s_b est définie par le nombre minimum de structures de données dont l'assignation doit changer pour passer de s_a à s_b. Cette définition est couramment utilisée en coloration de graphes [5].

Pour garantir l'amélioration et la diversité de la population, l'algorithme met à jour la population à chaque itération. Nous utilisons la variance statistique des coûts de conflits des solutions comme mesure de diversité de la population. Si cette variance est inférieure à notre seuil nous générons une nouvelle population aléatoirement. Dans chaque actualisation de population nous remplaçons les p

Algorithme 3 : **Croisement-EvoMemexBasic.**

Données : r parents
Résultat : Un enfant.
pour $j = 1, \ldots, m$ **faire**
  pour *chaque parent dans* $1, \ldots, r$ **faire**
    Calculer le taux de conflit du banc mémoire b_j
  fin
  Sélectionner les structures de données du parent avec le plus petit taux de conflit
  Assigner ces structures de données au banc memoire b_j pour construire le nouvel enfant
  pour *chaque parent dans* $1, \ldots, r$ **faire**
    Effacer les structures de données assignées au nouvel enfant
  fin
fin
Assigner les structures de données restantes de façon à minimiser le plus petit coût.

parents (solutions) dont le coût est le plus élevé, par les p enfants. Cette forme de mise à jour présente un bon compromis entre la diversité et la qualité de la population.

4 Résultats expérimentaux et discussion

Cette partie présente les aspects relevant de l'implémentation des algorithmes TabuMemexBasic et EvoMemexBasic, ainsi que les résultats obtenus sur un ensemble d'instances avec un PC basé sur un processeur Intel Pentium IV cadencé à 3 GHz, et 1 Go de mémoire vive (RAM).

L'algorithme TabuMemexBasic change itérativement la taille de la liste taboue toutes les N itérations de manière aléatoire en fonction de $a + N \times t$, où a est un nombre entier fixe et t est un nombre aléatoire entre 0 et 2. Nous avons choisi expérimentalement $N = 50$ et $a = 10$ pour notre méthode, et un nombre d'itérations maximum $Niter = 10000$.

Pour l'implémentation de EvoMemexBasic, nous avons repris les valeurs des paramètres issus des tests effectués dans [12] : une population de 15 éléments, le croisement avec $r = 3$ parents, la génération de $p = 3$ enfants, et un seuil d'acceptation de $R = 0,1 \times n$ pour la distance minimale entre deux éléments de la population. Dans notre expérience, nous avons fixé une limite de 100 itérations comme critère d'arrêt et un minimum de $0,3$ pour la variance de la population pour cette méthode.

La table 1 présente les résultats obtenus avec les paramètres fixés ci-dessus pour un ensemble d'instances. Ces instances proviennent de problèmes de coloration des graphes de DIMACS [11] que nous avons modifiés en donnant des coûts aléatoires aux conflits, sauf pour la première qui est une instance réelle provenant d'un problème d'électronique. Cette table donne le coût des meilleures solutions obtenues avec les deux métaheuristiques présentées ainsi qu'avec la formulation PLNE résolue avec GLPK. Le temps de calcul, exprimé en secondes, est reporté pour chaque méthode. Les deux premières colonnes présentent des informations de base sur les instances comme le nom, le nombre des structures de données, de conflits et de bancs mémoire. La dernière colonne indique, pour chaque instance, le statut (optimal ou non) de la meilleure solution retournée par GLPK après une heure de calcul.

Les résultats de cette expérience montrent que EvoMemexBasic trouve toujours la solution optimale lorsque le PLNE peut être résolu de façon optimale dans la limite d'une heure de calcul.

TabuMemexBasic est plus rapide que EvoMemexBasic, ce qui est normal puisqu'elle constitue un sous-programme de EvoMemexBasic. Il faut noter que la meilleure solution retournée par GLPK lorsque la recherche de la solution optimale n'a pas abouti après une heure est généralement de mauvaise qualité comparée aux résultats des deux métaheuristiques. Pour l'instance `r125.1c`, GLPK n'a pas obtenu de solution entière, même après une heure de calcul.

TABLE 1 – Résultats

Instances		TabuMemexBasic		EvoMemexBasic		GLPK		Optimal
Nom	$n\backslash o\backslash m$	coût	temps	coût	temps	coût	temps	
`mpeg2enc`	180 \227 \2	**32,09**	1,60	**32,09**	3,20	**32,09**	107,79	oui
`queen5_5`	25 \160 \3	**974**	0,73	**974**	3,68	**974**	492,38	oui
`queen6_6`	36 \290 \4	999	1,13	999	5,73	1253	3599,29	non
`queen7_7`	49 \476 \4	1896	1,46	1896	16,10	2430	3600,02	non
`queen8_8`	64 \728 \5	1617	2,06	1617	54,12	2443	3600,01	non
`myciel3`	11 \20 \2	**146**	0,26	**146**	1,82	**146**	0,03	oui
`myciel4`	23 \71 \3	**69**	0,52	**69**	2,47	**69**	1,16	oui
`myciel5`	47 \236 \3	591	0,81	591	3,60	591	3599,41	non
`myciel6`	95 \755 \2	9017	1,26	9017	18,52	9963	3600,43	non
`myciel7`	191 \2360 \4	2262	2,21	2262	55,93	4642	3607,71	non
`r125.1`	125 \209 \3	346	1,22	346	28,87	425	3599,73	non
`r125.1c`	125 \7501 \23	2719	11,27	2685	135,24	-	3820,08	non
`r125.5`	125 \3838 \18	785	8,58	734	156,97	1668	3648,99	non
`mug88_1`	88 \146 \2	**967**	1,09	**967**	15,72	**967**	157,23	oui
`mug88_25`	88 \146 \2	**881**	1,07	**881**	16,27	**881**	53,11	oui
`mug100_1`	100 \166 \2	1149	1,19	**1129**	26,79	**1129**	957,19	oui
`mug100_25`	100 \166 \2	**1142**	1,24	**1142**	18,54	**1142**	562,00	oui

5 Conclusion

Cet article présente deux métaheuristiques, EvoMemexBasic basée sur un algorithme hybride à base de population et TabuMemexBasic basée sur une recherche taboue. Il nous a semblé pertinent d'adapter ces algorithmes issus de la coloration de graphes car le problème abordé peut être vu comme un *k-weighted graph coloring problem*. Les meilleurs résultats sont obtenus avec EvoMemexBasic qui reprend les principales caractéristiques d'Evocol : un contrôle rigoureux de la diversité de la population et un croisement *multi-parent*. TabuMemexBasic se distingue d'une recherche taboue classique par la taille variable de la liste taboue.

Dans la table 1, nous avons comparé les résultats obtenus pour les métaheuristiques avec une formulation exacte résolue avec GLPK. Les résultats sont encourageants, et laissent supposer que les solutions retournées sont de très bonne qualité même pour les instances de grande taille pour lesquelles la solution optimale n'est pas connue.

L'utilisation de méthodes issues de la coloration de graphes pourrait ainsi être étendue avec succès à des modèles plus complexes de l'allocation de mémoire pour les système embarqués, ce qui permettrait d'apprécier les gains apportés par l'utilisation de ces méthodes à des cas concrets en terme de consommation énergétique.

Références

[1] D. Atienza, S. Mamagkakis, F. Poletti, J. Mendias, F. Catthoor, L. Benini, D. Soudris, Efficient system-level prototyping of power-aware dynamic memory managers for embedded systems. Integration, the VLSI Journal, 39(2), pp. 113-130 (2006).

[2] A. Chimientia, L. Fanucci, R. Locatellic, S. Saponarac, VLSI architecture for a low-power video codec system, Microelectronics Journal, 33(5), pp. 417-427, (2002).

[3] P. Coussy, E. Casseau, P. Bomel, A. Baganne, E. Martin, A formal method for hardware IP design and integration under I/O and timing constraints, ACM Transactions on Embedded Computing Systems, 5(1), pp. 29-53 (2006).

[4] R. Diestel, Graph Theory, Volume 98 of Graduate Texts in Mathematics. Springer-Verlag, Heidelberg (2005).

[5] P. Galinier, J-K. Hao. Hybrid Evolutionary Algorithms for Graph Coloring. Journal of Combinatorial Optimization 3, 379-397 (1999).

[6] GLPK (GNU Linear Programming Kit). URL : http ://www.gnu.org/software/glpk/

[7] M. Iverson, F. Ozguner, L. Potter, Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment, IEEE Transactions on Computers, 48(12), pp. 1374-1379 (1999).

[8] N. Julien, J. Laurent, E. Senn, E. Martin, Power consumption modeling and characterization of the TI C6201, IEEE Micro 23(5), pp. 40-49 (2003).

[9] P. Lacomme, C. Prins, M. Sevaux. Algorithmes de Graphes. Eyrolles, Paris (2003).

[10] W. Lee, M. Chang, A study of dynamic memory management in C++ programs, Computer Languages Systems & Structures, 28(3), pp. 237-272 (2002).

[11] D. Porumbel, DIMACS Graphs : Benchmark Instances and Best Upper Bounds, http ://www.info.univ-angers.fr/pub/porumbel/graphs/.

[12] D. Porumbel, J-K Hao, P Kuntz, Diversity control and multi-parent recombination for evolutionary graph coloring algorithms, Evolutionay Computation in Combinatorial Optimization, 9th Proceedings of the European Conference, EvoCOP 2009 Tübingen, Germany, pp. 121-132, (2009).

[13] K. Sörensen. A framework for robust and flexible optimization using metaheuristics. PhD thesis Universiteit Antwerpen (2003).

[14] T. Vredeveld, J.K Lenstra. On local search for the generalized graph coloring problem. Operations Research Letters vol. 31, pp. 28-34 (2003)

[15] S. Wuytack, F. Catthoor, L. Nachtergaele, H. De Man, Power exploration for data dominated video application, Proceedings of IEEE Symposium on Low Power Design, pp. 359-364 (1996).

[16] Dash Optimization joins FICO. URL : http ://www.dashoptimization.com/