Conception d'algorithmes hybrides pour l'optimisation de l'énergie mémoire dans les systèmes embarqués et de fonctions multimodales

Maha Idrissi Aouad

Nancy-Université
Université Henri Poincaré

Département de formation doctorale en informatique **École doctorale IAEM Lorraine**
UFR STMIA

Conception d'Algorithmes Hybrides pour l'Optimisation de l'Énergie Mémoire dans les Systèmes Embarqués et de Fonctions Multimodales

THÈSE

présentée et soutenue publiquement le 4 Juillet 2011

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1

(spécialité informatique)

par

Maha Idrissi Aouad

Composition du jury

Président :	Stephan Merz	Directeur de Recherche, INRIA Nancy - Grand Est
Rapporteurs :	Annie Choquet-Geniet	Professeur, Université de Poitiers
	Abderrafiâa Koukam	Professeur, UT de Belfort-Montbéliard
Examinateurs :	Lhassane Idoumghar	Maître de Conférences, Université de Haute-Alsace
	Abdellatif Miraoui	Professeur, UT de Belfort-Montbéliard
	Pierre-Alain Muller	Professeur, Université de Haute-Alsace
Directeur de thèse :	René Schott	Professeur, Université Henri Poincaré - Nancy 1
Co-encadrant :	Olivier Zendra	Chargé de Recherche, INRIA Nancy - Grand Est

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Loria
Laboratoire lorrain de recherche en informatique et ses applications

Mis en page avec la classe thloria.

Nancy-Université
Université Henri Poincaré

Département de formation doctorale en informatique **École doctorale IAEM Lorraine**
UFR STMIA

Conception d'Algorithmes Hybrides pour l'Optimisation de l'Énergie Mémoire dans les Systèmes Embarqués et de Fonctions Multimodales

THÈSE

présentée et soutenue publiquement le 4 Juillet 2011

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1

(spécialité informatique)

par

Maha Idrissi Aouad

Composition du jury

Président :	Stephan Merz	Directeur de Recherche, INRIA Nancy - Grand Est
Rapporteurs :	Annie Choquet-Geniet	Professeur, Université de Poitiers
	Abderrafiâa Koukam	Professeur, UT de Belfort-Montbéliard
Examinateurs :	Lhassane Idoumghar	Maître de Conférences, Université de Haute-Alsace
	Abdellatif Miraoui	Professeur, UT de Belfort-Montbéliard
	Pierre-Alain Muller	Professeur, Université de Haute-Alsace
Directeur de thèse :	René Schott	Professeur, Université Henri Poincaré - Nancy 1
Co-encadrant :	Olivier Zendra	Chargé de Recherche, INRIA Nancy - Grand Est

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Loria
Laboratoire lorrain de recherche en informatique et ses applications

Mis en page avec la classe thloria.

Remerciements

À travers ce mémoire de doctorat au sein du Laboratoire Lorrain de Recherche en Informatique et ses Applications (LORIA), je présente mes sincères remerciements :

– À la Directrice du LORIA et ancienne responsable de l'équipe TRIO, Mme Françoise SIMONOT-LION, Professeur à l'École des Mines de Nancy, pour ses conseils avisés et le temps qu'elle m'a accordé,

– À mon Directeur de thèse, M. René SCHOTT, Professeur à l'Université Henri Poincaré - Nancy 1, pour sa disponibilité, son écoute, sa réactivité, ses conseils qui m'ont été d'une grande utilité,

– À M. Lhassane IDOUMGHAR, Maître de Conférences à l'Université de Haute-Alsace, pour ses conseils et le travail de recherche effectué avec lui,

– À M. Olivier ZENDRA, Chargé de Recherche à l'INRIA Nancy - Grand Est, pour son travail d'encadrement sur le projet MORE,

– À Mme Annie CHOQUET-GENIET, Professeur à l'Université de Poitiers et M. Abderrafiâa KOUKAM, Professeur à l'Université de Technologie de Belfort-Montbéliard, pour avoir accepté de rapporter sur ma thèse et pour l'intérêt qu'ils ont manifesté pour mon travail,

– À M. Stephan MERZ, Directeur de Recherche et responsable de l'équipe VeriDis à l'INRIA Nancy - Grand Est, M. Abdellatif MIRAOUI, Professeur à l'Université de Technologie de Belfort-Montbéliard et M. Pierre-Alain MULLER, Professeur à l'Université de Haute-Alsace, pour avoir participé à mon jury de thèse,

– À Mme Laurence BENINI, pour sa gentillesse, son aide logistique dans les missions et sa bonne humeur quotidienne,

– À tous les membres de l'équipe TRIO, et plus particulièrement à tous mes collègues de bureau pour la bonne ambiance qui y règne,

– À mes parents pour leur présence de tous les jours à mes côtés, leurs encouragements ainsi que pour leur soutien moral qui fut pour moi la meilleure motivation,

– À mon mari, mes sœurs ainsi que toute ma famille pour leurs encouragements durant la période de ma thèse ainsi que pour tous les moments agréables passés à leurs côtés.

Je dédie cette thèse
à mes parents,
à mon mari,
à mes sœurs,
ainsi qu'à toute ma famille.
Et plus particulièrement,
à mes grand-mères.
Elles auraient, sans doute, été fières de moi.

Table des matières

Partie II Contributions

Liste de mes publications

Article publié dans un journal spécialisé :

[1] L. Idoumghar, M. Idrissi Aouad, M. Melkemi and R. Schott, *Hybrid PSO-SA Type Algorithms for Multi-Modal Function Optimization and Reducing Energy Consumption in Embedded Systems*. Applied Computational Intelligence and Soft Computing. Volume 2011, Article ID 138078, 12 pages, 2011.

Conférences internationales avec actes et comité de programme :

[2] M. Idrissi Aouad, L. Idoumghar, R. Schott and O. Zendra, *Sequential and Cooperative Distributed SA-Type Algorithms for Energy Optimization in Embedded Systems*. Proceedings of IEEE-CiSE 2010 (International Conference on Computational Intelligence and Software Engineering). Wuhan, Chine, Décembre 2010. Volume 1.

[3] M. Idrissi Aouad, L. Idoumghar, R. Schott and O. Zendra, *Sequential and Distributed Hybrid GA-SA Algorithms for Energy Optimization in Embedded Systems*. Proceedings of IADIS Applied Computing 2010. Timisoara, Roumanie, Octobre 2010.

[4] M. Idrissi Aouad, L. Idoumghar, R. Schott and O. Zendra, *Reduction of Energy Consumption in Embedded Systems : A Hybrid Evolutionary Algorithm*. Proceedings of META 2010 (3rd International Conference on Metaheuristics and Nature Inspired Computing). Djerba, Tunisie, Octobre 2010. Volume 95.

[5] L. Idoumghar, M. Idrissi Aouad, M. Melkemi and R. Schott, *Metropolis Particle Swarm Optimization Algorithm with Mutation Operator For Global Optimization Problems*. Proceedings of IEEE-ICTAI 2010 (22th International Conference on Tools with Artificial Intelligence). Arras, France, Octobre 2010. Pages 35-42.

[6] M. Idrissi Aouad, R. Schott and O. Zendra, *Hybrid Heuristics for Optimizing Energy Consumption in Embedded Systems*. Proceedings of ISCIS 2010 (25th International Symposium on Computer and Information Sciences). Londres, Royaume-Uni, Septembre 2010. Volume 62, pages 409-414.

[7] M. Idrissi Aouad, R. Schott and O. Zendra, *Genetic Heuristics for Reducing Memory Energy Consumption in Embedded Systems*. Proceedings of ICSOFT 2010 (5th International Conference on Software Engineering and Data Technologies). Athènes, Grèce, Juillet 2010. Volume 2, pages 394-402.

[8] M. Idrissi Aouad, R. Schott and O. Zendra, *A Tabu Search Heuristic for Scratch-Pad Memory Management*. Proceedings of ICSET 2010 (International Conference on Software Engineering and Technology). Rome, Italie, Avril 2010. Volume 64, pages 386-390, WASET pub.

Workshop international avec actes et comité de programme :

[9] M. Idrissi Aouad et O. Zendra, *A Survey of Scratch-Pad Memory Management Techniques for low-power and -energy*. In 2nd ECOOP Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2007), pages 31-38. Berlin, Allemagne, Juillet 2007.

Poster :

[10] M. Idrissi Aouad et O. Zendra, *Memory Management and Data Placement Optimizations for Low-Power*. Poster. In Colloque GDR SOC-SIP. Paris, Avril 2008 et in Advanced Computer Architecture and Compilation for Embedded Systems (ACACES'08). Italie, Juillet 2008.

Rapport technique :

[11] M. Idrissi Aouad et O. Zendra, *Outils de caractérisation du comportement mémoire et d'estimation de la consommation énergétique*. Rapport technique INRIA RT-0350, Février 2008.

Table des figures

Liste des tableaux

Introduction générale

Les systèmes embarqués envahissent notre vie quotidienne souvent de manière totalement transparente. Ces systèmes sont soumis à des critères de performances (optimisation de la réponse du système à des sollicitations extérieures) et, pour les plus critiques, à des contraintes temps-réel (temps de réponse à des sollicitations bornées). Les systèmes embarqués concernés sont très nombreux. Des systèmes de ce type peuvent être rencontrés dans le domaine des transports (ferroviaire, automobile, avionique), dans le domaine spatial, en médecine, en industrie, en télécommunications ou dans le cadre du contrôle de procédés industriels. Le spectre des applications potentielles est donc très large.

Le terme "embarqué" sous-entend une autonomie d'alimentation. Ces systèmes sont donc censés être autonomes en énergie. Or, malgré les avancées importantes dans le domaine des batteries, de nombreux systèmes restent très dépendants de l'énergie disponible et ont une autonomie encore trop limitée au goût de leurs utilisateurs. Citons par exemple : les téléphones portables, les PDAs [1], les ordinateurs portables, les consoles, *etc*. Cependant, bien que n'étant pas embarqués et étant alimentés en électricité, les gros systèmes comme les super calculateurs ont également leurs lots de problèmes liés à l'énergie. En effet, leur consommation énergétique est très forte, ce qui provoque des pics de puissance ainsi que des problèmes d'échauffement nécessitant la mise en place de systèmes de refroidissement ou de climatisation, généralement coûteux et encombrants.

La consommation énergétique de ce type de systèmes est un problème sensible et sa réduction est devenue une nécessité pour autoriser le déploiement de nouveaux produits. Les ressources mémoire peuvent sur ce plan avoir un impact considérable, tout particulièrement pour des applications qui traitent de grosses quantités de données (multimédia, traitement du signal, *etc*). Du fait que ces systèmes embarqués intègrent de plus en plus de fonctionnalités complexes (MMS vidéo, vidéo à la demande, télévision numérique TNT, audio, Internet, *etc*.) nécessitant de plus en plus de mémoire, celle-ci devient une cause principale de la consommation abusive d'énergie. Ainsi, des études affirment qu'en 2022, la consommation énergétique en mémoire représentera environ 75.4% de la consommation totale d'un système embarqué [ITRS, 2007]. Des chiffres alarmants qui nous incitent à réfléchir et à proposer des solutions afin de résoudre ce problème.

Dans ce cadre, cette thèse consiste à aborder le problème de la consommation d'énergie dans les systèmes embarqués temps réels d'un point de vue logiciel, en tenant compte de l'architecture sous-jacente. Bien que le problème de l'énergie a été souvent considéré d'un point de vue matériel, il est bien moins exploré du point de vue logiciel, où les optimisations et les stratégies de gestion mémoire se sont plutôt focalisées sur la vitesse ou la taille. Pourtant, des opportunités considérables existent à ce niveau, car les optimisations énergétiques peuvent souvent être meilleures lorsqu'elles sont faites au niveau logiciel (compilateur, gestion mémoire, ordonnanceur, *etc*) plutôt qu'au niveau matériel. En effet, dans le pre-

1. PDA : *Personal Digital Assistant.*

mier cas, la logique de l'optimisation est faite en partie lors de la compilation, sans aucun surcoût à l'exécution, alors que dans le matériel cette logique correspond à des circuits qui utilisent de l'énergie. Ceci permet de dédier beaucoup plus de ressources (temps CPU [2], mémoire) à la compilation d'un système, ce qui est impossible dans le matériel. Un contexte plus large et plus complet peut donc être pris en compte, ce qui permet au compilateur de "connaître le futur" de l'exécution d'un programme ou d'un ensemble de programmes, tout particulièrement en ce qui concerne son comportement mémoire. Bien entendu, les meilleurs résultats sont obtenus en combinant optimisations logicielles et matérielles, ou, au moins, en prenant en compte le matériel dans les optimisations logicielles. C'est pourquoi, nous allons nous focaliser sur les techniques et solutions logicielles assistées par le compilateur.

Dans un autre contexte, l'optimisation globale et plus particulièrement l'optimisation des fonctions tests mathématiques principalement multimodales est également un problème important. En effet, la qualité de la plupart des méthodes d'optimisation globale est fréquemment évaluée en utilisant ces fonctions tests standards de la littérature. Cette série de problèmes spécifiquement créés pour tester la performance des algorithmes d'optimisation est majoritairement multimodale, $i.e.$ de dimension élevée, avec un nombre élevé d'extrema locaux. Ces fonctions sont considérées comme étant des problèmes de test très difficiles. Trouver le minimum global de ces fonctions revêt une difficulté d'autant plus grande que le comportement de la procédure d'optimisation est généralement justifié, expliqué et soutenu par les intuitions humaines sur une surface $2D$. Or, des problèmes d'optimisation à deux dimensions apparaissent très rarement dans la pratique. En réalité, ce sont des problèmes d'optimisation avec un grand nombre de dimensions. Par exemple, *ft10*, le plus petit programme test connu actuellement appelé également *job shop scheduling problem* a une dimension de 90, alors que le plus grand connu a une dimension de 1980 ! Par conséquent, afin de tester la qualité réelle de tout algorithme d'optimisation proposé, il convient d'utiliser ces fonctions tests mathématiques.

Dans ce mémoire, nous tenterons de résoudre ces deux problèmes d'optimisation. La première partie abordera les deux problèmes et donnera notre positionnement par rapport à ce qui se fait dans les deux domaines. Le chapitre 1 présentera plus en détails les deux problématiques, leurs contextes, les motivations qui nous ont poussées à nous y intéresser ainsi que leurs enjeux. Le chapitre 2 quant-à-lui, décrira différentes méthodes et techniques utilisées dans la littérature pour répondre aux deux problématiques. La deuxième partie s'intéressera aux contributions proprement dites. Le chapitre 3 se focalisera sur la réduction de la consommation d'énergie en mémoire dans les systèmes embarqués et proposera des approches nouvelles tentant de remédier à ce problème. En revanche, le chapitre 4 s'intéressera aux deux problèmes d'optimisation et présentera de nouveaux algorithmes hybrides et distribués réalisant de meilleures performances que les méthodes utilisées dans la littérature.

2. CPU : *Central Processing Unit.*

Première partie

Positionnement

Chapitre 1

Description des problématiques

Sommaire

1.1 Réduction de l'énergie consommée en mémoire dans les systèmes embarqués

1.1.1 Motivation

Les systèmes embarqués se trouvent partout. Il est intéressant de constater à quel point nous les utilisons de manière journalière sans y prêter attention. À la maison : TV, cafetière, lecteur DVD, console de jeux, *etc*. Au travail : téléphone portable, ordinateur portable, PDA [3], *etc*. Autant de systèmes où la malédiction de la batterie vide est un problème courant. À cause de l'évolution de la technologie, ces systèmes doivent intégrer de plus en plus de fonctionnalités complexes (vidéo, audio, Internet, vidéophonie, *etc*.) nécessitant une mémoire de plus en plus grande et donc une batterie plus grande.

Ainsi, la mémoire deviendra l'élément le plus consommateur d'énergie dans un système embarqué. En effet, les tendances dans [ITRS, 2007] montrent que la consommation des systèmes intégrés (*SoC*) est dominée par la consommation mémoire dynamique et statique (voir figure 1.1).

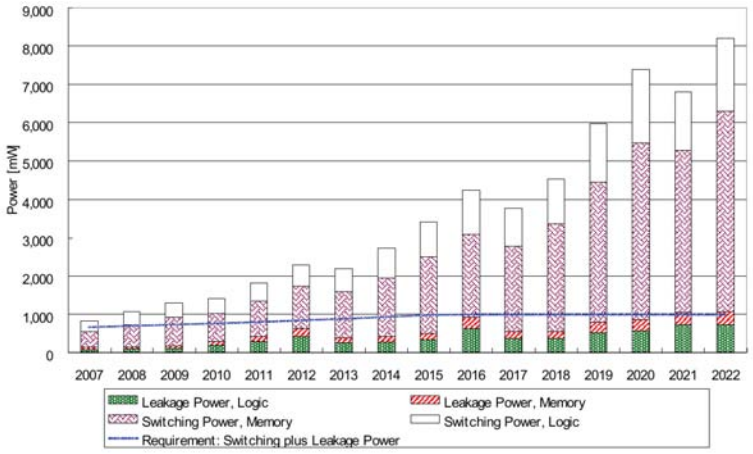

FIGURE 1.1 – Tendances de la consommation énergétique des SoCs [ITRS, 2007].

De ce fait, en 2022, la consommation mémoire représentera environ 75.4% de la consommation totale d'un système intégré [ITRS, 2007]. De plus, toujours selon les tendances données dans [ITRS, 2007], la mémoire occupera une place plus grande dans un système intégré comme on peut le constater à travers la figure 1.2. Une réelle invasion des systèmes contraints par l'énergie au point que la réduction de la consommation de l'énergie mémoire dans les systèmes embarqués n'a jamais été si cruciale et autant d'actualité.

3. PDA : *Personal Digital Assistant.*

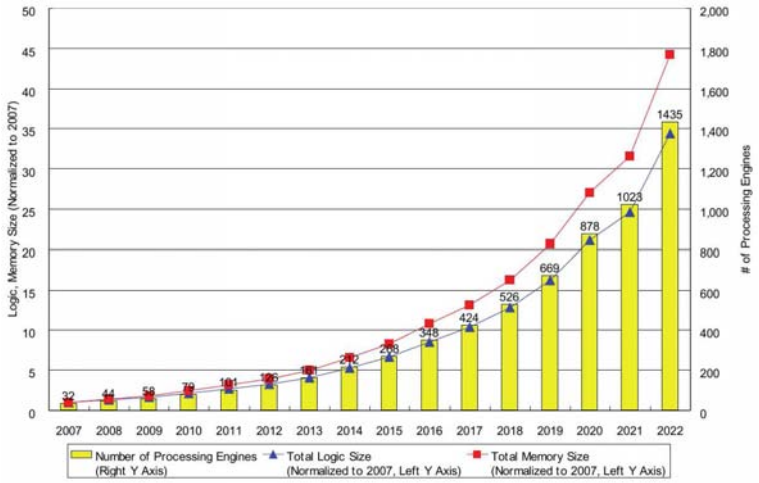

FIGURE 1.2 – Tendances de la complexité de conception des SoCs [ITRS, 2007].

1.1.2 Contexte

Ma thèse se situe dans le cadre du projet ANR [4] MORE [5]. L'objectif de ce projet, conjoint entre le LORIA, l'IRIT [6] et le LIP6 [7], est de définir des stratégies de recherche de compromis entre trois critères : consommation énergétique, occupation mémoire et temps d'exécution pire-cas (WCET [8]). Le LORIA devant se focaliser sur l'aspect énergie, un des trois critères du projet MORE. Cela consiste à attaquer le problème de l'utilisation énergétique dans les systèmes embarqués temps-réel d'un point de vue logiciel, en travaillant sur la gestion mémoire et en considérant l'architecture matérielle sous-jacente.

Diverses options existent pour économiser l'énergie et pour augmenter l'autonomie. Ces différentes approches peuvent être divisées en deux catégories : optimisations matérielles et optimisations logicielles. Les techniques d'optimisations matérielles, hors du contexte de mon doctorat, ne seront pas traitées dans ce mémoire, mais une quantité importante de littérature les concernant est disponible (voir les premières parties de [Graybill et Melhem, 2002]). Dans la suite des chapitres, nous nous focaliserons sur les techniques logicielles assistées par le compilateur. Dans ce cadre, les *Scratch-Pad Memories* (*SPMs*) ou mémoires Scratch-Pad présentent un intérêt considérable de par la facilité et la certitude avec lesquelles leur comportement (temps de réponse notamment) peut être prédit. Il faudra donc les intégrer dans notre plate-forme.

1.1.3 Composants de la hiérarchie mémoire

Avant d'aller plus loin, il convient de présenter, brièvement, les différents comportements énergétiques des principaux composants de la hiérarchie mémoire. En effet, une hiérarchie mémoire se compose principalement d'une mémoire cache, d'une mémoire Scratch-Pad et d'une mémoire principale.

4. ANR : Agence Nationale de la Recherche.
5. MORE : *Multicriteria Optimizations for Real-time Embedded systems.*
6. IRIT : Institut de Recherche en Informatique de Toulouse.
7. LIP6 : Laboratoire d'Informatique de Paris 6.
8. WCET : *Worst-Case Execution Time.*

1.1.3.1 Mémoire Cache

La mémoire cache est une mémoire rapide, de petite taille. Elle contient les mots mémoire les plus récemment utilisés accélérant ainsi l'accès à ces mots. Lorsqu'un pourcentage important de mots nécessaires se trouve en cache, la latence effective de la mémoire est fortement réduite. Pour améliorer à la fois la bande passante et la latence, on peut recourir à des caches multiples. Une technique de base très efficace, qualifiée de caches séparés, consiste à introduire deux caches distincts : un pour les instructions et un autre pour les données. Les opérations mémoire peuvent donc être réalisées indépendamment dans chaque cache, doublant la bande passante du système de mémoire [Tanenbaum, 2005]. D'après [Ben Fradj *et al.*, 2005], la mémoire cache est une mémoire qui consomme moins d'énergie que la mémoire principale (*DRAM* [9]) mais plus que la mémoire Scratch-Pad (*SPM*). Selon le choix de l'utilisateur, nous pouvons rencontrer différentes configurations possibles de mémoires cache ; un cache de données et un cache d'instructions, un cache de données seulement, un cache d'instructions seulement ou encore un cache unifié. Il est à noter que de nombreux processeurs contiennent une mémoire cache.

Bien que les mémoires cache améliorent la rapidité du programme, elles ne sont pas toujours adéquates dans le cas des systèmes embarqués : elles augmentent la taille du système ainsi que son coût énergétique (surface de la mémoire cache plus sa gestion logicielle).

1.1.3.2 Mémoire Scratch-Pad

Les mémoires Scratch-Pad (*SPMs*) présentent un intérêt considérable de par la facilité et la certitude avec lesquelles leur comportement (temps de réponse notamment) peut être prédit. La SPM est une petite zone mémoire SRAM [10] rapide qui ressemble à la mémoire cache mais qui est gérée au niveau logiciel, soit par le développeur, soit par le compilateur, alors que les mémoires cache nécessitent une circuiterie dédiée. Comparée à la mémoire cache, la mémoire Scratch-Pad possède plusieurs avantages [Zendra, 2006; Idrissi Aouad et Zendra, 2007]. En effet, la SPM consomme jusqu'à 40% moins d'énergie et 34% moins de surface que la mémoire cache [Banakar *et al.*, 2002]. De plus, le temps d'exécution d'une SPM utilisant un simple algorithme d'allocation statique de type sac à dos est 18% meilleur comparé à une mémoire cache [Banakar *et al.*, 2002]. [Ben Fradj *et al.*, 2005] ont également montré l'efficacité d'utiliser une SPM dans une architecture mémoire où une économie de consommation de 35% est réalisée en comparaison avec une architecture mémoire sans aucune SPM. De plus, son coût est plus bas et sa gestion logicielle la rend plus prévisible, ce qui est une caractéristique importante des systèmes embarqués temps-réel.

D'après [Adiletta *et al.*, 2002; Brash, 2002], une grande variété de puces comprenant une SPM est disponible aujourd'hui sur le marché. Les exemples de familles de processeurs embarqués contenant une SPM sont nombreux : Motorola MPC500, Analog Devices ADSP-21XX, Philips LPC2290, Atmel AT91-C140, ARM 968E-S, Hitachi M32R-32192, Infineon XC166, Analog Devices ADSP-TS201S, Hitachi SuperH-SH7050, Motorola Dragonball et tant d'autres [Nguyen *et al.*, 2007b]. Selon [Nguyen *et al.*, 2007a], il y a au moins 80 processeurs embarqués de ce type ayant une SPM et une mémoire principale (*DRAM*) directement accédées par le CPU [11], mais sans aucune mémoire cache. De plus, les tendances dans [LCTES, 2003] indiquent que la prévalence des mémoires SPM dans les systèmes embarqués est susceptible de continuer dans le futur.

9. DRAM : *Dynamic Random Access Memory.*
10. SRAM : *Static Random Access Memory.*
11. CPU : *Central Processing Unit.*

D'ailleurs, plusieurs auteurs ont essayé de tirer profit des avantages des SPMs et différentes directions de recherches ont été explorées [Absar et Catthoor, 2005; Angiolini *et al.*, 2003; Absar et Catthoor, 2006; Kandemir *et al.*, 2002]. Ces techniques et algorithmes, synthétisés dans [Zendra, 2006] et [Benini et Micheli, 1999], essayent d'allouer, de façon optimale, les données et/ou le code d'une application à la SPM afin de réduire la consommation énergétique des systèmes embarqués. Le lecteur intéressé peut voir [Benini et Micheli, 2000] pour une liste de références bibliographiques plus exhaustive. Nous allons donc, bien évidemment, utiliser les mémoires SPMs dans notre architecture mémoire.

1.1.3.3 Mémoire Principale

La mémoire principale (*DRAM*) est la plus grande en terme de superficie, mais c'est aussi la plus gourmande en consommation d'énergie comme c'est démontré dans [Ben Fradj *et al.*, 2005] où les énergies consommées dans les trois mémoires : cache, SPM et DRAM sont comparées. Notons également que la DRAM fait perdre beaucoup de temps d'exécution car l'accès y est très lent.

1.1.4 Modèle d'estimation de la consommation d'énergie en mémoire

Afin d'estimer les économies d'énergie réalisées, je propose dans cette section un modèle d'estimation général de la consommation d'énergie en mémoire. Mais avant, il convient d'expliquer brièvement le comportement de la mémoire cache. Dans [Shiue et Chakrabarti, 1999], un modèle d'estimation de la consommation d'énergie de la mémoire cache est proposé. Cependant, ce modèle est très simple car il ne prend en compte que les lectures. Dans ce qui suit, les équations du comportement de la mémoire cache sont détaillées car d'une part, elles vont être utilisées pour estimer la consommation de la mémoire cache en considérant différentes politiques d'écriture et d'autre part, le mécanisme de fonctionnement du cache est complexe.

1.1.4.1 Comportement de la mémoire cache

La mémoire cache contient des copies de données qui sont en DRAM. Avant tout accès à la mémoire principale, le processeur vérifie si les données ne sont pas présentes dans le cache. Auquel cas, le processeur utilise les données contenues dans le cache et n'accède pas à la DRAM. Sinon, il est nécessaire d'aller chercher les données en mémoire principale. Ainsi, dans le cas d'un accès avec succès à la mémoire cache (*cache hit*), on compte juste l'énergie nécessaire pour cet accès. En revanche, dans le cas d'un accès avec échec à la mémoire cache (*cache miss*), le processeur procède à l'éviction d'une donnée de la mémoire cache selon une stratégie de remplacement (voir l'Annexe A pour plus de détails), et cela afin de libérer la place pour la donnée qui est récupérée de la DRAM puis recopiée en cache. Ainsi, l'énergie consommée dans le cas d'un accès avec échec est résumée par l'équation 1.1, où les termes sont expliqués dans le tableau 1.1 :

$$E_{cm} = E_{cr} + E_{dramr} + E_{dramw} + E_{cw} \tag{1.1}$$

Lors d'une écriture du programme en mémoire, dans le cas de l'existence d'une mémoire cache, il existe plusieurs façons appelées "politiques d'écriture" de gérer ces écritures, on parle alors d'écriture simultanée (*write-through*) et d'écriture différée (*write-back*).

TABLE 1.1 – Liste des termes énergétiques.

Terme	Signification
E_{dramr}	Énergie consommée lors d'une lecture en DRAM.
E_{dramw}	Énergie consommée lors d'une écriture en DRAM.
E_{cr}	Énergie consommée lors d'une lecture en cache d'instructions ou de données.
E_{cw}	Énergie consommée lors d'une écriture en cache d'instructions ou de données.

Écriture simultanée : Cette politique consiste à répercuter en mémoire centrale chaque écriture dans le cache. Chaque écriture dans le cache provoque alors une écriture en DRAM. Les termes des équations suivantes sont expliqués dans le tableau 1.1.

– Succès en lecture : les opérations de lecture avec succès sont satisfaites au moyen du cache :

$$E_{CacheReadHitWriteThrough} = E_{cr} \tag{1.2}$$

– Échec en lecture : lorsque le processeur essaie de lire un mot qui n'est pas dans le cache, son contrôleur de cache charge la ligne qui contient le mot dans le cache. La ligne est fournie par la mémoire qui, dans le cas de cette politique, est toujours actualisée :

$$E_{CacheReadMissWriteThrough} = E_{dramr} + E_{cw} \tag{1.3}$$

– Succès en écriture : lors d'un succès en écriture, le cache est mis à jour et le mot est également écrit dans la mémoire principale :

$$E_{CacheWriteHitWriteThrough} = E_{cw} + E_{dramw} \tag{1.4}$$

– Échec en écriture : lorsqu'un échec en écriture se produit, le mot qui doit être modifié est écrit dans la mémoire principale. La ligne qui contient le mot référencé n'est pas chargée dans le cache :

$$E_{CacheWriteMissWriteThrough} = E_{dramw} \tag{1.5}$$

Cette politique assure qu'à chaque opération d'écriture, le mot modifié est écrit dans la mémoire principale afin de maintenir celle-ci à jour en permanence [Tanenbaum, 2005].

Écriture différée : Cette politique retarde au maximum les écritures en mémoire centrale. Chaque ligne de cache possède un bit interne appelé *Dirty Bit* (DB). Lorsqu'une ligne de cache est modifiée, son DB est activé pour indiquer que la ligne de cache est à jour, mais que la mémoire ne l'est pas. La mémoire est corrigée au moment où la ligne qui contient ces données est évincée du cache et donc probablement après qu'un grand nombre d'écritures et/ou de lectures eurent été réalisées en cache [Tanenbaum, 2005].

– Succès en lecture : les opérations de lecture avec succès sont satisfaites au moyen du cache quelle que soit la valeur du *dirty bit* :

$$E_{CacheReadHitWriteBack} = E_{cr} \tag{1.6}$$

– Échec en lecture : lorsque le processeur essaie de lire un mot qui n'est pas dans le cache, le *dirty bit* est vérifié avant de procéder à l'éviction de la ligne concernée :

– S'il est à zéro, cela implique que la ligne de cache et la mémoire sont toutes les deux à jour. Le contrôleur de cache charge alors depuis la mémoire la ligne qui contient le mot et la met dans le cache :

$$E_{CacheReadMissWriteBack0} = E_{dramr} + E_{cw} \quad (1.7)$$

– Si le *dirty bit* est à un, cela implique que la mémoire n'est pas à jour, il faut la corriger :

$$E_{CacheReadMissWriteBack1} = E_{dramr} + E_{cw} + E_{cr} + E_{dramw} \quad (1.8)$$

– Succès en écriture : lors d'un succès en écriture, le cache est mis à jour quelle que soit la valeur du *dirty bit* :

$$E_{CacheWriteHitWriteBack} = E_{cw} \quad (1.9)$$

– Échec en écriture : lorsqu'un échec en écriture se produit, le *dirty bit* est vérifié avant de procéder à l'éviction de la ligne concernée :

– Si le *dirty bit* est à zéro, le mot qui doit être modifié est écrit dans le cache :

$$E_{CacheWriteMissWriteBack0} = E_{cw} \quad (1.10)$$

– Si le *dirty bit* est activé, la mémoire est mise à jour à partir du cache et le mot qui doit être modifié est ensuite écrit dans le cache :

$$E_{CacheWriteMissWriteBack1} = E_{cw} + E_{cr} + E_{dramw} \quad (1.11)$$

Afin de pouvoir utiliser ces équations dans une écriture mathématique plus concise, nous avons regroupé les différentes équations. Pour illustrer cela, voici à titre d'exemple le cas de l'opération de lecture. Dans le cas d'un succès en lecture, en regroupant les deux équations 1.2 et 1.6, nous obtenons l'équation suivante :

$$E_{CacheReadHit} = E_{cr} \quad (1.12)$$

Dans le cas d'un échec en lecture, en regroupant les équations 1.3, 1.7 et 1.8, nous obtenons l'équation suivante, où WP représente la politique d'écriture du cache considérée (1 pour l'écriture simultanée, 0 pour l'écriture différée) et DB représente le fait que le *dirty bit* est activé ou non (1 activé, 0 sinon) :

$$E_{CacheReadMiss} = E_{dramr} + E_{cw} + (1 - WP) * DB * (E_{cr} + E_{dramw}) \quad (1.13)$$

En regroupant les équations 1.12 et 1.13, nous pouvons traduire une opération de lecture par l'équation suivante, où h représente le type d'accès au cache (1 pour accès avec succès, 0 pour accès avec échec) :

$$E_{crt} = h * E_{cr} + (1 - h) * [E_{dramr} + E_{cw} + (1 - WP) * DB * (E_{cr} + E_{dramw})] \quad (1.14)$$

Les équations de l'opération d'écriture sont obtenues de la même façon. Maintenant que nous avons modélisé l'énergie consommée par un accès en mémoire cache, revenons à notre modèle d'estimation de la consommation d'énergie en mémoire.

1.1.4.2 Architecture mémoire considérée

En ce qui concerne la réduction de la consommation d'énergie en mémoire dans les systèmes embarqués, nous considérons pour nos expérimentations une architecture mémoire composée d'une mémoire Scratch-Pad (*SPM*), d'une mémoire principale (*DRAM*) et d'un cache d'instructions. Les instructions du code source des programmes tests sont allouées en cache d'instructions alors que les données sont allouées, soit en SPM, soit en DRAM, selon la politique de l'heuristique considérée. Nous utilisons dans notre simulation une configuration mémoire spécifique qui respecte certaines contraintes :

- La taille de la mémoire cache doit être supérieure à 64 octets d'après une contrainte du projet *MORE*.
- Les caractéristiques du cache de données et de la SPM doivent être les mêmes en vue de pouvoir comparer leurs performances énergétiques de façon équitable.
- Nous sommes dans un cas de systèmes embarqués donc il y a certaines valeurs technologiques à respecter.

C'est pourquoi, nous considérons les caractéristiques présentées dans le tableau 1.2. Nous avons ajouté l'exemple d'un cache de données en vue de montrer clairement la différence entre ce dernier et la SPM.

TABLE 1.2 – Caractéristiques des mémoires considérées.

Type de mémoire	Taille	Largeur des blocs	Associativité	Technologie
Cache d'instructions	128 *Ko*	16 *o*	1	40 *nm*
Mémoire DRAM DDR3	800 *Ko*		0	78 *nm*
Cache de données	1 *Ko*	8 *o*	0	40 *nm*
Mémoire SPM	1 *Ko*	8 *o*	0	40 *nm*

Du fait que la SPM est gérée de façon logicielle, il n'y a aucun accès avec échec puisqu'à tout moment on connaît l'emplacement des données en SPM. Le calcul de l'énergie consommée en SPM dépend alors du coût d'un seul accès à cette mémoire ainsi que du nombre total d'accès effectués à la SPM. Le calcul de l'énergie consommée en DRAM se fait de la même manière.

Pour notre architecture mémoire composée d'une SPM, d'un cache d'instruction et d'une DRAM, le modèle général de la consommation d'énergie est donné par l'équation 1.15 où les trois termes font référence à l'énergie totale consommée respectivement en SPM, en cache d'instruction et en DRAM.

$$E = E_{tspm} + E_{tic} + E_{tdram} \tag{1.15}$$

Ainsi, nous obtenons finalement la formule détaillée ci-dessous :

$$
\begin{aligned}
E &= N_{spmr} * E_{spmr} && (1.16)\\
&+ N_{spmw} * E_{spmw} && (1.17)\\
&+ \sum_{k=1}^{N_{icr}} [h_{i_k} * E_{icr} + (1 - h_{i_k}) * [E_{dramr} + E_{icw} \\
&\quad + (1 - WP_i) * DB_{i_k} * (E_{icr} + E_{dramw})]] && (1.18)\\
&+ \sum_{k=1}^{N_{icw}} [WP_i * E_{dramw} + h_{i_k} * E_{icw} + (1 - WP_i) * \\
&\quad (1 - h_{i_k}) * [E_{icw} + DB_{i_k} * (E_{icr} + E_{dramw})]] && (1.19)\\
&+ N_{dramr} * E_{dramr} && (1.20)\\
&+ N_{dramw} * E_{dramw} && (1.21)
\end{aligned}
$$

Les lignes (1.16) et (1.17) représentent respectivement l'énergie totale consommée lors d'une lecture et lors d'une écriture en SPM. Les lignes (1.18) et (1.19) représentent respectivement les énergies totales consommées lors d'une lecture et lors d'une écriture en cache d'instructions. Alors que les lignes (1.20) et (1.21) représentent respectivement l'énergie totale consommée lors d'une lecture et lors d'une écriture en DRAM. Les différents termes utilisés ci-dessus sont expliqués dans le tableau 1.3.

Notre objectif est de minimiser la formule de l'estimation de la consommation d'énergie de notre architecture mémoire. Dans cette architecture mémoire considérée, le cache d'instruction ne sert qu'à stocker les instructions de l'application et ne peut donc pas être utilisé au stockage des données de l'application. Et étant donné les avantages des deux mémoires restantes (SPM et DRAM), il est préférable de stocker le maximum de données possibles en SPM. Dans d'autres termes, dans la formule énergétique de la ligne (1.16) à la ligne (1.21), les termes N_{spmr} et N_{spmw} doivent être maximisés autant que possible. Ainsi, notre problème devient de maximiser les nombres d'accès à la SPM. Cela est expliqué dans la section qui suit.

1.1.5 Classification du problème d'optimisation

Notre problème relève de l'optimisation combinatoire. Vu de plus prêt, on constate que c'est un problème de type sac à dos (*knapsack*) [H. Kellerer et Pisinger, 2004].

Nous voulons remplir la mémoire Scratch-Pad, qui peut contenir une capacité maximale de C, avec une combinaison de données à partir d'une liste de N données possibles. Chaque donnée est caractérisée par sa taille $size_i$ et par le nombre de fois que l'on y accède : son nombre d'accès an_i. Soit s une solution au problème. s est alors une suite finie de N termes telle que $s[n]$ est soit 0, soit la taille de la n_{ieme} donnée. $s[n] = 0$ si et seulement si la n_{ieme} donnée n'est pas sélectionnée dans l'espace de la solution.

Le but est de maximiser la somme des nombres d'accès des données allouées en SPM. Cette fonction objectif est exprimée par l'équation 1.22.

$$Maximiser \sum_{i=1}^{N} an_i \frac{s[i]}{size_i} \qquad (1.22)$$

TABLE 1.3 – Liste des termes du modèle d'estimation de la consommation énergétique.

Terme	Signification
E_{spmr}	Énergie consommée lors d'une lecture en SPM.
E_{spmw}	Énergie consommée lors d'une écriture en SPM.
N_{spmr}	Nombre d'accès en lecture à la SPM.
N_{spmw}	Nombre d'accès en écriture à la SPM.
E_{icr}	Énergie consommée lors d'une lecture en cache d'instructions.
E_{icw}	Énergie consommée lors d'une écriture en cache d'instructions.
N_{icr}	Nombre d'accès en lecture au cache d'instructions.
N_{icw}	Nombre d'accès en écriture au cache d'instructions. Généralement, comme on ne modifie pas les instructions, ce terme est souvent à zéro.
E_{dramr}	Énergie consommée lors d'une lecture en DRAM.
E_{dramw}	Énergie consommée lors d'une écriture en DRAM.
N_{dramr}	Nombre d'accès directs en lecture à la DRAM.
N_{dramw}	Nombre d'accès directs en écriture à la DRAM.
WP_i	Politique d'écriture (*Write Policy*) du cache considérée (*write-through/write-back*). Dans le cas de la politique d'écriture simultanée (*write-through*), $WP_i = 1$. Dans le cas de la politique d'écriture différée (*write-back*), $WP_i = 0$.
DB_{i_k}	*Dirty Bit* utilisé dans le cas de la politique d'écriture différée pour indiquer lors de l'accès k si la ligne du cache d'instructions avait été modifiée auparavant ($DB_i = 1$) ou pas ($DB_i = 0$).
h_{i_k}	Type de l'accès k au cache d'instructions. Dans le cas d'un accès avec succès au cache d'instructions, $h_{i_k} = 1$. Dans le cas d'un accès avec échec au cache d'instructions, $h_{i_k} = 0$.

La solution trouvée ne doit biensûr pas dépasser la capacité maximale de la SPM. Autrement dit, elle doit satisfaire la contrainte exprimée par l'équation 1.23.

$$\sum_{i=1}^{N} s[i] \leq C \tag{1.23}$$

1.1.6 Programmes tests utilisés

Les expérimentations concernant l'optimisation de la consommation d'énergie en mémoire sont réalisées avec un ensemble de programmes tests sélectionnés dans le cadre du projet *MORE*. Ces programmes tests sont issus de six suites différentes : *MiBench* [Guthaus *et al.*, 2001], *SNU-RT*, *Mälardalen*, *Mediabenchs*, *Spec 2000* et *Wcet Benchs*. Le tableau 1.4 donne une description de ces programmes tests. Ils peuvent également être téléchargés à partir de [Benchmarks, 2010]. La majorité des programmes tests ne nécessite aucun argument à l'exception des programmes tests suivants dont les arguments utilisés sont :

- BitcountCE : bitcntsce 75000
- ShaCE : shace very_small_input1.asc
- DjpegCE : djpegce _-dct int _-ppm _-outfile very_small_albumart.jpeg.ppm very_small_albumart.jpeg

TABLE 1.4 – Liste des programmes tests utilisés.

Programmes tests	Suite	Description
ShaCE	MiBench	The secure hash algorithm that produces a 160-bit message digest for a given input.
BitcountCE	MiBench	Tests the bit manipulation abilities of a processor by counting the number of bits in an array of integers.
FirCE	SNU-RT	Finite impulse response filter (signal processing algorithms) over a 700 items long sample.
JfdctintCE	SNU-RT	Discrete-cosine transformation on 8x8 pixel block.
AdpcmCE	Mälardalen	Adaptive pulse code modulation algorithm.
CntCE	Mälardalen	Counts non-negative numbers in a matrix.
CompressCE	Mälardalen	Data compression using lzw.
DjpegCE	Mediabenchs	JPEG decoding.
GzipCE	Spec 2000	Compression.
NsichneuCE	Wcet Benchs	Simulate an extended Petri net. Automatically generated code with more than 250 if-statements.
StatemateCE	Wcet Benchs	Automatically generated code.

– GzipCE : gzipce crosstool_build.sh

1.2 Optimisation de fonctions tests mathématiques multimodales

1.2.1 Introduction

La qualité des méthodes d'optimisation est fréquemment évaluée en utilisant les fonctions tests standards de la littérature. Cette série de problèmes spécifiquement créés pour tester la performance des algorithmes d'optimisation, est regroupée en classes toutes continues :

– (a) : unimodale, convexe, de dimension élevée,
– (b) : multimodale, à deux dimensions avec un faible nombre d'extrema locaux,
– (c) : multimodale, à deux dimensions avec un nombre élevé d'extrema locaux,
– (d) : multimodale, de dimension élevée, avec un nombre élevé d'extrema locaux.

La classe (a) contient des fonctions très pratiques ainsi que des cas malveillants causant un ralentissement de la convergence vers l'extremum global unique. La classe (b) est médiane entre (a) et (c)-(d) et est utilisée pour tester la qualité des méthodes d'optimisation standards dans les environnements hostiles, spécifiquement ceux ayant peu d'extrema locaux avec un extremum global unique. Les classes (c)-(d) sont recommandées pour tester la qualité des méthodes d'optimisation intelligentes (résistantes), comme

par exemple les algorithmes métaheuristiques. Ces classes sont considérées comme étant des problèmes de test très difficiles. La classe (c) est "artificielle", dans un certain sens, puisque le comportement de la procédure d'optimisation est généralement justifié, expliqué et soutenu par les intuitions humaines sur une surface $2D$. En outre, des problèmes d'optimisation à deux dimensions apparaissent très rarement dans la pratique. Malheureusement, les problèmes d'optimisation discrets pratiques donnent des exemples avec un grand nombre de dimensions portant sur la classe (d). Par exemple, *ft10*, le plus petit programme test connu actuellement, appelé également *job shop scheduling problem* a une dimension de 90 alors que le plus grand connu a une dimension de 1980. Par conséquent, afin de tester la qualité réelle des algorithmes proposés, nous avons besoin d'examiner principalement des instances de la classe (d). En général, les approches basées sur le principe des algorithmes génétiques proposées pour l'optimisation continue ne dépassent pas la dimension 10. Par la suite, nous proposerons des solutions en dimensions 20 et 30.

Dans cette partie, nous présentons des fonctions tests classiques de la littérature. La plupart de ces fonctions sont décrites en détails dans [Suganthan *et al.*, 2005] et dans [Molga et Smutnicki, 2005]. Nous les avons réparties en deux séries de fonctions : les fonctions unimodales et les fonctions multimodales. Ces fonctions possèdent quelques propriétés similaires aux problèmes du monde réel et fournissent une bonne base pour tester la crédibilité d'un algorithme d'optimisation. Notons que, la polarisation [12] n'a aucune influence sur le résultat de la minimisation. Par conséquent, les définitions de fonctions trouvées dans la littérature peuvent différer de celles présentées dans la suite par une constante. Tous les tests sont formulés, ci-après, comme étant des problèmes de minimisation. Ils peuvent également être utilisés pour des problèmes de maximisation en inversant simplement le signe de la fonction.

1.2.2 Fonctions tests unimodales

1.2.2.1 Fonction Sphère

La fonction Sphère ou paraboloïde est un problème unimodal, convexe, continu et symétrique. Elle est définie par l'équation 1.24 :

$$F_2(\vec{x}) = \sum_{i=1}^{n} x_i^2 \tag{1.24}$$

où $\vec{x}$ est un vecteur de dimension n dont les composantes x_i appartiennent à l'intervalle $[-5.12; 5.12]$. Le minimum global se trouve à l'origine et la valeur de sa fonction est égale à zéro. La figure 1.3 montre une fonction Sphère en 3 dimensions.

1.2.2.2 Fonction bicarrée bruitée

La difficulté de cette fonction est liée au bruit stochastique ajouté. La fonction bicarrée bruitée ou *Noisy* est définie par l'équation 1.25 :

$$F_5(\vec{x}) = \left[\sum_{i=1}^{n}(i+1)x_i^4\right] + rand[0,1] \tag{1.25}$$

où $\vec{x}$ est un vecteur de dimension n dont les composantes x_i appartiennent à l'intervalle $[-1.28; 1.28]$. La valeur de la fonction du minimum global vaut zéro.

12. une constante ajoutée à la valeur d'une fonction.

FIGURE 1.3 – Fonction Sphère en 3 dimensions.

1.2.2.3 Fonction de Rosenbrock

La fonction de Rosenbrock est unimodale, mais le minimum se trouve dans une région très étroite, à l'intersection de vallées très peu pentues. Sa formule mathématique est définie par l'équation 1.26 :

$$F_4(\vec{x}) = \sum_{i=1}^{n-1} \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right] \tag{1.26}$$

où $\vec{x}$ est un vecteur de dimension n dont les composantes x_i appartiennent à l'intervalle $[-30; 30]$. Le minimum global est localisé en $(1, ..., 1)$ et la valeur de sa fonction vaut zéro. Cette fonction présente une vallée profonde de forme parabolique. Trouver la vallée est trivial, mais parvenir à une convergence vers l'optimum global est une tâche difficile. Dans la littérature, cette fonction est considérée comme étant un problème difficile en raison de l'interaction non linéaire entre les variables [Ortiz-Boyer *et al.*, 2005]. La figure 1.4 montre une fonction de Rosenbrock en 3 dimensions.

FIGURE 1.4 – Fonction de Rosenbrock en 3 dimensions.

1.2.3 Fonctions tests multimodales

1.2.3.1 Fonction de Rastrigin

La fonction de Rastrigin est fortement multimodale et de dimension élevée. Elle est définie mathématiquement par l'équation 1.27 :

$$F_1(\vec{x}) = \sum_{i=1}^{n} (x_i^2 - 10\cos(2\pi x_i) + 10) \tag{1.27}$$

où $\vec{x}$ est un vecteur de dimension n dont les composantes x_i appartiennent à l'intervalle $[-5.12; 5.12]$. L'emplacement des minima locaux est distribué régulièrement. Le minimum global se trouve à l'origine et la valeur de sa fonction est égale à zéro. La figure 1.5 montre une fonction de Rastrigin en 3 dimensions.

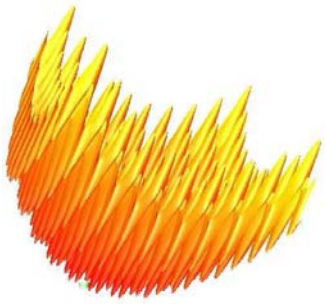

FIGURE 1.5 – Fonction de Rastrigin en 3 dimensions.

1.2.3.2 Fonction de Griewank

La fonction de Griewank présente plusieurs minima locaux étalés sur tout le domaine et sont uniformément distribués. Sa formule mathématique est définie par l'équation 1.28 :

$$F_3(\vec{x}) = \frac{1}{4000} \sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos(\frac{x_i}{\sqrt{i}}) + 1 \tag{1.28}$$

où $\vec{x}$ est un vecteur de dimension n dont les composantes x_i appartiennent à l'intervalle $[-600; 600]$. Le minimum global se trouve à l'origine et la valeur de sa fonction est égale à zéro. La figure 1.6 montre une fonction de Griewank en 3 dimensions.

FIGURE 1.6 – Fonction de Griewank en 3 dimensions.

1.2.3.3 Fonction de Schwefel

Cette fonction est également connue sous le nom de *Schwefel's sine root function*. Elle est définie par l'équation 1.29 :

$$F_6(\vec{x}) = \sum_{i=1}^{n} \left[x_i \sin(\sqrt{|x_i|}) \right] \tag{1.29}$$

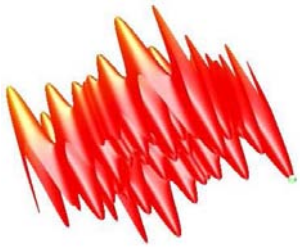

FIGURE 1.7 – Fonction de Schwefel en 3 dimensions.

où $\vec{x}$ est un vecteur de dimension n dont les composantes x_i appartiennent à l'intervalle $[-500; 500]$. La surface de la fonction de Schwefel est composée d'un grand nombre de pics et de vallées. La principale difficulté de cette fonction est que de nombreux algorithmes de recherche sont pris au piège du deuxième meilleur minimum qui est très loin de l'optimum global. En outre, le minimum global est proche des limites du domaine. Les algorithmes de recherche sont potentiellement sujets à converger vers la mauvaise direction dans l'optimisation de cette fonction. La valeur de la fonction du minimum global vaut zéro (pour F_6 le minimum global est $-420.9687 \times 20 = -8419.368$). La figure 1.7 montre une fonction de Schwefel en 3 dimensions.

1.2.3.4 Fonction d'Ackley

La fonction d'Ackley, généralisée à n dimensions par [Bäck, 1996], est définie par l'équation 1.30 :

$$F_7(\vec{x}) = 20 + e - 20\, e^{-0.2\ (\frac{1}{n}\sum_{i=1}^{n} x_i^2)^{\frac{1}{2}}} - e^{\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)} \tag{1.30}$$

où $\vec{x}$ est un vecteur de dimension n dont les composantes x_i appartiennent à l'intervalle $[-32; 32]$. La fonction d'Ackley est hautement multimodale avec des optima locaux uniformément distribués. Le minimum global se trouve à l'origine et la valeur de sa fonction est égale à zéro. La figure 1.8 montre une fonction d'Ackley en 3 dimensions.

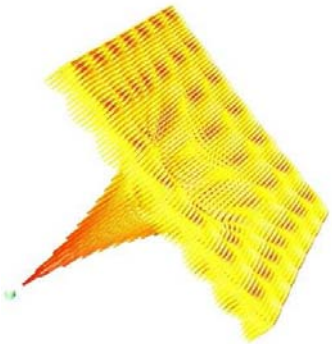

FIGURE 1.8 – Fonction d'Ackley en 3 dimensions.

1.2.3.5 Fonction de Michalewicz

La fonction de Michalewicz est paramétrée et multimodale avec plusieurs optima locaux ($n!$) situés entre les plateaux. Le paramètre m définit la "raideur" des vallées ou des bords. Plus le paramètre m est élevé, plus l'optimum global devient difficile à trouver. Dans nos expérimentations, nous prenons $m = 10$. La formule mathématique de la fonction de Michalewicz est donnée par l'équation 1.31 :

FIGURE 1.9 – Fonction de Michalewicz en 3 dimensions.

$$F_8(\vec{x}) = -\sum_{i=1}^{n} \sin(x_i) \sin^{2m}(\frac{i - x_i^2}{\pi}) \tag{1.31}$$

où $\vec{x}$ est un vecteur de dimension n dont les composantes x_i appartiennent à l'intervalle $[-\pi; \pi]$. La figure 1.9 montre une fonction de Michalewicz en 3 dimensions.

1.2.3.6 Fonction d'Himmelblau

Dans l'optimisation mathématique, la fonction d'Himmelblau est une fonction multimodale utilisée pour tester les performances des algorithmes d'optimisation. Elle est définie par l'équation 1.32 :

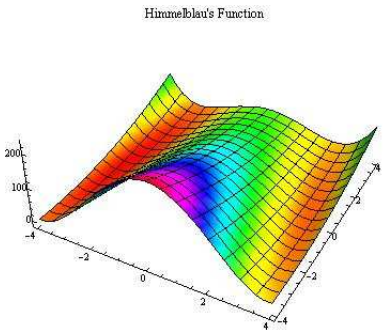

FIGURE 1.10 – Fonction d'Himmelblau en 2 dimensions.

$$F_9(\vec{x}) = (x_2 + x_1^2 - 11)^2 + (x_1 + x_2^2 - 7)^2 + x_1 \tag{1.32}$$

où $-5 \leq x_i \leq 5$ pour $i = 1, 2$. La valeur de la fonction du minimum global vaut -3.78396. La figure 1.10 montre une fonction d'Himmelblau en 2 dimensions.

1.2.3.7 Fonction de Shubert

C'est une fonction multimodale définie par l'équation 1.33 :

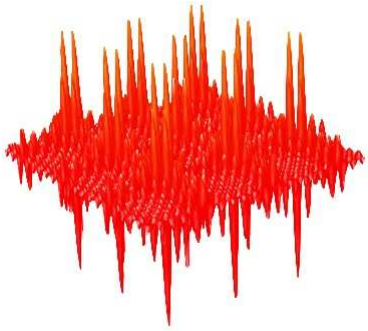

FIGURE 1.11 – Fonction de Shubert en 3 dimensions.

$$F_{10}(\vec{x}) = \sum_{i=1}^{5} i\cos((i+1)x_1 + i) \sum_{i=1}^{5} i\cos((i+1)x_2 + i) \tag{1.33}$$

où $-10 \leq x_i \leq 10$ pour $i = 1, 2$. Elle possède 760 minima locaux, dont 18 sont des minima globaux avec une valeur de fonction de -186.7309. En outre, les optima globaux sont inégalement répartis. La figure 1.11 montre une fonction de Shubert en 3 dimensions.

1.3 Conclusion

Ce premier chapitre présente les deux problématiques traitées durant mon doctorat, à savoir : la réduction de la consommation d'énergie en mémoire dans les systèmes embarqués et l'optimisation de fonctions tests mathématiques multimodales. Comme cela est expliqué en première section 1.1, l'omniprésence des systèmes embarqués est un fait. Leur très forte dépendance en énergie devient un problème récurrent et primordial. L'identification de la mémoire comme étant un principal consommateur d'énergie dans ce type de systèmes incite à se focaliser dessus. Pour ce faire, le besoin s'est fait ressentir de mesurer l'impact des solutions proposées sur l'énergie consommée. C'est pourquoi, la sous-section 1.1.4 présente un modèle d'estimation de la consommation d'énergie dans une architecture mémoire constituée des principaux composants décrits en sous-section 1.1.3. Ce modèle a été proposé durant cette thèse. Quant à la sous-section 1.1.5, elle montre que ce problème est combinatoire de type sac à dos et explique ses contraintes.

La deuxième section 1.2 présente le problème de l'optimisation globale de fonctions tests mathématiques qui sont réparties en deux séries : des fonctions unimodales et des fonctions multimodales. Ces fonctions possèdent des propriétés similaires aux problèmes du monde réel et fournissent une bonne base pour tester la crédibilité d'un algorithme d'optimisation, notamment du fait de leur grand nombre d'optima locaux.

Dans le chapitre suivant, nous allons présenter les différentes méthodes et techniques existantes afin de résoudre ces deux problèmes d'optimisation. L'accent est mis sur le problème de la réduction de la

consommation d'énergie en mémoire car c'est un domaine pour lequel il n'existe pas, à notre connaissance, un état de l'art assez large et général traitant de différents domaines et s'appliquant à différents types de mémoire.

Chapitre 2

Optimisation de la consommation d'énergie et de fonctions multimodales

Sommaire

2.1 Réduction de la consommation d'énergie en mémoire

2.1.1 Introduction

Il existe diverses options pour économiser l'énergie et pour augmenter l'autonomie des batteries. Ces différentes approches peuvent être divisées en deux catégories : optimisations matérielles et optimisations logicielles. Les techniques d'optimisations matérielles, hors du contexte de mon doctorat, ne seront pas traitées dans ce mémoire, mais une quantité importante de littérature les concernant est disponible [Egger *et al.*, 2006; Hallnor et Reinhardt, 2000] et voir les premières parties de [Graybill et Melhem, 2002]. Notons que quelques travaux combinent de façon intéressante les techniques d'optimisation matérielles et logicielles, comme [Poletti *et al.*, 2004] (qui s'appuie sur un DMA [13] pour réduire l'énergie

13. DMA : *Direct Memory Access.*

et le coût de copie entre les mémoires SPM et DRAM), ou encore [Benini *et al.*, 2000] (qui utilise une petite mémoire ASM [14] placée à côté du processeur à la place de la traditionnelle mémoire cache afin de réduire la consommation énergétique des accès mémoire). Dans cette thèse, nous nous focaliserons sur les techniques logicielles assistées par le compilateur.

Quelques techniques et algorithmes, synthétisés dans [Benini et Micheli, 1999], essayent d'allouer, de façon optimale, le code et/ou les données de l'application en SPM afin de réduire la consommation d'énergie dans les systèmes embarqués. Le lecteur intéressé peut voir [Benini et Micheli, 2000] pour une liste complète de références. Toutefois, ces deux références datent d'une décennie et considèrent tous les constituants des systèmes : processeurs, mémoires et ressources de communication et ce pour les optimisations matérielles et logicielles. En revanche, dans ce chapitre, nous nous consacrerons uniquement aux techniques logicielles pour optimiser la consommation d'énergie dans tous les types de mémoires (caches, SPMs, DRAMs, *etc*.) et nous donnerons, dans les sections suivantes, des références plus récentes.

2.1.2 Allocation des données en mémoire

Les approches présentées, dans cette section, tentent d'aider à la détermination de l'allocation mémoire optimisée permettant de réduire la consommation énergétique en fonction du type de mémoire et du comportement de l'application. Pour ce faire, ces méthodes utilisent les données des profils d'exécution pour collecter des informations sur les fréquences d'accès (nombre de fois qu'une donnée est accédée). Une approche consiste à placer les données fréquemment utilisées dans une mémoire à faible coût d'accès alors que les autres données seront placées dans une mémoire à faible coût de stockage [Avissar *et al.*, 2001]. La figure 2.1 illustre ce principe avec l'exemple de la mémoire RAM [15], caractérisée par une consommation en énergie permanente mais par de faibles coûts d'accès et du disque dur HDD [16], caractérisé par ses coûts d'accès élevés mais par un coût énergétique de stockage très faible.

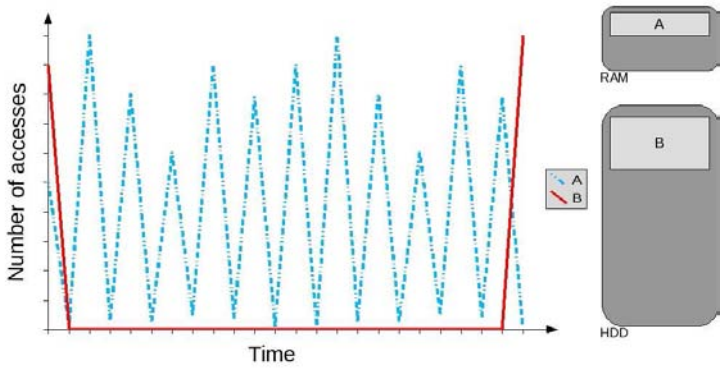

FIGURE 2.1 – Allocation des données en mémoire.

Dans ces techniques, la plupart des auteurs modélisent le problème comme étant une programmation linéaire d'entier 0/1 (ILP [17]) et utilisent par la suite un *IP solver* disponible pour le résoudre. Les

14. ASM : *Application-Specific Memory.*
15. RAM : *Random Access Memory.*
16. HDD : *Hard Disk Drive.*
17. ILP : *Integer Linear Programming.*

principales différences entre ces approches concernent les objets considérés (tableaux, boucles, variables globales, de tas ou de pile, *etc*.). Dans [Avissar *et al.*, 2002], à cause de la taille réduite de la mémoire SRAM, les données les moins utilisées sont d'abord allouées aux bancs mémoire lents (*DRAM*), alors que les données les plus fréquemment utilisées sont gardées en mémoire rapide (*SRAM*) le plus longtemps possible. Ils considèrent les variables globales et de pile et choisissent entre SPM et cache alors que [Steinke *et al.*, 2002; Wehmeyer *et al.*, 2004] considèrent les variables globales, les fonctions et les blocs de base et choisissent entre les différents bancs SPM seulement. Quant à [Udayakumaran *et al.*, 2002], ils considèrent les variables globales et statiques et choisissent également entre les bancs de la SPM.

Ces techniques sont toutes fondées sur la fréquence d'accès aux données. En revanche, une autre approche consiste à se focaliser sur les données les plus conflictuelles dans la mémoire cache. [Panda *et al.*, 1997] considèrent les tableaux et les variables scalaires et choisissent entre DRAM et SPM. Ils assignent d'abord les constantes et les variables scalaires à la SPM. Les tableaux dont la taille est supérieure à la l'espace libre en SPM sont alloués en DRAM. [Truong *et al.*, 1998] quant à eux, considèrent les applications utilisant des structures de données hétérogènes allouées dynamiquement (listes, B-arbres, *quad* arbres, *etc*.). Les auteurs se basent sur les données des profils d'exécution pour placer les données les plus conflictuelles en SRAM.

Toutes ces approches nécessitent la connaissance de la taille de la SPM à la compilation, mais [Nguyen *et al.*, 2005] présentent une méthode dont l'exécutable produit est portable à travers les SPMs de n'importe quelle taille. La méthode consiste à découvrir d'abord la taille de la SPM en faisant un appel système soit bas niveau, soit au niveau de l'OS s'il est disponible. Le sondage des adresses en mémoire en utilisant un modèle de recherche binaire et l'observation de la latence permet également de trouver la rangée d'adresses appartenant à la SPM. Puis, l'algorithme d'allocation mémoire est le même que celui dans [Avissar *et al.*, 2002]. Leurs résultats indiquent qu'en moyenne, une accélération de 36% est atteinte comparé à toutes les allocations en DRAM, alors que [Avissar *et al.*, 2002] atteignent une accélération de 41%. Les résultats montrent également que le surcoût avoisine 1% à la fois en ce qui concerne la taille du code et le temps d'exécution. Toutefois, aucune mesure énergétique n'est réalisée.

Le choix correct d'une allocation mémoire dynamique est d'une grande importance. Dans ce contexte, certains auteurs commencent à rendre disponibles les nouvelles méthodologies de conception et des outils aux concepteurs afin de les aider à explorer les différences entre les différentes configurations d'allocation de mémoire dynamiques, ce qui permet d'utiliser convenablement les ressources des dispositifs embarqués. [Mamagkakis *et al.*, 2006] proposent un script et un support d'automatisation complet (avec une interface utilisateur graphique) qui personnalise l'allocateur de mémoire dynamique en fonction du domaine de l'application cible et de la hiérarchie mémoire sous-jacente du système embarqué. L'allocateur mémoire dynamique réside au niveau intermédiaire ou au niveau du système d'exploitation (si jamais il est disponible). Pour ce faire, les auteurs suivent quatre étapes. D'abord, ils déterminent le profil dynamique de l'application sous étude. Ensuite, en fonction de ces données de profil, ils évaluent les décisions de conception faites par l'allocateur de mémoire dynamique (*DM*[18]) (une explication complète du fonctionnement de l'allocateur DM est donnée dans [Atienza *et al.*, 2004]). Après cela, ils explorent les différentes configurations générées par l'étape précédente. Finalement, le concepteur peut sélectionner entre un large choix de configurations Pareto-optimales d'allocations dynamiques en mémoire. Le résultat de leur script et outils automatisés est la réduction de la consommation de l'énergie de 72% en moyenne et la réduction du temps d'exécution de 40% en moyenne, chose qui est démontrée par l'utili-

18. DM : *Dynamic Memory.*

sation d'une application réseau sans fil de la vie réelle et par une application multimédia.

Les travaux de [Udayakumaran et Barua, 2003] sont très intéressants. Ils se basent sur ceux de [Kandemir *et al.*, 2001] et essaient de les étoffer. En effet, [Udayakumaran et Barua, 2003] ont amélioré la généralité de la méthode proposée dans [Kandemir *et al.*, 2001] en l'appliquant à toutes les variables globales et de pile ainsi qu'à tous les enchaînements (*access pattern*) des accès à ces variables y compris pour des codes contenant des enchaînements d'accès irréguliers au lieu de considérer seulement les tableaux de codes scientifiques et multimédia bien structurés. De plus, la méthode proposée dans [Udayakumaran et Barua, 2003] est optimisée globalement pour le programme en entier, alors qu'elle l'est localement pour chaque boucle dans [Kandemir *et al.*, 2001]. Notons également que les deux méthodes font aller et revenir les données entre DRAM et SPM sous contrôle du compilateur, mais [Kandemir *et al.*, 2001] rendent la SPM disponible en entier pour chaque boucle imbriquée. En revanche, [Udayakumaran et Barua, 2003] peuvent choisir de n'utiliser qu'une partie de la SPM pour les données qui sont partagées entre des instructions de contrôle successives économisant ainsi le temps et l'énergie de transfert à la DRAM. Enfin, [Udayakumaran *et al.*, 2006; Udayakumaran et Barua, 2006] étendent le travail effectué en manipulant aussi les objets du code et les tableaux. Les résultats obtenus par simulation montrent une réduction en moyenne du temps d'exécution de plus de 39.8% et de la consommation d'énergie de plus de 31.3% pour leurs programmes tests (en fonction de la taille SRAM utilisée) par rapport à [Avissar *et al.*, 2002].

2.1.3 Localité des accès mémoire

Le principe de la localité consiste à diviser la SPM en bancs, à y disposer les données et à les allouer au même banc autant que possible, tout en mettant les autres bancs en mode de puissance bas. Il existe deux types de localités : la localité spatiale et la localité temporelle. Certaines méthodes sont basées sur l'optimisation de la localité spatiale ; des accès SPM successifs utilisent le même banc SPM le plus longtemps possible. D'autres méthodes se basent sur la localité temporelle ; les bancs SPM accédés récemment sont susceptibles de l'être encore dans un futur proche [Tanenbaum, 2005].

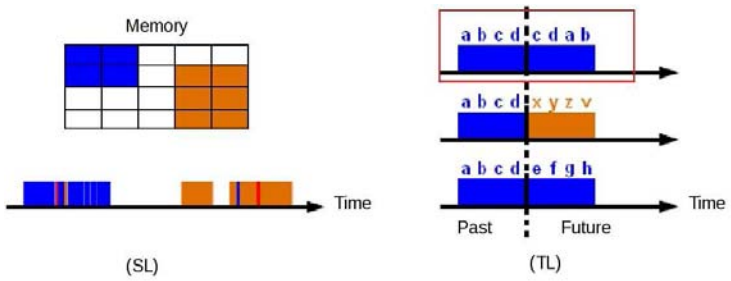

FIGURE 2.2 – Localité spatiale (SL) et localité temporelle (TL).

En effet, augmenter la localité d'un ensemble de bancs mémoire fait croître clairement la durée pendant laquelle les autres bancs mémoire ne sont pas accédés, ce qui aide à amortir le coût de transition d'un banc en mode basse consommation et son retour en mode de consommation normal.

2.1.3.1 Localité spatiale

La méthode présentée dans [Athavale *et al.*, 2001] explore la consommation d'énergie des mécanismes d'allocations des tableaux en Java. En utilisant un ensemble de programmes tests (*benchmarks*)

dominés par des tableaux et une architecture mémoire partitionnée supportant des modes basse puissance, les auteurs étudient deux techniques d'optimisation des données : la modification de la disposition mémoire et l'entrelacement de tableaux. La modification de la disposition mémoire consiste à changer l'ordre de stockage des données à l'intérieur d'un tableau afin d'améliorer sa localité spatiale. L'entrelacement de tableaux regroupe ensemble, dans le même module mémoire, les éléments appartenant à différents tableaux multidimensionnels augmentant ainsi l'intervalle d'inter-accès (temps entre deux références du même module) des modules non utilisés. Cela fournit une opportunité pour mettre les modules mémoire en mode basse puissance durant une période plus longue. Les résultats expérimentaux obtenus par les auteurs montrent que la modification de la disposition mémoire et l'entrelacement de tableaux fournissent respectivement 9,68% et 14,96% en moyenne de gains en énergie. En revanche [De La Luz *et al.*, 2002] présentent un mécanisme automatique d'entrelacement de tableaux valable pour n'importe quel langage et n'importe quelle mémoire. Dans cet article, l'entrelacement de tableaux est une technique de transformation de l'espace des données (disposition de tableau) qui prend plusieurs tableaux et les regroupe en un seul. Les auteurs réalisent une économie de la consommation d'énergie de 54.2% pour différentes configurations mémoire.

La stratégie de compilation proposée dans [Kandemir *et al.*, 2004; Kandemir *et al.*, 2005] est également efficace dans la réduction du courant de fuite (*leakage energy*) des SPMs intégrées (*on-chip*). Cette méthode a l'avantage de considérer les tableaux et les boucles en général sans restrictions à un langage particulier. L'idée dans [Kandemir *et al.*, 2005] est de diviser la SPM en bancs tout en optimisant la disposition des données (guidé par le compilateur) et en faisant migrer des données afin de maximiser le ralentissement des bancs SPM augmentant ainsi les chances de placer les bancs mémoires en mode basse puissance. Ce travail se focalise sur la réduction de la consommation d'énergie des SPMs *on-chip* sans heurter la performance globale du système. Le pourcentage d'économie des courants de fuite réalisé par leur approche est de 41.3% en moyenne sur tous leurs programmes tests.

L'article [Zhang *et al.*, 2003] présente des techniques de restructuration de code pour les applications contenant principalement des tableaux et des pointeurs afin de réduire la consommation énergétique des mémoires cache. L'idée est de permettre au compilateur d'analyser le code et d'y insérer des instructions qui éteignent les lignes de la mémoire cache contenant les variables qui ne sont pas utilisées dans le calcul en cours. Cette extinction ne détruit pas le contenu d'une ligne de cache et l'allumage de la ligne en question induit un très faible surcoût. Leurs résultats indiquent que leur approche réduit la consommation énergétique de la mémoire cache de manière significative : des économies de 47.1% à 62.4% sont atteintes par rapport à une approche matérielle.

2.1.3.2 Localité temporelle

D'autres méthodes se basent sur la localité temporelle ; les bancs SPM accédés récemment sont susceptibles de l'être encore dans un futur proche. [Verma *et al.*, 2004] présentent une approche fondée sur les profils pour remplir le contenu de la SPM. Ces profils sont déterminés sur la base des cycles de vie des variables et des segments de code. Ces variables et ces segments de code sont choisis de façon à minimiser le surcoût énergétique causé par les allers-retours des objets mémoire depuis la SPM vers la DRAM. Cette technique calcule aussi les adresses mémoire appartenant à la SPM où les variables et les segments de code doivent être copiés. Ces adresses sont calculées de façon telle qu'un grand nombre de variables et de segments de code partagent le même espace SPM. Leurs expérimentations rapportent respectivement une réduction moyenne de 34% et 18% de la consommation d'énergie et du temps d'exécution. Une faible augmentation de la taille du code (moins de 1%) est également notée. Dans [Issenin *et al.*, 2004], les auteurs présentent une approche automatisée pour exploiter les opportunités de

réutilisation des données dans un programme. Cette méthode crée une SPM personnalisée employant une organisation hiérarchique des tampons et insère le code approprié dans le code source afin de réaliser les transferts nécessaires vers/depuis cette SPM personnalisée. En utilisant cette approche, les auteurs arrivent à réduire la consommation énergétique du sous-système mémoire contenant une SPM par un facteur de deux en moyenne en comparaison avec un système contenant une mémoire cache de même taille.

2.1.4 Partitionnement de la mémoire

Pour n'importe quelle technologie donnée, les temps d'accès et l'énergie nécessaire à un accès mémoire sont fonction de la taille mémoire : plus large sera la mémoire, plus grands seront les temps d'accès et l'énergie consommée par accès. L'impact de la taille mémoire sur l'énergie et sur les performances est décrit en figure 2.3, où l'on peut remarquer que plus la taille de la mémoire grandit, plus l'énergie consommée et les temps de cycles augmentent [Wehmeyer *et al.*, 2004]. Les valeurs données dans cette figure sont obtenues en utilisant le modèle de l'outil CACTI [Wilton et Jouppi, 1996; Mamidipaka et Dutt, 2004] et en supposant une taille de 0.5μm.

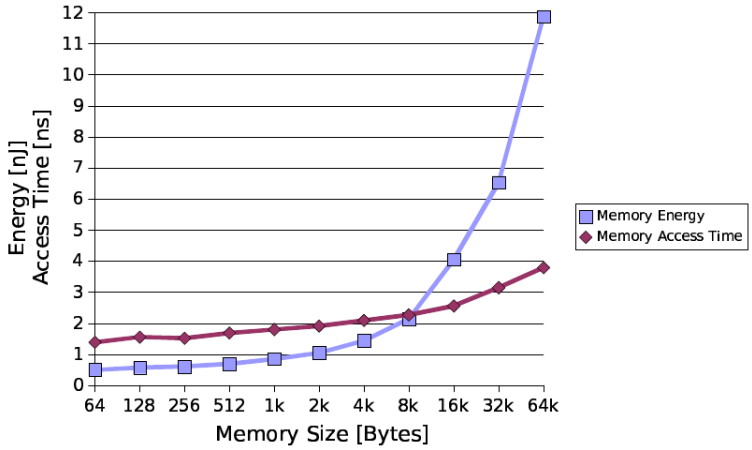

FIGURE 2.3 – Relation entre énergie et taille mémoire [Wehmeyer *et al.*, 2004].

En général, à cause de l'augmentation des tailles des applications et des tailles des mémoires, les temps d'accès ainsi que la consommation énergétique croissent de plus en plus avec le temps. De plus, la tendance en industrie est d'aller vers de petites mémoires partitionnées comme, par exemple, le dernier processeur ARM. C'est pourquoi, le fait d'allouer les points chauds des applications à de petites mémoires au lieu d'utiliser une seule et grande mémoire hétérogène, prend tout son sens. D'ailleurs, plusieurs auteurs ont exploré la possibilité d'utiliser des mémoires partitionnées de façon optimale.

Effectivement, plusieurs travaux existent. [Angiolini *et al.*, 2005] présentent un algorithme pour résoudre le problème du partitionnement optimal de la SPM. Le principe de leur algorithme est de relier les segments de la mémoire externe aux bancs physiquement partitionnés d'une SPM *on-chip* ; cette architecture fournit de significatifs gains en énergie. L'algorithme peut manipuler un nombre arbitraire de partitions et prend en compte le surcoût induit par le partitionnement. Dans [Wehmeyer *et al.*, 2004], une SPM est partitionnée en plusieurs petites régions contigües où chacune est destinée à recevoir aussi bien des données que des instructions. Leurs résultats simulés montrent, qu'en utilisant une SPM partitionnée, des économies allant jusqu'à 22% en termes de consommation d'énergie mémoire peuvent être obtenues.

En ce qui concerne les mémoires DRAMs, [Ozturk et Kandemir, 2005a] abordent le problème de l'énergie mémoire en utilisant deux techniques complémentaires : la migration des données et la compression des données. La migration des données fait bouger les données d'un banc mémoire à un autre banc à l'exécution et puis, la compression de données serre les blocs de données en mémoire, ce qui permet une meilleure utilisation de l'espace disponible en DRAM. Ensuite, les auteurs essaient d'exploiter au mieux les modes basse puissance existants. Ils se focalisent sur les applications embarquées dominées par les tableaux avec des fonctions et des boucles. Toutefois, le problème est formulé à l'aide d'un ILP [19] pour les bancs mémoire de tailles uniformes mais aucun résultat expérimental n'est donné. En revanche, l'idée d'utiliser des bancs mémoire de tailles non-uniformes afin de réduire la consommation d'énergie proposée dans [Ozturk et Kandemir, 2005b] est nouvelle et intéressante. Là, les auteurs proposent une approche basée sur l'ILP qui retourne les tailles optimales des bancs mémoire non-uniformes et le mapping donnée/banc mémoire correspondant. Ils étudient également comment la migration de données peut encore faire mieux par rapport à l'allocation aux bancs mémoire de tailles non-uniformes. Les résultats obtenus, dans cet article, montrent que la stratégie proposée apporte d'importants gains par rapport à l'approche basée sur des bancs mémoire de tailles uniformes et puis, le fait d'utiliser en plus la migration de données entre les bancs tend à augmenter ces gains.

Pour les mémoires cache de données et d'instructions, [Kim *et al.*, 2001] se concentrent sur le partitionnement architectural des ressources de la mémoire cache afin d'optimiser l'énergie et le retard induit. Ils s'intéressent spécifiquement à la façon de partager la mémoire cache en plusieurs petites unités où chacune d'elles devient un cache. Cette petite unité est appelée sous-cache. Ensuite, ils activent sélectivement celle contenant la donnée sous considération. Non seulement les architectures sous-cache réduisent les coûts énergétiques par accès, mais elles peuvent améliorer, potentiellement, le comportement de la localité. Les auteurs présentent une plate-forme unifiée pour la conception, l'implémentation et l'évaluation des différentes architectures sous-cache. Différentes techniques de placement de données, de prédiction de sous-cache et de sondage sélectif sont proposées et évaluées en utilisant un ensemble d'applications diversifié. Les résultats montrent qu'avec un mécanisme de sous-cache intelligent, une amélioration énergétique de 42% dans le système mémoire peut être atteinte et cela en comparaison avec une architecture contenant une mémoire cache directe de $32Ko$ pour les applications MediaBench et SpecJVM98. Nous pouvons également citer [Shrivastava *et al.*, 2005] dans lequel les auteurs proposent et explorent des algorithmes visant à réduire la consommation d'énergie dans les architectures horizontales de caches de données partitionnés.

Dans tous les articles présentés précédemment, l'algorithme de partitionnement a été conçu pour des hiérarchies mémoire spécifiques. En revanche, l'algorithme EMBARC décrit dans [Hiser et Davidson, 2004] essaie de réaliser, à la compilation, un algorithme complet et adaptatif pour assigner efficacement les variables aux partitions mémoire dans une hiérarchie mémoire arbitraire afin d'améliorer l'énergie et les performances du programme. Pour ce faire, EMBARC fait des suppositions, à priori, sur la hiérarchie mémoire de la machine au lieu de lire le fichier de description de la hiérarchie mémoire (fournit par le compilateur, le concepteur ou l'utilisateur final). En se basant sur cette information et sur les données de profil, l'algorithme EMBARC crée un schéma d'allocation aux partitions pour chaque variable dans le programme. Il supporte une large variété de modèles mémoire incluant les mémoires SRAMs intégrées, plusieurs couches de mémoires cache et même des partitions de DRAM. Bien qu'il soit conçu pour manipuler des hiérarchies mémoire variées, EMBARC est capable de générer des schémas d'allocation aux partitions de qualité similaire à celle des algorithmes conçus pour des architectures mémoire spécifiques. Cependant, ces allocations ne sont que statiques ; elles ne changent pas au fur et à mesure de l'exécution

19. ILP : *Integer Linear Programming.*

du programme. EMBARC ne peut pas, non plus, prendre des décisions de mouvement de données dynamiques ; un schéma d'allocation des variables aux partitions ne peut pas changer au fur et à mesure que le programme entre dans différentes phases.

2.1.5 Les bancs mémoire et les modes de puissance

Récemment, l'utilisation des bancs mémoire s'est répandue pour la réduction de la consommation de l'énergie mémoire. Là, l'espace mémoire est divisé en de multiples bancs mémoire et chacun peut être contrôlé indépendamment des autres. L'autre avantage d'une architecture mémoire organisée en bancs est la possibilité d'utiliser les modes de faible puissance en plaçant un banc mémoire qui n'est pas activement utilisé en mode de faible consommation énergétique. Des efforts ont déjà été entrepris d'une part pour déterminer quand et comment les bancs mémoire devraient être éteints, d'autre part pour sélectionner le mode de faible puissance à utiliser quand plusieurs modes sont disponibles. Ainsi, les optimisations présentées dans cette section tirent avantage des modes de consommation de faible puissance et des structures mémoire organisées en bancs.

Dans [De La Luz *et al.*, 2006], les auteurs utilisent pour chaque banc mémoire l'un des trois modes suivants : lecture/écriture (*R/W*), actif et inactif. Dans l'état R/W, le banc mémoire est accédé, soit en lecture, soit en écriture. Ce mode consomme aussi bien l'énergie dynamique que l'énergie des courants de fuite. Dans l'état actif, le banc mémoire est actif (allumé), mais pas accédé en lecture ou en écriture. Dans cet état, le banc mémoire consomme l'énergie des courants de fuite. Finalement, dans l'état inactif, le banc mémoire ne contient aucune donnée intéressante et son approvisionnement est fermé. [De La Luz *et al.*, 2006] s'intéressent exclusivement aux applications qui font des allocations mémoire dynamiques (*malloc(), free(), etc*.) et regroupent les données allouées dynamiquement ayant une affinité temporelle dans l'espace d'adressage physique de telle sorte que ces données occupent un petit nombre de bancs mémoire. Les bancs mémoire restants peuvent être éteints pour économiser l'énergie. Les auteurs emploient également la migration dynamique des données pour augmenter les gains en énergie. Ils utilisent pour cela un seuil de migration (*MT*). Si la taille totale des données résidant dans un banc mémoire donné descend en dessous du seuil MT, toutes les données de ce banc migrent vers d'autres bancs mémoire actifs (s'il y en a) en utilisant l'opération de copie mémoire à mémoire. Les résultats expérimentaux montrent que la stratégie des auteurs économise significativement l'énergie. On constate que lorsque la migration est utilisée toute seule, les gains moyens en termes de courants de fuite sont de 28.8%. Alors que lorsqu'elle est utilisée en conjonction avec le regroupement, les gains s'élèvent en moyenne à 48.7%.

D'autres auteurs considèrent, en revanche, cinq modes de puissance : un mode actif (le seul mode durant lequel une opération de lecture ou d'écriture peut avoir lieu) et quatre modes de basse puissance : veille (*standby*), sieste (*napping*), hors tension (*power-down*) et hors service (*disabled*). Chacun de ces modes a sa propre consommation d'énergie et son propre temps de re-synchronisation. [De La Luz *et al.*, 2000] abordent le problème de la réduction automatique de la consommation d'énergie d'un système mémoire *off-chip* partitionné (DRAM) ayant des bancs multiples en organisant l'ordre des calculs et la disposition des données. Leur approche, basée sur le compilateur, consiste en un algorithme d'allocation de tableaux (placement de données) et en un module d'optimisation qui utilise à la fois deux techniques d'optimisation de boucles (*loop fission* et *loop splitting*) et une approche d'optimisation des données (renommage de tableaux) afin d'augmenter l'efficacité de l'algorithme d'allocation de tableaux. L'idée de cet algorithme d'allocation est de regrouper les tableaux ayant des modèles de temps de vie similaires dans le(s) même(s) banc(s) mémoire afin d'éteindre le banc en question une fois que le temps de vie du tableau est terminé. Leurs expérimentations montrent que la plate-forme proposée améliore l'énergie mémoire de 86% par rapport à un schéma qui garde tous les bancs mémoire dans le mode actif

(complètement opérationnel) tout le temps et de 70% par rapport à un schéma qui utilise les modes de faible puissance sans faire aucune optimisation de boucle ou de données. Alors que [De La Luz *et al.*, 2002] présentent un mécanisme automatique d'entrelacement de tableaux valable pour n'importe quelle mémoire. Ce mécanisme automatique est développé sur une plate-forme mathématique dans laquelle les modèles d'accès aux tableaux d'une application sont capturés en utilisant des représentations basées sur les graphes (*ATG*) et transformés en utilisant des matrices de transformation de données linéaires. Ils observent des gains en énergie à travers les différentes configurations mémoire en utilisant un ensemble d'applications basées sur les tableaux : en utilisant le contrôle des modes de puissance, ils atteignent des économies en énergie de 24.6% en moyenne comparé à une version sans les modes de puissance (tous les modules mémoire sont actifs durant toute la durée d'exécution du programme).

D'autres auteurs considèrent quatre modes de puissance : un mode actif et trois modes de basse puissance ; veille (*standby*), sieste (*napping*) et hors tension (*power-down*). Les modes faible puissance peuvent être utilisés seulement si le banc est inactif (*idle*). Chaque mode de faible puissance possède ses propres coûts de consommation d'énergie et de re-synchronisation à cause de la mise hors service de certaines parties de la DRAM. L'article [Ozturk et Kandemir, 2005a], déjà présenté en section 2.1.4, adresse le problème de l'optimisation de l'énergie consommée en mémoire en utilisant deux techniques complémentaires : la migration des données et leur compression. Ainsi, les auteurs essaient d'exploiter au mieux les modes de faible puissance disponibles. Cela est possible du moment que la migration des données est capable de placer les blocs des données ayant des modèles d'accès et/ou des temps de vie similaires dans le même ensemble de bancs augmentant la chance de mieux utiliser les modes de faible puissance. De la même façon, l'augmentation du nombre de bancs mémoire inactifs (*idle*) qui sont candidats pour être mis en modes de faible puissance est possible. Les auteurs s'intéressent aux applications embraquées basées sur les tableaux, les fonctions et les boucles. Le problème est formulé en utilisant un ILP. En revanche, [Ozturk et Kandemir, 2006] proposent et évaluent un nouveau schéma de gestion de l'énergie lequel est basé sur la réplication des données. L'idée derrière leur approche est d'utiliser la réplication afin de prévenir la ré-activation d'un banc mémoire hors tension. Pour ce faire, ils ont implémenté à la fois une heuristique et une solution basée sur l'ILP pour le placement des données et le problème de la réplication dans une architecture mémoire organisée en bancs. Pour les deux solutions, au lieu de déterminer les blocs de données à répliquer, ils doivent détecter quels bancs mémoire sont accédés ensemble. De cette façon, ils peuvent placer les blocs de données qui sont accédés de manière simultanée dans le même banc mémoire. Quant à la méthode heuristique, ils ont construit une structure de données appelée table d'affinités (*AT*) qui garde trace des blocs mémoire accédés, les uns après les autres, dans une certaine période de temps définie par l'utilisateur. Concernant la solution basée sur l'ILP, ils déterminent les blocs de données à répliquer et le mapping donnée/banc en utilisant des variables $0-1$. Pour chaque emplacement possible du bloc de données, ils définissent des variables $0-1$. En utilisant ces variables $0-1$, ils déterminent l'emplacement de chaque bloc de données. Ils se focalisent sur les programmes embarqués contenant beaucoup de données, construits en utilisant des boucles imbriquées (avec un temps de compilation de la boucle connu) et des tableaux (avec des expressions affines). Cependant, aucun résultat expérimental n'est donné dans les deux articles.

Plusieurs autres techniques essaient de tirer profit des modes de faible puissance en vue de réduire la consommation d'énergie puis les appliquent à tous les types de mémoire. Pour la mémoire cache de données [Zhang *et al.*, 2003], pour la SPM [Kandemir *et al.*, 2004; Kandemir *et al.*, 2005] et pour la DRAM [Athavale *et al.*, 2001]. Comme ces approches sont déjà décrites dans les sections précédentes, ici elles sont simplement listées afin d'éviter la redondance. L'objectif étant de montrer que l'organisation de la mémoire en bancs et les modes de faible puissance peuvent être utilisés pour différents types de mémoire.

2.1.6 Synthèse des heuristiques classiques utilisées

Dans tous les travaux présentés dans les sections précédentes, le placement des données est, comme nous l'avons vu, guidé par les caractéristiques de la mémoire considérée (rapidité d'accès, coût énergétique, grand nombre de cas des accès avec échec, *etc*.). À cause de la taille réduite de la mémoire SRAM, il s'agit d'y allouer les données de façon optimale afin de réaliser des gains en énergie plus importants. Une approche est de placer les données intéressantes dans une mémoire à faible coût d'accès (*SPM*) alors que les autres données sont placées dans une mémoire à faible coût de stockage (*DRAM*). Ces données peuvent être les données les plus fréquemment accédées/utilisées [Avissar *et al.*, 2002; Dominguez *et al.*, 2005; Udayakumaran *et al.*, 2002; Wehmeyer *et al.*, 2004] ou les données les plus conflictuelles dans le cache [Panda *et al.*, 1997; Truong *et al.*, 1998].

En vue de déterminer ces données intéressantes, ces méthodes utilisent des données de profils (*data profiling*) dans le but d'obtenir des informations sur la fréquence des accès mémoire. Ces informations peuvent être collectées soit statiquement en analysant le code source des programmes tests, soit dynamiquement, à travers les profils d'exécution des programmes (nombre de fois qu'une donnée est accédée, taille des données, fréquence d'accès, *etc*.). Ainsi, au lieu d'utiliser une heuristique gloutonne, la plupart des auteurs utilisent l'une des trois heuristiques suivantes.

Allouer les données en mémoire SRAM selon leur taille : les données les plus petites sont allouées en SRAM tant qu'il y a de la place disponible, sinon elles sont allouées en mémoire principale DRAM. Cette méthode a l'avantage d'être simple à implanter puisqu'elle ne tient compte que de la taille des données, mais elle a pour inconvénient d'allouer les données les plus grandes en DRAM sachant que si ces données sont souvent accédées, il en résultera très peu d'économie d'énergie.

Allouer les données en mémoire SRAM selon leur nombre d'accès : les données les plus souvent accédées/utilisées sont allouées en SRAM tant qu'il y a de la place disponible, sinon elles sont allouées en mémoire principale. Cette transformation est déjà plus optimale que la première puisque les données les plus souvent accédées/utilisées seront allouées dans une mémoire qui consomme moins d'énergie. Elle permettra donc de réaliser plus d'économies comme cela est expliqué et démontré dans [Sjödin *et al.*, 1998; Steinke *et al.*, 2002]. Cependant, nous pouvons noter des problèmes de granularité dans certains cas. Citons comme exemple, une structure dont seule une partie est très souvent accédée/utilisée.

Allouer les données en mémoire SRAM selon leur nombre d'accès et leur taille (nombre/octet) : c'est en quelque sorte une combinaison des deux heuristiques précédentes. L'idée est de combiner leurs avantages. Si l'on considère l'exemple d'une structure dont seule une partie est la plus souvent accédée/utilisée, on tient compte du nombre moyen d'accès à cette structure. On évite ainsi les problèmes de granularité. Dans cette heuristique, les données sont triées en fonction de leurs ratios (nombre d'accès/taille) dans l'ordre descendant. La donnée ayant le ratio le plus élevé est allouée la première en SRAM tant qu'il y a de la place disponible. Autrement, elle est allouée en mémoire principale. Cette heuristique a l'inconvénient d'utiliser une méthode de tri qui peut être lourde et coûteuse en temps de calcul pour une quantité importante de données. De plus, cette heuristique ne fonctionnera pas très bien dans une perspective dynamique où la capacité maximale de la mémoire SRAM n'est pas connue à l'avance. Il s'agit là de l'heuristique la plus utilisée en littérature que nous appellerons dans la suite (*BEH*[20]). Dans le reste de ce mémoire, nous nous référerons à l'heuristique BEH comme base de nos optimisations de la consommation d'énergie en mémoire.

20. BEH : *Best known Existing Heuristic.*

2.1.7 Plate-forme expérimentale

Notre travail d'implantation et de développement logiciel est basé sur la plate-forme *OTAWA* [21] fournie et développée par nos partenaires de l'IRIT. *OTAWA* [Cassé et Rochange, 2007] est un environnement de simulation basé sur *SystemC* et dédié à l'estimation du temps d'exécution pire-cas (*WCET*) par analyse statique. *OTAWA* est utilisable sous la forme d'une bibliothèque *C ++* sous licence *LGPL* et a été conçu pour supporter différentes architectures comme *PowerPC*, *ARM* ou *M68HC*. D'autres architectures sont en cours de développement comme *TriCore*. Dans notre cas, nous nous intéressons aux architectures *PowerPC*. Les outils d'*OTAWA* chargent et analysent donc un code binaire *PowerPC* et fournissent des mesures de *WCET* (contrôle de flux graphiques, boucle de détection, *etc*.). Pour rappel, nous avons considéré d'autres outils auparavant, ces outils sont décrits dans le rapport technique [Idrissi Aouad et Zendra, 2008]. Ce rapport présente différents outils de caractérisation du comportement mémoire et d'estimation de la consommation énergétique ; il décrit toutes les étapes de compilation et d'installation de chacun de ces outils ainsi que la manière de les utiliser en détaillant leurs entrées/sorties et en expliquant leurs résultats obtenus.

Afin de mesurer l'impact énergétique de l'heuristique BEH présentée en section 2.1.6 et pouvoir le comparer à celui des différentes nouvelles heuristiques proposées dans la partie suivante, une partie du travail consiste à intégrer à *OTAWA* un module dédié à l'énergie. Pour ce faire, nous avons développé et ajouté le modèle d'estimation de la consommation d'énergie présenté en section 1.1.4 dans *OTAWA* et plus précisément dans les modules du simulateur structurel du processeur générique et de la simulation de la mémoire. Ce travail est constitué de deux étapes principales : l'étape de lecture de la configuration et l'étape de calcul.

L'étape de lecture de la configuration signifie la prise en compte de la configuration spécifiée par l'utilisateur de façon automatique. Autrement dit, l'utilisateur spécifie une configuration de l'architecture mémoire ainsi qu'un ensemble de caractéristiques technologiques propres à chaque type de mémoire via un fichier *XML* fourni en entrée, en ligne de commande au moment du lancement de la simulation. Il s'agit là de lire, de façon automatique, cette configuration mémoire ainsi spécifiée en vue de récupérer les différentes valeurs des caractéristiques de chacune des différentes mémoires composant la configuration. *OTAWA* est capable de lire et de prendre en compte la configuration de la mémoire cache (voir l'Annexe B pour un exemple de fichier de description de la mémoire cache utilisé). Cependant, *OTAWA* ne fait que lire la configuration de la hiérarchie mémoire, mais ne la prend pas encore en compte dans la simulation (voir l'Annexe C pour un exemple de fichier de description de la hiérarchie mémoire). La SPM n'est donc pas prise en compte. Étant donné que la prochaine étape (le calcul) dépend, partiellement, des valeurs issues de cette première partie (la lecture de la configuration) et vu l'importance d'utiliser une mémoire Scratch-Pad dans notre architecture mémoire, nous avons dû contourner ce problème.

C'est pourquoi, nous avons développé une solution semi-automatique qui est basée sur le langage et le solveur *MATLAB* [22] et sur l'outil d'instrumentation de code *Gcov*. *Gcov* permet de savoir précisément quelles lignes de codes sont effectivement exécutées et combien de fois une ligne de code donnée, de chaque programme test, est exécutée. Autrement dit, *Gcov* permet de générer pour chacun des programmes tests utilisés des données de profils (*data profiling*). Ainsi, pour chaque programme test, un fichier texte est généré. Ce fichier texte contient des informations concernant les données utilisées par le programme test en question. Ces données sont collectées en analysant, de façon statique, le code source du programme test (taille des données) et dynamiquement en utilisant *Gcov* (nombre de fois qu'une don-

21. OTAWA : *Open Tool for Adaptative WCET Analysis.*
22. MATLAB : *MATrix LABoratory.*

née est accédée, fréquence d'accès). Ce procédé fournit à l'heuristique en question (dans l'étape suivante) suffisamment d'informations pour décider du meilleur placement en mémoire réduisant l'énergie globale consommée en mémoire. Une fois ces informations connues, il est possible de vérifier si l'on peut optimiser les parties les plus coûteuses [Idrissi Aouad et Zendra, 2008].

L'étape de calcul, se divise en deux parties ; la partie nombre d'accès et la partie énergie :

- Pour la partie nombre d'accès, nous avons utilisé les différentes heuristiques proposées. Selon la politique de l'heuristique considérée, une combinaison optimale de données à allouer dans telle ou telle mémoire est générée et une allocation optimale en mémoire est déterminée. Cette allocation se traduit par les nombres d'accès à chacune des mémoires de notre architecture. Ces nombres d'accès représentent les différents termes N_{spmr}, N_{spmw}, N_{icr}, N_{icw}, N_{dramr} et N_{dramw} décrits dans le tableau 1.3 (page 20).
- Quant à la partie énergie, nous avons utilisé l'outil d'estimation de consommation *CACTI* [Wilton et Jouppi, 1996]. *CACTI* collecte les informations concernant l'énergie par accès à chaque type de mémoire (le lecteur intéressé trouvera plus de détails dans le rapport technique de CACTI 4.0 [Tarjan *et al.*, 2006]). Ces valeurs sont celles des termes E_{spmr}, E_{spmw}, E_{icr}, E_{icw}, E_{dramr} et E_{dramw} décrits dans le tableau 1.3 (page 20).

Une fois toutes les valeurs récupérées (valeurs de la formule page 19), nous pouvons calculer les énergies consommées par les différentes heuristiques. Notons également que les différents algorithmes et heuristiques proposés dans la seconde partie de ce mémoire, ainsi que l'heuristique BEH ont été développés avec les langages *C* et *C++*. Tous les résultats présentés sont exprimés en nanoJoules (*nJ*).

Dans toutes les expérimentations, 30 différentes exécutions, pour chacune des heuristiques présentées dans la suite, sont générées étant donné que la solution trouvée diffère d'une exécution à l'autre et cela afin d'obtenir une estimation non biaisée et convergente. Les résultats moyens et les meilleurs résultats sont enregistrés. Pour l'heuristique BEH, la solution trouvée ne change pas d'une exécution à l'autre (car c'est toujours un tri). Dans ce qui suit, nous générons pour chaque comparaison d'heuristiques deux graphiques différents : un, dans le cas de la politique d'écriture simultanée (WT) et un autre, dans le cas de la politique d'écriture différée (WB). Comme les allures des deux graphiques obtenus sont pratiquement les mêmes, seuls les résultats obtenus en considérant la politique d'écriture différée sont donnés. Sachant que $E_{WTmode} \neq E_{WBmode}$.

2.2 Optimisation de fonctions tests mathématiques

2.2.1 Introduction

L'optimisation globale [Weise, 2009] est la branche des mathématiques appliquées et des analyses numériques qui se concentrent sur l'optimisation. L'optimisation globale peut être définie comme suit : $Minimiser\ f(x) : S \rightarrow R$ où $f(.)$ est la fonction objectif (appelée également *fitness*) sujette à l'optimisation, $S \subset \Re^D$ et D est la dimension de l'espace de recherche S. Résoudre les problèmes d'optimisation globale signifie trouver $x^* \in S$ tel que $f(x^*) \leq f(x), \forall x \in S$. x^* est appelé le minimiseur global de $f(.)$ et $f(x^*)$ est appelée la valeur minimum globale de $f(.)$. Trouver ce minimum global est très difficile du fait de l'existence de plusieurs optima locaux.

2.2.2 Méthodes basées sur une métaheuristique

Au cours des dernières décennies, plusieurs heuristiques d'optimisation ont été proposées pour résoudre l'optimisation globale. Parmi ces heuristiques variées, mentionnons le recuit simulé (*SA*) [Locatelli, 2002], les algorithmes génétiques (*GAs*) [Zheng et Kiyooka, 1999] et les algorithmes d'optimisation qui utilisent les comportements sociaux et évolutifs comme la méthode d'optimisation par essaims particulaires (*PSO*) [Pant *et al.*, 2008; Kennedy et Russell, 2001; Marco et Stützle, 2008; Engelbrecht, 2005; Poli *et al.*, 2007]. La méthode PSO est relativement populaire pour résoudre des problèmes d'optimisation complexes, mais elle possède quelques limitations dont principalement la convergence prématurée. Afin d'éviter cette convergence prématurée de PSO [Ratnaweera *et al.*, 2004], plusieurs travaux tentent d'hybrider PSO avec d'autres algorithmes [Yu-Xuan *et al.*, 2010; Premalatha et Natarajan, 2010; Liang *et al.*, 2006]. D'autres travaux proposent différentes règles de mouvement pour résoudre ce problème (voir [Yu-Xuan *et al.*, 2010; Montes de Oca et Stützle, 2008; Engelbrecht, 2005; Poli *et al.*, 2007]).

Dans [Liang *et al.*, 2006], l'algorithme *Comprehensive Learning PSO* (*CLPSO*) est proposé. Pour chaque dimension d'une particule, un nombre aléatoire est généré. Si ce nombre aléatoire est supérieur à une probabilité d'apprentissage, la dimension correspondante va apprendre de sa propre meilleure position précédente *pbest* sinon, elle va apprendre du *pbest* d'une autre particule selon une procédure de sélection par tournoi. Les trois principales différences entre *CLPSO* et le PSO classique sont :

1. Au lieu d'utiliser les propres *pbest* et *gbest* de la particule comme exemplaires, les *pbests* de toutes les particules peuvent potentiellement être utilisés comme exemplaires afin de guider la direction de déplacement de la particule.
2. Au lieu d'apprendre du même exemplaire de particule pour toutes les dimensions, chaque dimension d'une particule peut apprendre de la dimension correspondante de différents *pbests* de particules.
3. Au lieu d'apprendre de deux exemplaires (*pbest* et *gbest*), à la fois, à chaque génération comme dans le PSO classique, chaque dimension d'une particule apprend juste d'un seul exemplaire pour quelques générations.

Les auteurs réussissent à optimiser la plupart des fonctions tests mathématiques par rapport à d'autres algorithmes de la littérature mais pas tous.

2.2.3 Méthodes basées sur une hybridation de métaheuristiques

2.2.3.1 PSO - recherche locale

D'autres auteurs proposent des méthodes hybrides basées sur PSO et une méthode de recherche locale. L'algorithme *Tabu List PSO* (*TL-PSO*) décrit dans [Nakano *et al.*, 2008] est basé sur la combinaison de PSO et de la recherche avec tabous (*TS*). Dans PSO, lorsqu'une particule trouve une solution optimale locale, toutes les autres particules se rassemblent autour d'elle et se retrouvent coincées dans cette solution locale. En revanche, TS peut s'enfuir de cette solution optimale locale en se déplaçant loin de la meilleure solution courante. La méthode *TL-PSO* essaie de combiner les avantages des deux méthodes PSO et TS. Pour ce faire, *TL-PSO* stocke l'historique des meilleures informations détenues par chaque particule (*pbest*) dans une liste tabou. Lorsque la capacité de recherche d'une particule se trouve réduite, la particule sélectionne un *pbest* précédent à partir des valeurs de l'historique et l'utilise pour mettre à jour l'équation de vélocité. Cela permet à chaque particule de rester active et la capacité de recherche de l'essaim progresse. La méthode proposée dans cet article est vérifiée par des simulations qui montrent

son efficacité sur un ensemble de fonctions tests mathématiques en comparaison avec des méthodes proposées dans la littérature. Cependant, *TL-PSO* obtient de mauvais résultats en ce qui concerne la fonction unimodale de Rosenbrock et n'arrive pas à améliorer ses résultats.

Dans [Wang *et al.*, 2007], lors de l'exécution de PSO, si la meilleure valeur courante de la fonction *fitness* trouvée par l'essaim ne s'améliore plus depuis un certain nombre d'évaluations, deux actions sont entreprises. D'abord, les auteurs mettent la meilleure position courante dans une liste tabou et considèrent son voisinage comme étant tabou et cela afin d'éviter des attractions répétées vers cette zone dans les prochaines itérations. Ensuite, ils génèrent de façon répétitive des directions appropriées pour que la particule puisse s'extraire de ce minimum local jusqu'à ce qu'elle se retrouve dans une zone non-tabou. Les résultats des simulations montrent l'efficacité de la méthode sur cinq fonctions tests mathématiques en comparaison avec le PSO classique et une méthode proposée dans la littérature. Cependant, les auteurs obtiennent de mauvais résultats en ce qui concerne les fonctions de Rosenbrock et de Griewank.

Certains auteurs proposent des méthodes hybrides basées sur PSO et SA. Ainsi, dans [Premalatha et Natarajan, 2010], lorsqu'une particule du PSO stagne, SA est appliqué sur le résultat pour diversifier la position de la particule même si la solution s'empire. D'autres travaux du même genre peuvent être trouvés dans [Wang et Li, 2004; Idoumghar *et al.*, 2009].

2.2.3.2 PSO - opérateur génétique

Enfin, d'autres techniques hybrides se basent sur PSO et sur un des deux opérateurs génétiques (croisement ou mutation). Ainsi, dans [Pant *et al.*, 2008], les auteurs ont évalué les performances de quatre algorithmes basés sur l'optimisation par essaims particulaires. Ces algorithmes sont : l'algorithme PSO classique, *Attraction-Repulsion based PSO* (*ATREPSO*), *Quadratic Interpolation based PSO* (*QIPSO*) et *Gaussian Mutation based PSO* (*GMPSO*). Tandis que tous les algorithmes présentés dans cet article sont guidés par la diversité de la population afin de rechercher la solution globale optimale d'un problème d'optimisation donné, *GMPSO* utilise le concept de mutation et *QIPSO* utilise l'opérateur de reproduction pour générer un nouveau membre de l'essaim. En effet, l'algorithme *ATREPSO* est un algorithme sur trois phases dans lequel les particules de l'essaim passent alternativement entre la phase d'attraction (quand les particules sont attirées vers l'optimal global), la phase de répulsion (les particules sont repoussées de la position optimale) et la phase intermédiaire (combinaison équilibrée d'attraction et de répulsion). Le déplacement des particules de l'essaim dans les différentes phases est contrôlé par la diversité de l'essaim. L'algorithme *QIPSO* utilise le concept de reproduction basé sur la "'polygamie"'. Il utilise la diversité de l'essaim comme mesure de guidage du croisement. Ainsi, lorsque la diversité passe en dessous d'un seuil minimal, l'opérateur de croisement quadratique non-linéaire est activé jusqu'à ce que la diversité atteigne un seuil maximal. Les seuils, minimal et maximal, de diversité sont prédéfinis par l'utilisateur. L'algorithme *GMPSO* utilise les équations générales de position et de vélocité de la méthode PSO classique auxquelles il rajoute un opérateur de mutation généré par une distribution gaussienne. Là encore, c'est la diversité de l'essaim qui décide quand appliquer la mutation (en dessous d'un seuil de diversité). Les résultats de simulation obtenus sur des fonctions tests mathématiques montrent l'efficacité des algorithmes proposés et particulièrement celle de *QIPSO*.

L'algorithme *Gaussian PSO with jumps* (*GPSO-J*) proposé dans [Krohling, 2005] est fondé sur des sauts pour s'échapper des minima locaux. Lorsqu'il n'y a plus d'améliorations de la fonction fitness (minimum local), un compteur est incrémenté à chaque itération. Une fois que ce compteur atteint un seuil pré-spécifié, la particule effectue un saut à un nouveau point. Cela est effectué grâce à l'introduction d'un opérateur de mutation qui change la position de la particule. L'opérateur de mutation est basé

sur les distributions gaussienne et de Cauchy. *GPSO-J* améliore les résultats obtenus sur des fonctions tests mathématiques en comparaison avec des algorithmes de la littérature basés sur PSO, excepté pour la fonction de Griewank.

Dans l'algorithme *Particle Swarm Optimizer with Adaptive Tabu and Mutation* (*ATM-PSO*) décrit dans [Yu-Xuan *et al.*, 2010], des mutations sont effectuées sous le guidage d'informations tabous de telle sorte que plusieurs mutations redondantes sont évitées. *ATM-PSO* consiste en deux phases :

1. La phase de détection tabou : durant l'exécution du PSO classique, pour une particule, s'il n'y a plus d'améliorations de sa fonction fitness pendant des générations successives ; sa meilleure position est placée dans une liste tabou. Après quoi, toutes les particules proches de cette particule sont mutées en utilisant une variable aléatoire de Cauchy.
2. La phase de mutation et de contrôle : dans les générations suivantes, la position de chaque particule est contrôlée. Lorsqu'une particule se dirige vers sa surface tabou, la mutation se produit. Un compteur tabou est assigné à chaque particule. Seules les particules ayant le compteur tabou le plus élevé seraient mutées.

2.3 Conclusion

Dans ce chapitre, la section 2.1 donne un résumé de la plupart des stratégies et des approches existantes qui permettent de gérer de façon optimale l'énergie consommée dans différents types d'architectures mémoire. Cet état de l'art essaye de couvrir un large champ de problèmes et leurs solutions. Ce travail n'a pas la prétention d'être exhaustif. Il permet en revanche, d'avoir une vue globale et précise de ce qui a été fait dans la plupart des domaines de la gestion mémoire : le choix optimal d'une allocation mémoire, la localité, le partitionnement mémoire, l'organisation de la mémoire en bancs ainsi que l'utilisation des modes de faible puissance. Certains problèmes, dans cette section comme la réplication ou la consommation énergétique des systèmes multi-processeurs, n'ont pas été couverts en profondeur du fait que ce sont toujours des domaines de recherche active. Nous nous sommes également limités à l'étude des techniques d'optimisations logicielles économisant la consommation d'énergie dans tous les types de mémoire. La sous-section 2.1.6 fait ressortir de manière plus concise les principales heuristiques utilisées dans les différents travaux cités dans ce chapitre. Dans le reste de ce mémoire, nous nous référerons à l'heuristique BEH comme base de nos optimisations de la consommation d'énergie en mémoire. Notons enfin la sous-section 2.1.7 où la présentation est faite du fonctionnement de notre plate-forme expérimentale dans le but d'alléger la lecture de la partie suivante où les résultats seront directement donnés.

La section 2.2 quant à elle, propose une sélection de travaux qui tentent de résoudre la problématique de l'optimisation globale en se basant sur les principes des métaheuristiques simples ou combinées, voire hybridées entre elles. Ainsi, après avoir cité quelques travaux utilisant une seule métaheuristique à la fois parmi le recuit simulé, les algorithmes génétiques et l'optimisation par essaims particulaires, nous avons présenté d'autres travaux qui proposent des méthodes hybrides basées sur différentes métaheuristiques. La prévalence de la méthode PSO dans ces algorithmes s'explique par la popularité de celle-ci. En effet, de plus en plus d'études portent sur PSO et tentent de l'hybrider avec d'autres méthodes afin de décupler son potentiel au maximum.

Ce qui ressort de ce travail d'inventaire, avant toute chose, c'est le fait qu'aucune méthode existante pour réduire la consommation d'énergie en mémoire ne se base sur l'approche des métaheuristiques, à

l'instar de ce qui se fait en optimisation globale. L'idée alors est de proposer des algorithmes de type métaheuristiques afin d'essayer de résoudre ce problème. C'est justement le sujet de la prochaine partie. Le chapitre suivant présentera brièvement quelques unes de ces métaheuristiques ainsi que leurs avantages par rapport à des méthodes de recherche itérative. Puis, nous proposerons de nouveaux algorithmes basés sur ces principes et adaptés au cas de la réduction de la consommation d'énergie en mémoire dans les systèmes embarqués.

Deuxième partie

Contributions

Chapitre 3

Métaheuristiques

Sommaire

3.1 Introduction

Beaucoup de problèmes de grande complexité, dans des secteurs techniques très divers, sont difficilement solubles. La difficulté ne vient pas de la complexité du problème, mais de la taille excessive de l'espace des solutions. Par exemple, le problème du voyageur de commerce a une taille de l'espace de solutions variant en factorielle $(n-1)$ où n est le nombre de villes où il faut passer. On s'aperçoit qu'à seulement 100 villes, il y a 9.3×10^{153} solutions. Il est alors impensable de pouvoir les tester

toutes pour trouver la meilleure. Le problème à résoudre peut souvent s'exprimer comme un *problème d'optimisation* : on définit une fonction objectif, ou fonction de coût (il peut y en avoir plusieurs), que l'on cherche à minimiser ou à maximiser par rapport à tous les paramètres concernés. La définition du problème d'optimisation est souvent complétée par la donnée de *contraintes* : tous les paramètres des solutions retenues doivent respecter ces contraintes, faute de quoi ces solutions ne sont pas réalisables. Dans cette thèse, nous nous intéressons à un groupe de méthodes, dénommées *métaheuristiques* [Dréo *et al.*, 2003] qui ont été essayées pour la résolution de problèmes dits d'*optimisation difficile*.

3.2 Pourquoi les métaheuristiques ?

On distingue, en réalité, deux types de problèmes d'optimisation : les problèmes discrets et les problèmes à variables continues. Pour fixer les idées, citons deux exemples. Parmi les problèmes discrets, on trouve le célèbre problème du voyageur de commerce, qui doit visiter un certain nombre de villes, avant de retourner à la ville de départ. Un exemple classique de problème continu est celui de la recherche des valeurs à affecter aux paramètres d'un modèle numérique de processus pour que ce modèle reproduise au mieux le comportement réel observé. En pratique, on rencontre aussi des problèmes comportant, à la fois, des variables discrètes et des variables continues. Cette différenciation est nécessaire pour cerner le domaine de l'optimisation difficile. En effet, deux sortes de problèmes reçoivent, dans la littérature, cette appellation :

- certains problèmes d'optimisation discrète, pour lesquels on ne connaît pas d'algorithme exact polynomial (c'est-à-dire dont le temps de calcul est proportionnel à N^n où N désigne le nombre de paramètres inconnus du problème et n une constante entière). C'est le cas, en particulier, des problèmes dits "NP-difficiles", pour lesquels on conjecture qu'il n'existe pas de constante n telle que le temps de résolution soit borné par un polynôme de degré n.
- certains problèmes d'optimisation à variables continues, pour lesquels on ne connaît pas d'algorithme permettant de repérer un optimum global (c'est-à-dire la meilleure solution possible), à coup sûr, et en un nombre fini de calculs.

Des efforts ont longtemps été menés, séparément, pour résoudre ces deux types de problèmes. Dans le domaine de l'optimisation continue, il existe un nombre de méthodes classiques dites d'*optimisation globale*, mais ces techniques sont souvent inefficaces si la fonction objectif ne possède pas une propriété structurelle particulière telle que la convexité. Dans le domaine de l'optimisation discrète, un grand nombre d'heuristiques, qui produisent des solutions proches de l'optimum, ont été développées, mais la plupart d'entre elles ont été conçues spécifiquement pour un problème donné.

L'arrivée des métaheuristiques marque une réconciliation des deux domaines : en effet, celles-ci s'appliquent à toutes sortes de problèmes discrets et elles peuvent s'adapter aussi aux problèmes continus. Ces méthodes ont en commun, en outre, les caractéristiques suivantes :

- elles sont, au moins pour partie, *stochastiques* : cette approche permet de faire face à l'explosion combinatoire des possibilités ;
- généralement d'origine discrète, elles ont l'avantage décisif, dans le cas continu, d'êtres discrètes, c'est-à-dire qu'elles ne recourent pas au calcul, souvent problématique, des gradients de la fonction objectif ;
- elles sont inspirées par des *analogies* : avec la physique (recuit simulé, ...), avec la biologie (algorithmes génétiques, recherche avec tabous, ...) ou avec l'éthologie (essaims particulaires, ...) ;

– elles partagent aussi les mêmes inconvénients : les difficultés de réglages des paramètres de la méthode et le temps de calcul élevé.

Ces méthodes ne s'excluent pas mutuellement : en effet, il est le plus souvent impossible de prévoir, avec certitude, l'efficacité d'une méthode donnée quand elle est appliquée à un problème donné. Soulignons enfin, une autre richesse des métaheuristiques : elles se prêtent à toutes sortes d'extensions :

– l'optimisation *multiobjectif* où il s'agit d'optimiser simultanément plusieurs objectifs contradictoires ;
– l'optimisation *multimodale* où l'on s'efforce de repérer tout un jeu d'optima globaux ou locaux ;
– l'optimisation *dynamique* qui fait face à des variations temporelles de la fonction objectif ;
– le recours à des *implémentations parallèles*.

La figure 3.1 propose une classification générale des méthodes d'optimisation mono-objectif [Collette et Siarry, 2002]. Parmi les métaheuristiques, on peut différencier entre les métaheuristiques de *voisinage*, qui font progresser une seule solution à la fois (recuit simulé, recherche avec tabous, ...) et les métaheuristiques distribuées, qui manipulent en parallèle toute une population de solutions (algorithmes génétiques, optimisation par essaims particulaires, ...).

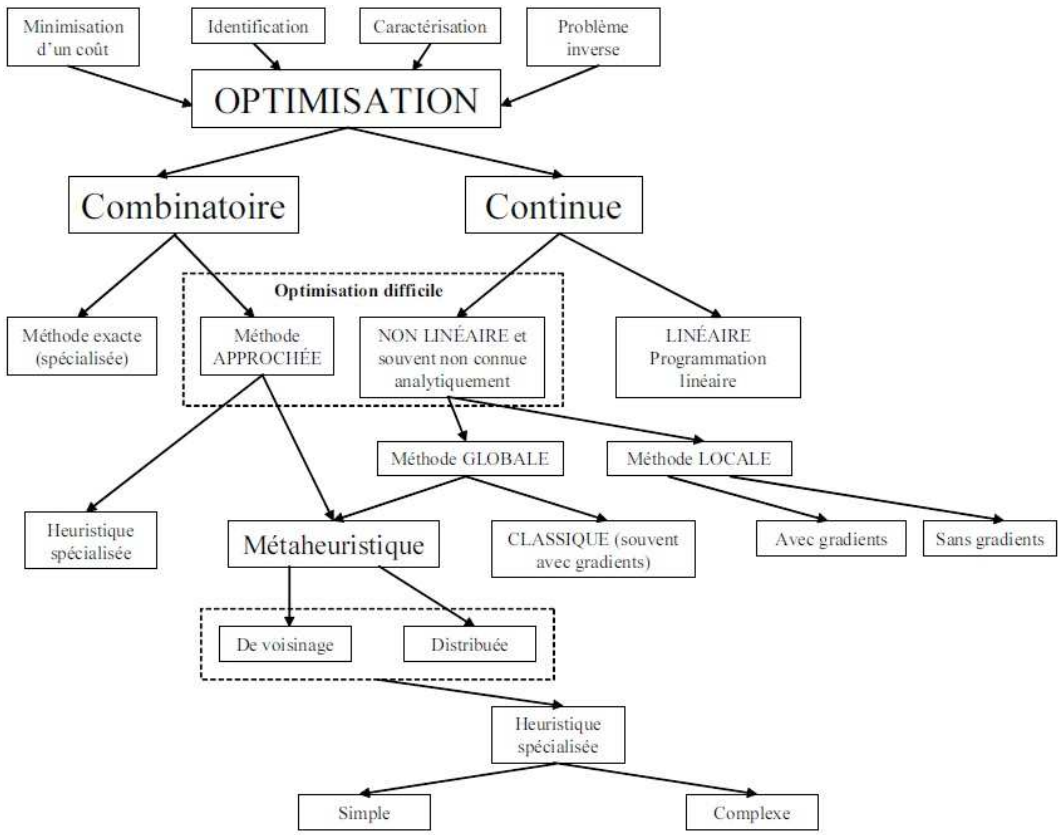

FIGURE 3.1 – Classification générale des méthodes d'optimisation mono-objectif [Collette et Siarry, 2002].

Ces contextes particuliers requièrent, de la part des méthodes de résolution, des propriétés spécifiques qui ne sont pas présentes dans toutes les métaheuristiques. Le réglage et la comparaison des métaheuristiques sont souvent effectués empiriquement en exploitant des jeux de fonctions analytiques de test où les minima globaux et locaux sont connus. Avant de présenter la démarche fructueuse des métaheuristiques, nous commencerons par expliquer l'échec d'un algorithme itératif classique.

3.3 Minimum local vs minimum global

3.3.1 Approche d'un algorithme itératif classique

Le principe d'un algorithme classique d'amélioration itérative est le suivant : on part d'une configuration initiale c_0, qui peut être choisie au hasard, ou qui peut être donnée par un concepteur. On essaie alors une modification élémentaire, souvent appelée *mouvement* (par exemple, on permute deux composants choisis au hasard, ou bien on translate l'un d'entre eux), puis on compare les valeurs de la fonction objectif, avant et après cette modification. Si le changement conduit à une diminution de la fonction objectif, il est accepté et la configuration c_1 obtenue, qui est voisine de la précédente, sert de point de départ pour un nouvel essai. Dans le cas contraire, on revient à la configuration précédente, avant de faire une autre tentative. Le processus est itéré jusqu'à ce que toute modification rende le résultat moins bon. La figure 3.2 montre que cet algorithme d'amélioration itérative (désigné aussi sous les termes de *méthode classique* ou *méthode de descente*) ne conduit pas, en général, au minimum absolu, mais seulement à un minimum local c_n, qui constitue la meilleure des solutions accessibles, compte tenu de l'hypothèse initiale.

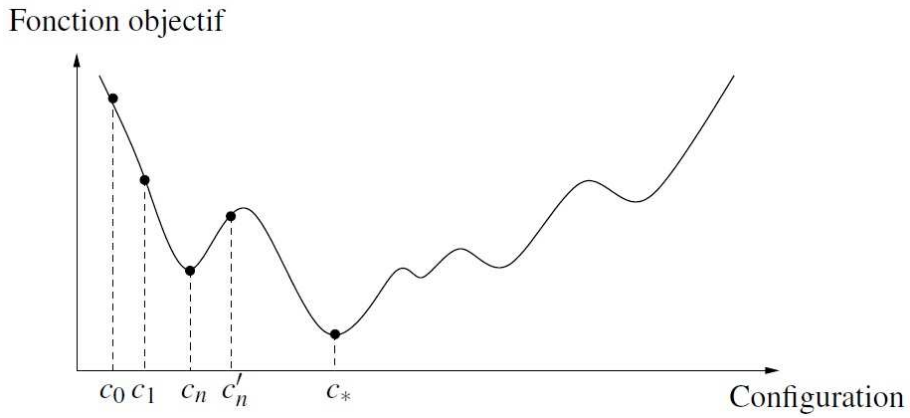

FIGURE 3.2 – Allure de la fonction objectif d'un problème d'optimisation difficile en fonction de la configuration [Dréo *et al.*, 2003].

Pour améliorer l'efficacité de la méthode, on peut, bien entendu, l'appliquer plusieurs fois avec des conditions initiales différentes choisies arbitrairement et retenir comme solution finale le meilleur des minima locaux obtenus. Cependant, cette procédure augmente sensiblement le temps de calcul de l'algorithme et ne garantit pas de trouver la configuration optimale c_*. L'application répétée d'une méthode de descente est particulièrement inefficace lorsque le nombre de minima locaux croît exponentiellement avec la taille du problème.

3.3.2 Approche des métaheuristiques

Pour surmonter l'obstacle des minima locaux, une autre idée s'est montrée très fructueuse, au point qu'elle est à la source de toutes les métaheuristiques dites de *voisinage* (recuit simulé, recherche avec tabous) : il s'agit d'autoriser, de temps en temps, des mouvements de remontée, autrement dit, d'accepter une dégradation temporaire de la situation lors du changement de la configuration courante. C'est le cas si l'on passe de c_n à c'_n (voir figure 3.2). Un mécanisme de contrôle des dégradations, spécifique à chaque métaheuristique, permet d'éviter la divergence du procédé. Il devient dès lors possible de s'extraire du piège que représente un minimum local pour partir explorer une autre vallée plus prometteuse. Les métaheuristiques distribuées, (telles que les algorithmes évolutionnaires) ont, elles aussi, des mécanismes permettant la sortie d'une solution particulière hors d'un puits local de la fonction objectif. Ces mécanismes (comme la *mutation* dans les algorithmes génétiques) affectant une solution viennent, dans ce cas, seconder le mécanisme collectif de lutte contre les minima locaux, que représente le contrôle en parallèle de toute une *population* de solutions.

Il existe un grand nombre de métaheuristiques différentes allant de la simple recherche locale à des algorithmes complexes de recherche globale. Dans cette thèse, nous distinguons entre les métaheuristiques qui travaillent avec une population de solutions : les méthodes évolutives de celles qui ne manipulent qu'une seule solution à la fois : les méthodes de recherche locale.

3.4 Les méthodes de recherche locale

Les méthodes qui tentent itérativement d'améliorer une solution sont appelées méthodes de recherche locale. En effet, face à une solution d'un problème dont on n'est pas capable de trouver une solution optimale, on essaie de modifier légèrement la solution proposée et de vérifier qu'il n'est pas possible d'obtenir de meilleures solutions en procédant à des changements locaux. En d'autres termes, on s'arrête dès qu'on rencontre un optimum local relatif aux modifications qu'on s'autorise à faire sur une solution. Avec un tel processus, rien n'indique que la solution ainsi obtenue soit un optimum global et en pratique c'est rarement le cas. Afin de pouvoir trouver des solutions meilleures que le premier optimum local rencontré, on peut essayer de poursuivre le processus de modifications locales, mais si l'on ne prend pas de précautions, on s'expose à visiter cycliquement un nombre restreint de solutions. La recherche avec tabous ou le recuit simulé sont deux techniques de recherche locale qui tentent de remédier à cet inconvénient.

3.4.1 Recherche avec tabous

3.4.1.1 Description

La méthode de recherche avec tabous (*TS*[23]) [Gendreau, 2003] a été formalisée en 1986 par Glover. Sa principale particularité tient dans la mise en œuvre de mécanismes inspirés de la mémoire humaine. Le principe de base de la méthode de recherche avec tabous est simple : elle fonctionne avec une seule configuration courante à la fois (au départ, une solution aléatoire), qui est actualisée au cours d'itérations successives. À chaque itération, le mécanisme de passage d'une configuration s à la suivante t comporte deux étapes :

23. TS : *Tabu Search.*

Algorithm 1 Un algorithme générique de la recherche avec tabous.

```
1: Generate an initial solution s_0 ; s := s_0
2: s* := s ; f* := f(s)
3: T := ∅ {tabu list}
4: repeat
5:    m := the best movement among the non tabu movements and the exceptional tabu movements
      (aspiration criterion)
6:    s := s(+)m
7:    if f(s) < f(s*) then
8:       s* := s
9:       f* := f(s)
10:   end if
11:   update T
12: until goal is satisfied or the stopping condition is reached
13: return s*
```

- on construit l'ensemble des voisins de s, c'est-à-dire l'ensemble des configurations accessibles en un seul mouvement (une modification apportée à la solution courante) élémentaire à partir de s, soit $N(s)$ l'ensemble de ces voisins ;
- on évalue la fonction objectif f du problème dans chacune des configurations appartenant à $N(s)$. La configuration t, qui succède à s dans la suite de solutions construite par la méthode de recherche avec tabous, est la configuration de $N(s)$ en laquelle f prend la valeur minimale. Notons que cette configuration t est adoptée même si elle est moins bonne que s, $i.e.$ si $f(t) > f(s)$: c'est grâce à cette particularité que la méthode de recherche avec tabous permet d'éviter le piégeage dans les minima locaux de f.

Telle qu'elle est décrite, la procédure précédente est inopérante, parce qu'il y a le risque important de retourner à une configuration déjà retenue lors d'une itération précédente, ce qui engendre un cycle. Pour éviter ce phénomène, on tient à jour, puis on exploite, à chaque itération, une liste de mouvements interdits, la "liste de tabous" : cette liste, qui a donné son nom à la méthode, contient m mouvements $(t \to s)$, qui sont les inverses des m derniers mouvements $(s \to t)$ effectués. L'algorithme général de la méthode de recherche avec tabous est donné dans l'algorithme 1.

L'algorithme modélise ainsi une forme rudimentaire de mémoire, la mémoire à court terme des solutions visitées récemment. Deux mécanismes supplémentaires, nommés intensification et diversification, sont souvent mis en œuvre pour doter aussi l'algorithme d'une mémoire à long terme. Ces processus n'exploitent plus la proximité dans le temps d'événements particuliers, mais plutôt la fréquence de leur occurrence, sur une période plus longue. L'intensification consiste à approfondir l'exploration de certaines régions de l'espace des solutions identifiées comme particulièrement prometteuses. La diversification est, au contraire, la réorientation périodique de la recherche d'un optimum vers des régions très rarement visitées jusqu'ici.

Pour certains problèmes d'optimisation, la méthode de recherche avec tabous a donné d'excellents résultats ; en outre, sous sa forme de base, la méthode comporte moins de paramètres de réglage que le recuit simulé, ce qui la rend plus simple d'emploi. Cependant, les divers mécanismes annexes, comme l'intensification et la diversification, apportent une notable complexité.

3.4.1.2 Cas de la réduction de la consommation d'énergie

Dans un premier temps, nous avons proposé, dans [Idrissi Aouad *et al.*, 2010d], une heuristique basée sur les principes de la méthode de recherche avec tabous. Là, la SPM, d'une capacité maximale C, est remplie avec une combinaison de données d'une liste de N données possibles, chacune de taille $size_i$ et de nombre d'accès an_i de telle sorte que la somme des nombres d'accès des données allouées en SPM soit maximisée. Dans l'algorithme de type recherche avec tabous développé, noté *TS*, une solution initiale est d'abord générée de façon aléatoire. Comme N constitue le nombre total de données possibles, alors une solution est une séquence s finie de N termes telle que $s[n]$ est soit 0, soit la taille de la n_{ieme} donnée. $s[n] = 0$ si et seulement si la n_{ieme} donnée n'est pas sélectionnée dans la solution. Bien entendu, cette solution doit respecter la contrainte de ne pas dépasser la capacité maximale de la SPM (*i.e.* $\sum_{i=1}^{N} s[i] \leq C$). Un nombre maximal d'itérations et une durée de vie pour la liste de tabous sont également fixés. Initialement, la solution optimale est égale à la solution initiale, le nombre d'accès optimal est égal au nombre d'accès de la solution initiale et la liste de tabous est vide. Tant que le nombre d'itérations n'est pas dépassé, on répète ce qui suit :

- Le e^{ieme} voisinage de la solution courante est généré.
- Une nouvelle matrice contenant les vecteurs voisins est calculée.
- Sur la base des solutions contenues dans cette matrice, un vecteur de valeurs correspondants à la taille courante et un vecteur de valeurs correspondants au nombre d'accès courant sont calculés.
- La meilleure solution est gardée dans le voisinage.
- La liste de tabous est mise à jour afin de permettre une transition vers l'ancienne solution qui était impossible pour une période.
- Enfin, la mise à jour est effectuée si ce nouveau nombre d'accès est meilleur que le nombre d'accès optimal existant.

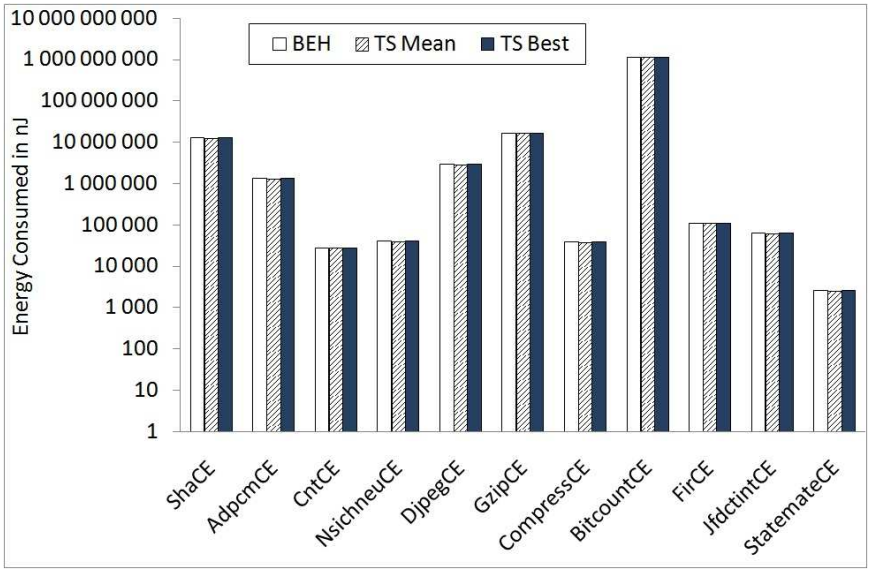

FIGURE 3.3 – Énergies consommées par notre algorithme de type recherche avec tabous.

La figure 3.3, présente les résultats obtenus sur les programmes tests en comparant notre heuristique TS à l'heuristique de base utilisée dans la littérature : BEH (présentée en section 2.1.6). Rappelons que 30 différentes exécutions de l'heuristique sont générées. *TS Best* fait référence à la meilleure solution

trouvée par TS parmi les 30 exécutions alors que *TS Mean* fait référence à la solution moyenne obtenue sur les 30 exécutions de TS. Comme on peut le constater, *TS Best* atteint les mêmes performances énergétiques que BEH. De plus, *TS Mean* produit, à peu près, les mêmes gains énergétiques. Le fait que les programmes tests utilisés contiennent des données uniformes conduit à un grand nombre de minima locaux, et il est vraissemblable que les deux méthodes ont été piégées dans l'un d'entre eux.

BEH étant une méthode de tri des nombres d'accès/tailles comme cela est expliqué en section 2.1.6, cette méthode de tri peut donc être lourde et coûteuse en temps de calcul pour une quantité importante de données. De plus, cette heuristique ne fonctionnera pas très bien dans une perspective dynamique où la capacité maximale de la SPM n'est pas connue à l'avance. Ainsi, l'heuristique TS constitue une alternative à l'heuristique BEH présentée.

3.4.2 Recuit simulé

3.4.2.1 Description

Le recuit simulé (*SA* [24]) est une métaheuristique inspirée d'un processus utilisé en métallurgie [Kirkpatrick *et al.*, 1983]. Pour modifier l'état d'un matériau, on dispose d'un paramètre de commande : la température. La technique consiste à chauffer préalablement le matériau pour lui conférer une énergie élevée. Puis on refroidit lentement le matériau en marquant des paliers de température de durée suffisante. Si la descente en température est très rapide, il apparaît des défauts et on obtient une structure métastable caractérisée par des minima locaux d'énergie. Cette stratégie de baisse contrôlée de la température permet d'obtenir un état solide cristallisé, qui est un état stable, correspondant à un minimum absolu d'énergie.

En pratique, la technique exploite l'algorithme de *Metropolis* qui permet de décrire le comportement d'un système en "équilibre thermodynamique" à une certaine température T : partant d'une configuration donnée (par exemple, un placement initial aléatoire de tous les composants) ; on fait subir à la solution courante s_c une modification élémentaire (par exemple, on translate un composant ou bien, on échange deux composants). Si cette nouvelle solution s_n a pour effet de diminuer la fonction objectif (ou énergie) du système, elle est acceptée ; si elle provoque, au contraire, une augmentation $\Delta = f(s_n) - f(s_c)$ de la fonction objectif, elle peut être acceptée tout de même, mais avec une certaine probabilité comme cela est expliqué dans l'équation 3.1.

$$p = \begin{cases} 1 & \text{si} \quad f(s_n) - f(s_c) < 0 \\ exp(\frac{-|f(s_n)-f(s_c)|}{T}) & \text{sinon} \end{cases} \tag{3.1}$$

On itère ensuite ce procédé en gardant la température constante jusqu'à ce que l'équilibre thermodynamique soit atteint, concrètement au bout d'un nombre "suffisant" de modifications. On abaisse alors la température, avant d'effectuer une nouvelle série de transformations. Dans notre cas, nous utilisons le procédé suivant pour réduire la température, où $i = 0, 1, ...$ et $\gamma = 0.99$.

$$T_{i+1} = \gamma \ T_i \tag{3.2}$$

Le rôle de la température est primordial : à haute température, $exp(\frac{-\Delta}{T})$ est voisin de 1, donc la plupart des mouvements sont acceptés ; à basse température, $exp(\frac{-\Delta}{T})$ est voisin de 0, donc la plupart des mouvements augmentant l'énergie sont refusés ; à température intermédiaire, l'algorithme autorise, de temps en temps, des transformations qui dégradent la fonction objectif : il laisse ainsi au système une

24. SA : *Simulated Annealing.*

Algorithm 2 Un algorithme générique du recuit simulé.

```
1: Random initial solution s_0 ; s_c := s_0
2: T := T_0
3: repeat
4:    nb_moves := 0
5:    for i = 1 to iter do
6:       Generate a neighbor s_n of s_c
7:       Compute Δ = f(s_n) − f(s_c)
8:       if CritMetropolis(Δ, T) then
9:          s_c := s_n
10:         nb_moves := nb_moves + 1
11:      end if
12:   end for
13:   acceptance_rate := i/nb_moves
14:   T := T ∗ coeff
15: until stopping condition
16: return best solution
```

chance de s'extraire d'un minimum local. Le principe du recuit simulé est décrit dans l'algorithme 2.

La méthode du recuit simulé a des avantages, car des solutions sont obtenues pour un nombre important de problèmes, le plus souvent de grande taille. Elles sont généralement de bonne qualité. En outre, c'est une méthode générale, applicable et facile à implémenter. Elle offre également une grande souplesse d'emploi vis-à-vis des évolutions du problème (incorporation de nouvelles contraintes). Cependant, les inconvénients du recuit simulé résident, d'une part dans les réglages du nombre importants de paramètres (température initiale, taux de décroissance de la température, durée des paliers de température, critère d'arrêt du programme, *etc.*) ; de bons réglages relèvent, souvent, du savoir-faire de l'utilisateur. D'autre part, les temps de calculs peuvent devenir très importants dans certaines applications. Des concepts additionnels relatifs au sujet du recuit simulé et à quelques unes de ses applications peuvent être trouvées dans [Al-khedhairi, 2008; Li et Silva, 2008; Sydow *et al.*, 2009; Matsui et Yamada, 2008; Lazarova, 2008].

3.4.2.2 Cas de la réduction de la consommation d'énergie

Nous avons proposé, dans [Idrissi Aouad *et al.*, 2010c], une heuristique basée sur les principes de la méthode du recuit simulé. Cet algorithme séquentiel de type recuit simulé développé, noté *SA_Seq*, est décrit dans l'algorithme 3 où les caractéristiques suivantes sont respectées :

- **Solution initiale** : elle est choisie d'une manière aléatoire. Une solution est représentée par un tableau dont la taille est égale au nombre total de données. Chaque élément de ce tableau indique si la donnée est incluse dans la SPM ('1') ou pas ('0').
- **Variable** $best\_solution$: contient la meilleure solution trouvée.
- $iter$: compte le nombre d'itérations de la boucle principale.
- **Fonction *Evaluate*** : la fonction objectif que l'on souhaite minimiser. Appelée également fonction *fitness*, elle sert à tester chacune des solutions trouvées pour voir si elles satisfont les conditions sous considération.

Algorithm 3 Algorithme séquentiel *SA_Seq*.

1: maximum iterations $maxiter \leftarrow 10000$
2: Initialize T
3: $stop\_criterion \leftarrow maxiter$, $iter \leftarrow 0$
4: Initialize $current\_solution$
5: $best\_solution \leftarrow current\_solution$
6: $current\_cost \leftarrow$ Evaluate ($current\_solution$)
7: $best\_cost \leftarrow current\_cost$
8: **while** Not stop_criterion **do**
9: **while** inner-loop stop criterion **do**
10: $Neighbor \leftarrow$ Generate ($current\_solution$)
11: $Neighbor\_cost \leftarrow$ Evaluate ($Neighbor$)
12: **if** Accept (current_cost, Neighbor_cost, T) **then**
13: $current\_solution \leftarrow Neighbor$
14: $current\_cost \leftarrow Neighbor\_cost$
15: **end if**
16: $iter \leftarrow iter + 1$
17: **if** $current\_solution$ is better than $best\_solution$ **then**
18: $best\_solution \leftarrow current\_solution$
19: **end if**
20: **end while**
21: Update (T) according to $T = 0.99 \times T$
22: Update ($stop\_criterion$)
23: **end while**

$$Fitness(solution) = Total\_Number\_Access\_all\_data - Number\_Access(solution). \tag{3.3}$$

– La boucle principale se déroule jusqu'à ce que le nombre d'itérations dépasse $Max\_iteration$ ou jusqu'à ce que la solution optimale soit atteinte.
– **Fonction *Generate*** : *SA_Seq* est un algorithme de recherche aléatoire de l'espace des solutions faisables utilisant la notion de relation de voisinage. Soit S l'ensemble des solutions faisables et $f : S \rightarrow \Re$ la fonction objectif à minimiser. Une relation de voisinage est une relation binaire $N \subseteq S \times S$ avec quelques propriétés spécifiques. L'interprétation de $N(s, s')$ est que la solution s est voisine de la solution s' dans l'espace de recherche de toutes les solutions S. Une heuristique de voisinage commence sa recherche par une solution initiale s_0 et à chaque étape passe d'une solution courante à une solution voisine selon certaines règles spécifiques à l'heuristique. À chaque itération, notre algorithme *SA_Seq* génère un voisin aléatoire de la solution courante $current\_solution$ (ligne 10). Nous définissons la relation de voisinage comme suit :

  – avec une probabilité égale à 0.03, la valeur de chaque élément de $current\_solution$ passe de 1 à 0 ou de 0 à 1.
  – tant que *pas faisable*[25]($current\_solution$) retirer la donnée j ayant un faible nombre d'accès de $current\_solution$ ($current\_solution[j] \leftarrow 0$).

25. $current\_solution$ devant satisfaire la contrainte de ne pas dépasser la capacité maximale de la SPM.

– **Fonction *Accept*** : cette fonction est l'idée principale dans l'approche *SA_Seq*. Elle spécifie la probabilité d'accepter un mouvement de $current\_solution$ à une solution voisine $Neighbor$, chose qui dépend également de la température (T). La fonction *Accept* doit satisfaire les conditions suivantes :

1. $p = 1$ si la solution $Neighbor$ est meilleure que $current\_solution$ en termes de fonction coût f ($i.e.$ $f(Neighbor) < f(current\_solution)$ dans un problème de minimisation).
2. si $Neighbor$ est pire que $current\_solution$, la valeur de p est positive ($i.e.$ elle permet le déplacement vers une solution pire) mais décroît avec $|f(Neighbor) - f(current\_solution)|$.
3. pour $current\_solution$ et $Neighbor$ fixes, lorsque $Neighbor$ est pire que $current\_solution$, la valeur de p décroît avec le temps et tend vers 0.

La fonction $Accept(c_{cost}, n_{cost}, T)$ est décidée par la probabilité d'accepter la configuration de la solution $Neighbor$. Cette probabilité est donnée par la formule suivante :

$$p = \begin{cases} 1 & \text{si} \quad n_{cost} < c_{cost} \\ \frac{1}{2} rand * (1 + e^{\frac{c_{cost} - n_{cost}}{T}}) & \text{sinon} \end{cases} \tag{3.4}$$

où T est la température et $rand$ un nombre aléatoire généré indépendamment dans l'intervalle $[0, 1]$.

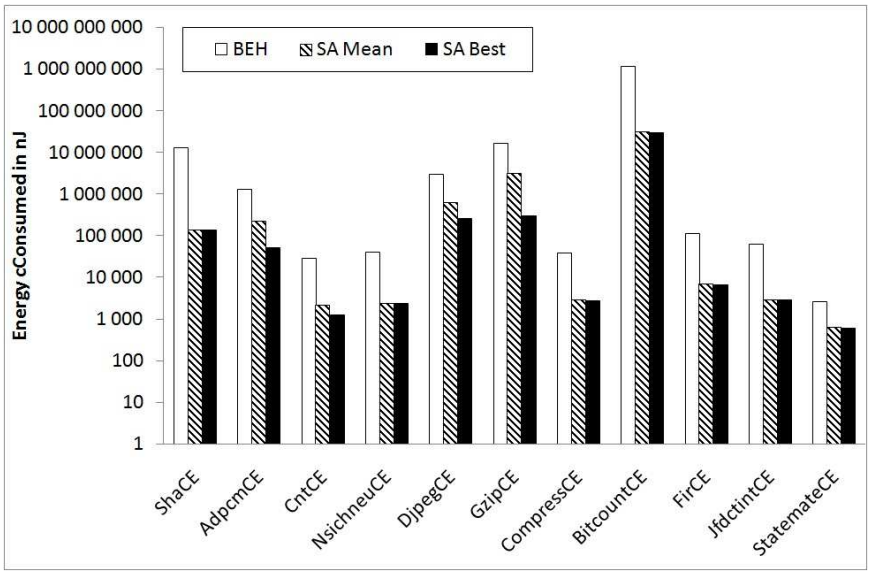

FIGURE 3.4 – Énergies consommées par notre algorithme de type recuit simulé.

La figure 3.4, présente les résultats obtenus sur les programmes tests en comparant notre heuristique *SA_Seq* à l'heuristique de base : BEH. *SA Best* fait référence à la meilleure solution trouvée par *SA_Seq* parmi les 30 exécutions alors que *SA Mean* réfère à la solution moyenne obtenue sur les 30 exécutions de *SA_Seq*. Comme on peut le voir sur cette figure, les deux heuristiques basées sur le principe de la méthode du recuit simulé donnent de meilleurs résultats énergétiques que la traditionnelle méthode de tri BEH. En effet, ces résultats montrent que ces deux heuristiques consomment de 76.23% (StatemateCE)

jusqu'à 98.92% (ShaCE) moins d'énergie que la méthode BEH. Pour certains programmes tests, *SA Best* donne de meilleurs résultats que *SA Mean*. En l'occurrence, *SA Best* réduit l'énergie de 3.18% pour CntCE, de 12.18% pour DjpegCE, de 12.82% pour AdpcmCE et de 16.36% pour GzipCE par rapport à *SA Mean*. Grâce donc à cette nouvelle heuristique, nous avons également dépassé les performances énergétiques atteintes avec la méthode de type de recherche avec tabous présentée en section 3.4.1.

3.5 Les méthodes évolutives

Les méthodes basées sur des populations de solutions sont parfois appelées méthodes évolutives, parce qu'elles font évoluer une population d'individus selon des règles bien précises. Ces méthodes alternent des périodes d'adaptation individuelle et des périodes de coopération durant lesquelles les individus peuvent échanger l'information. Parmi les méthodes évolutives, nous nous intéressons aux algorithmes génétiques et à l'optimisation par essaims particulaires.

3.5.1 Algorithmes génétiques

Les algorithmes génétiques (*GAs*[26]) [Sivanandam et Deepa, 2007] sont inspirés de la théorie de l'évolution proposée par Charles Darwin et des processus biologiques qui permettent à des organismes de s'adapter à leur environnement :

- Dans chaque environnement, seules les espèces les mieux adaptées perdurent au cours du temps ; les autres étant condamnées à disparaître.
- Au sein de chaque espèce, le renouvellement des populations est essentiellement dû aux meilleurs individus de l'espèce.

On parlera ainsi d'individu dans une population et bien souvent l'individu sera résumé par un seul chromosome. Les chromosomes sont eux-mêmes constitués de gènes qui contiennent les caractères héréditaires de l'individu. On retrouvera aussi les principes fondamentaux de l'évolution naturelle, à savoir les principes de sélection, de croisement, de mutation, *etc*.

Dans le cadre de l'optimisation, chaque individu représente un point de l'espace d'état auquel on associe la valeur du critère à optimiser. On génère ensuite, aléatoirement, une population d'individus pour laquelle l'algorithme génétique s'attache à sélectionner les meilleurs individus tout en assurant une exploration efficace de l'espace d'état. Les algorithmes génétiques diffèrent des algorithmes classiques d'optimisation et de recherche essentiellement en quatre points fondamentaux :

- Ils utilisent un codage des éléments de l'espace de recherche et non pas les éléments eux-mêmes.
- Ils recherchent une solution à partir d'une population de points et non pas à partir d'un seul point.
- Ils n'imposent aucune régularité sur la fonction étudiée (continuité, dérivabilité, convexité, ...). C'est un des gros atouts des algorithmes génétiques.
- Ils ne sont pas déterministes ; ils utilisent des règles de transition probabilistes.

L'algorithme génétique commence par générer une population d'individus, de façon aléatoire. Puis, on sélectionne un certain nombre d'individus dans la population afin de générer une population intermédiaire, appelée aussi *mating pool*. Deux parents sont ensuite sélectionnés (P_1 et P_2) en fonction de leurs adaptations. On applique, aléatoirement, un opérateur de croisement avec une probabilité P_c qui génère

26. GA : *Genetic Algorithms.*

Algorithm 4 Fonctionnement d'un algorithme génétique de base.

1: Select random population of n chromosomes.
2: Evaluate the fitness $f(x)$ of each chromosome x in the population.
3: **repeat**
4: Select two parent chromosomes P_1 and P_2 from a population.
5: Cross over the two parents P_1 and P_2 to form new children C_1 and C_2 with a crossover probability P_c.
6: Mutate new children C_1 and C_2 with a mutation probability P_m.
7: Place new offspring C_1' and C_2' in the new population.
8: Use new generated population for a further sum of the algorithm.
9: **until** the end condition is satisfied
10: **return** best solution.

deux enfants C_1 et C_2. Si la probabilité de croisement P_c vaut 0.6, par exemple, on appliquera l'opérateur de croisement dans 60% des cas sur les individus P_1 et P_2. On modifie ensuite certains gènes de C_1 et C_2 en appliquant un opérateur de mutation avec une probabilité P_m, ce qui produit deux nouveaux individus C_1' et C_2' pour lesquels on évalue le niveau d'adaptation avant de les insérer dans la nouvelle population. Contrairement à la reproduction et au croisement qui favorisent l'intensification, l'opérateur de mutation favorise la diversification des individus. On réitère les opérations de sélection, de croisement et de mutation afin de compléter la nouvelle population. Cela termine le processus d'élaboration d'une génération. Enfin, on réitère les opérations précédentes à partir de la sélection jusqu'à ce qu'un critère d'arrêt soit satisfait. Différents critères d'arrêt de l'algorithme peuvent être choisis : nombre de générations fixé, limite de convergence de la population, population qui n'évolue plus suffisamment, *etc*. L'algorithme 4 résume le fonctionnement d'un algorithme génétique de base.

Il n'y a pas une seule manière de définir les mécanismes des algorithmes génétiques. Le codage et l'évaluation des individus, la sélection, le croisement et la mutation peuvent différer d'un problème à un autre (et chaque utilisateur aura même ses préférences). Pour une liste complète des applications des algorithmes génétiques, le lecteur intéressé peut voir [Zheng et Kiyooka, 1999].

3.5.1.1 Terminologies de base

Pour utiliser un algorithme génétique sur un problème particulier, nous donnons dans ce chapitre, brièvement, quelques terminologies de base et décrivons, plus en détail dans les sous-sections qui suivent, les différents opérateurs génétiques.

Individu : un individu est une solution unique au problème traité.

Population : une population est un ensemble d'individus. Les deux aspects importants d'une population utilisée dans les algorithmes génétiques sont : la génération de la population initiale et la taille de la population. Pour chaque problème, la taille de la population dépendra de la complexité du problème. Idéalement, la population initiale devrait être aussi diversifiée que possible afin de pouvoir explorer correctement l'ensemble de l'espace de recherche. Pour ce faire, dans la majorité des cas, la population initiale est choisie de manière aléatoire.

Fonction de performance : la fonction de performance (ou *fonction fitness* ou *fonction coût*) associe une valeur de performance à chaque individu afin de déterminer le nombre de fois qu'il sera sélectionné

pour être reproduit ou bien, s'il sera remplacé. La fonction de performance peut également être la fonction objectif du problème. Elle n'indique pas seulement la qualité de la solution, mais également l'écart entre l'individu et la solution optimale. La qualité de la fonction de performance conditionne, pour une grande part, l'efficacité d'un algorithme génétique.

Critère d'arrêt : le critère d'arrêt conditionne l'arrêt de l'algorithme. Il existe différentes conditions d'arrêt : le nombre maximum de générations est atteint ; une durée spécifique s'est écoulée, pas de changements sont notés sur la fonction de performance des individus de la population pour un nombre spécifié de générations ou durant un intervalle de temps, *etc*.

3.5.1.2 Opérateur de croisement

Le croisement a pour but d'enrichir la diversité de la population en manipulant la structure des chromosomes. Classiquement, le croisement est envisagé avec deux parents P_1 et P_2 et génère deux enfants C_1 et C_2. Il consiste à échanger les gènes des parents afin de donner des enfants qui portent des propriétés combinées. Bien qu'il soit aléatoire, cet échange d'informations offre aux algorithmes génétiques une part de leur puissance : parfois, de "bons" gènes d'un parent viennent remplacer les "mauvais" gènes d'un autre parent et créent des fils mieux adaptés aux parents. Il existe deux techniques classiques de croisement : le croisement en "un point" et le croisement en "deux points". L'opérateur de croisement est traditionnellement l'heuristique prépondérante d'un algorithme génétique. La figure 3.5 représente le principe de fonctionnement du croisement en un point. On tire, aléatoirement, une position dans chacun des parents P_1 et P_2. Puis ; on échange les deux sous-chaînes de chacun des chromosomes, ce qui produit deux enfants C_1 et C_2. Ce mécanisme présente l'inconvénient de privilégier les extrémités des individus. Et selon le codage choisi, il peut générer des fils plus ou moins proches de leurs parents.

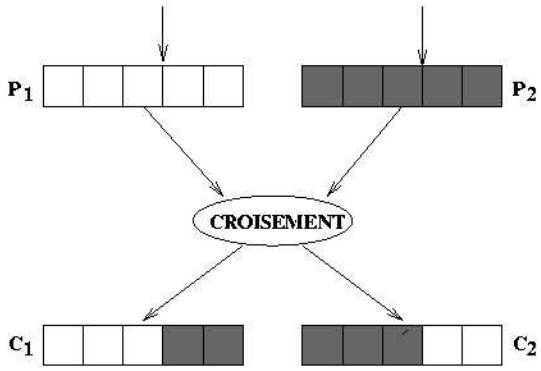

FIGURE 3.5 – Principe du croisement en un point.

Pour éviter ce problème, on peut étendre ce principe en découpant le chromosome en plusieurs sous-chaînes. La figure 3.6 représente le principe de fonctionnement du croisement en deux points. Ici, on tire aléatoirement deux positions dans chacun des parents P_1 et P_2. Puis, on échange les deux sous-chaînes du milieu de chacun des chromosomes, ce qui produit deux enfants C_1 et C_2.

Une probabilité de croisement P_c est le paramètre déterminant la proportion des individus qui sont croisés parmi ceux qui remplaceront l'ancienne génération. S'il n'y a pas de croisement, les descendants sont les copies exactes de leurs parents. Si la probabilité de croisement P_c vaut 1, cela veut dire que tout

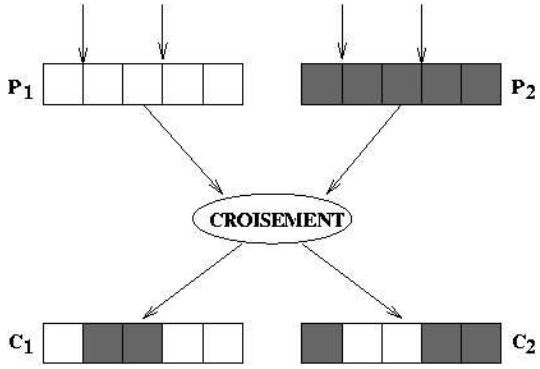

FIGURE 3.6 – Principe du croisement en deux points.

descendant est obtenu par croisement. Si P_c vaut 0, toute la nouvelle génération est fabriquée à partir des copies exactes des chromosomes de l'ancienne génération. La probabilité de croisement est un paramètre de l'algorithme génétique et dépend du problème et de la technique de recombinaison adoptée. Elle est alors comprise strictement entre 0 et 1.

3.5.1.3 Opérateur de mutation

La mutation prévient l'algorithme d'être piégé dans un optimum local. La mutation est vue comme un opérateur chargé de maintenir la diversité génétique de la population. Elle sert à éviter une convergence prématurée de l'algorithme. La figure 3.7 représente le principe de fonctionnement de l'opérateur de mutation. L'opérateur de mutation consiste à tirer, aléatoirement, un gène dans le chromosome et à remplacer ce dernier par une valeur tirée, aléatoirement aussi, de l'alphabet propre au gène sélectionné.

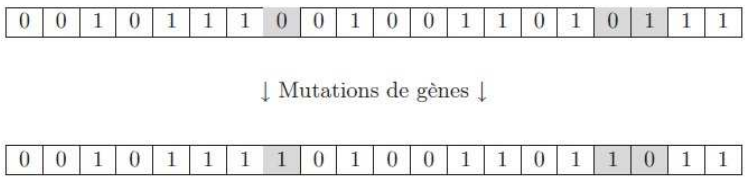

FIGURE 3.7 – Principe de la mutation.

De la même manière que pour les croisements, on définit ici une probabilité de mutation P_m lors des changements de population. La probabilité de mutation P_m définit la proportion des individus mutés dans la population. S'il n'y a pas de mutation, les descendants sont générés, immédiatement, après le croisement sans aucun changement. Si P_m vaut 1, tous les gênes du chromosome changent et si P_m vaut 0, rien ne change. Il faut, cependant, être prudent et ne pas laisser la mutation avoir lieu très souvent. En effet, il est nécessaire de choisir, pour ce taux, une valeur relativement faible de manière à ne pas tomber dans une recherche aléatoire et conserver le principe de sélection et d'évolution.

3.5.2 Optimisation par essaims particulaires

L'optimisation par essaims particulaires (*PSO*[27]) est une technique relativement récente développée par [Kennedy et Eberhart, 1995] et inspirée à l'origine du monde du vivant. Elle s'appuie, notamment, sur un modèle permettant de simuler le déplacement d'un groupe d'oiseaux, d'un banc de poissons ou d'un essaim d'abeilles. Il s'agit d'une méthode faisant appel à une population d'agents, appelés ici *particules*. Cette méthode est fondée sur la collaboration des individus entre eux. Ici, l'efficacité est due à la collaboration plutôt qu'à la compétition. En effet, cette idée veut qu'un groupe d'individus, peu intelligents, possède une organisation globale complexe. Ainsi, grâce à des règles de déplacement très simples (dans l'espace des solutions), les particules peuvent converger progressivement vers un minimum global. Des règles très simples comme "rester relativement proche des autres individus", "aller dans la même direction", "à la même vitesse", suffisent à maintenir la cohésion du groupe tout entier et à permettre des comportements collectifs complexes et adaptés.

La méthode met en jeu de larges groupes de particules sous forme de vecteurs se déplaçant sur l'espace de recherche. Chaque particule i est caractérisée par sa position x_i et par un vecteur de changement de position appelé *vélocité* v_i. La i_{eme} particule est représentée comme $X_i = (x_{i1}, x_{i2}, ..., x_{in})$. Le cœur de la méthode consiste à choisir comment définir v_i. D'après la socio-psychologie, chaque individu est influencé par son comportement passé et par celui de ses voisins pour décider de son propre déplacement. Autrement dit, à chaque itération, chaque particule se déplace en fonction des deux meilleures valeurs suivantes :

- La meilleure position passée de la particule, appelée $pbest$.
- La meilleure position obtenue parmi n'importe quel voisin (voisin dans le réseau social et non, nécessairement, dans l'espace), appelée $gbest$. Lorsqu'une particule considère une partie de la population comme ses voisins topologiques (voisins dans l'espace de recherche), la meilleure valeur est une meilleure position locale, appelée dans ce cas $lbest$.

À chaque itération, ces deux meilleures valeurs sont combinées afin d'ajuster la vélocité et calculer ainsi un nouveau mouvement pour la particule. $pbest$ est considéré comme étant le composant cognitif (*cognition component*) relatif à l'*expérience individuelle* et $gbest$ (ou $lbest$) est considéré comme étant le composant social (*social component*) relatif à la *communication sociale*. Notons qu'il semble être communément admis qu'un voisinage social $gbest$ (un individu x_2 ayant, par exemple, pour voisins les individus x_1 et x_3, quelles que soient les localisations spatiales de x_1, x_2, x_3) donne de meilleurs résultats qu'un voisinage spatial $lbest$ (fonction de la proximité des individus dans l'espace de recherche). La figure 3.8 schématise le déplacement d'une particule selon ces informations.

Les équations de changement de position sont donc les suivantes :

$$\begin{aligned} v_{i+1} = \omega v_i + c_1 * random(0,1) * (pbest_i - x_i) \\ + c_2 * random(0,1) * (gbest_i - x_i) \end{aligned} \tag{3.5}$$

$$x_{i+1} = x_i + v_{i+1} \tag{3.6}$$

où $random(0,1)$ est un nombre aléatoire tiré dans l'intervalle $[0,1]$ et c_1 et c_2 sont deux paramètres qui ont pour rôle de pondérer les rôles relatifs de l'*expérience individuelle* (c_1) et de la *communication sociale* (c_2) (habituellement, on prend $c_1 = c_2 = 2$, voir [Shi et Eberhart, 1999]). Le tirage aléatoire uniforme est justifié si l'on ne considère aucun a priori sur l'importance de l'une ou l'autre source

27. PSO : *Particle Swarm Optimization.*

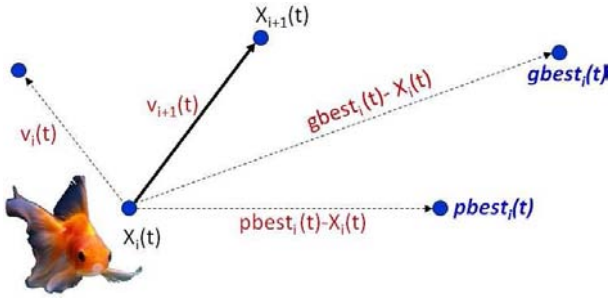

FIGURE 3.8 – Mouvement de chaque particule.

d'informations. Notons également qu'il est possible de considérer, à la place de ce tirage aléatoire et uniforme, un tirage basé sur la distribution de Lévy ou encore sur la distribution de Cauchy.

Pour éviter que le système n'explose en cas d'amplification d'oscillations trop grande, un paramètre V_{max} permet de limiter la vélocité (sur chaque dimension). Dans la version de base, c'est ce paramètre V_{max} qui va empêcher le système d'exploser par amplification des rétroactions positives. Dans le cas de l'équation 3.5, si la somme à droite de l'égalité dépasse une certaine valeur constante, alors la vélocité sur cette dimension est assignée à être V_{imin} ou V_{imax}. Ainsi, les vélocités des particules sont limitées à l'intervalle $[V_{imin}, V_{imax}]$ qui sert comme contrainte de contrôle de la capacité d'exploration globale (la divergence) de l'algorithme. Cela réduit également la probabilité que les particules quittent l'espace de recherche. Notons que cela ne restreint pas les valeurs de x_i à l'intervalle $[V_{imin}, V_{imax}]$; mais limite seulement la distance maximale qu'une particule va parcourir au cours d'une itération. Cette méthode permet de provoquer une convergence de l'algorithme (l'amplitude des mouvements des individus diminue jusqu'à son annulation). On réalise ainsi un compromis efficace entre intensification et diversification.

Dans la même idée, une version met en place un poids d'inertie (*inertia weight*), noté ω dans l'équation de vélocité (voir équation 3.5) [Shi et Eberhart, 1998]. Pour résumer, le poids d'inertie décroît en fonction du temps, ce qui provoque une convergence contrôlable par ce paramètre permettant ainsi de contrôler la dynamique intensification/diversification du système. La dynamique générale reste la même que dans la version précédente, à l'exception près de l'impossibilité de repartir dans une dynamique de diversification si un nouveau point, meilleur que le précédent, est trouvé.

Le lecteur intéressé trouvera un état de l'art complet entièrement dédié à l'optimisation par essaims particulaires et les concepts qui lui sont associés en français dans [Clerc, 2005].

3.5.3 Cas de la réduction de la consommation d'énergie

Nous nous sommes alors intéressés aux méthodes évolutives et avons proposé dans [Idrissi Aouad *et al.*, 2010e] une heuristique basée sur les principes des algorithmes génétiques présentés en section 3.5.1. Notre heuristique de type algorithme génétique fonctionne en utilisant les opérateurs suivants :

- **Individu** : un individu est une solution unique.
- **Population** : une population est un ensemble d'individus choisis aléatoirement.
- **Encodage** : soit N le nombre total de données. Une solution est une séquence s finie de N termes telle que $s[n]$ est soit 0, soit la taille de la n_{ieme} donnée. $s[n] = 0$ si et seulement si la n_{ieme}

donnée n'est pas sélectionnée dans la solution. Cette solution doit respecter la contrainte de ne pas dépasser la capacité maximale de la SPM ($i.e.$ $\sum_{i=1}^{N} s[i] \leq C$).

- **Fonction de performance** : à chaque étape (*génération*), tout individu de l'espace des solutions est évalué selon la fonction de performance (selon la capacité de la SPM qu'il remplit dans notre cas) et les meilleurs et les pires individus sont identifiés.
- **Croisement** : lorsque les conditions sont satisfaites pour procéder au croisement (fonction de la probabilité de croisement), la meilleure solution est croisée avec une solution aléatoire non-extrême et le résultat enrichit la population. L'enfant hérite d'une partie des gènes de chaque parent. Nous avons utilisé les techniques de croisement et la probabilité suivantes :

  - **Croisement en un point** : les deux chromosomes à croiser sont coupés une fois à un endroit aléatoire et les sections après l'endroit de la coupe sont permutés (voir figure 3.5 page 60).
  - **Croisement en deux points** : deux endroits de coupe sont choisis aléatoirement dans chaque chromosome parent et le contenu entre ces deux endroits de coupe est permuté (voir figure 3.6 page 61).
  - **Probabilité de croisement (P_c)** : elle est comprise entre 0 et 1. C'est l'un des paramètres de réglage de l'algorithme génétique qui se fait par plusieurs expérimentations. Plusieurs valeurs ont été testées ; seules les valeurs pour lesquelles nous obtenons les meilleurs résultats sont présentées.

- **Mutation** : lorsque les conditions sont satisfaites pour procéder à la mutation (fonction de la probabilité de mutation), la solution est mutée. Rappelons qu'une solution est une séquence de termes. Ainsi, la mutation d'une solution se déroule de la façon suivante. Si un terme de la séquence n'est pas actuellement à 0, on le passe à 0. Si ce terme est actuellement à 0, on le fait passer à la taille de la donnée correspondante.

  - **Probabilité de mutation (P_m)** : elle est comprise entre 0 et 1. C'est un des paramètres de réglage de l'algorithme génétique qui se fait par plusieurs expérimentations. Plusieurs valeurs ont été testées ; seules les valeurs pour lesquelles nous obtenons les meilleurs résultats sont présentées.

- **Critère d'arrêt** : nombre maximum de générations atteint.

3.5.3.1 Algorithme génétique avec croisement et mutation

Dans cette sous-section, nous testons les performances de notre algorithme génétique en utilisant les deux opérateurs génétiques : le croisement et la mutation. *Genetic 1* représente les résultats obtenus avec $P_m = 0.1$, $P_c = 0.5$ et un croisement en un point. *Genetic 2* représente les résultats obtenus avec $P_m = 0.1$, $P_c = 0.5$ et un croisement en deux points. 30 exécutions de chaque heuristique ont été générées. Pour chaque heuristique, le résultat moyen est égal au meilleur résultat ; cela veut dire que l'algorithme trouve à chaque fois le résultat optimal. De plus, les résultats obtenus avec *Genetic 1* et *Genetic 2* sont égaux ; cela veut dire que, dans notre cas, effectuer un croisement en un point ou un croisement en deux points n'influe pas sur la qualité de la solution obtenue. Ainsi, dans le but d'alléger le graphique, nous utilisons la notation *Genetic (1,2)* qui représente les résultats moyens obtenus indifféremment avec *Genetic 1* ou avec *Genetic 2*. La figure 3.9 présente les résultats énergétiques obtenus en comparant l'heuristique BEH à nos algorithmes génétiques.

Comme on peut le constater d'après cette figure, *Genetic (1,2)* atteint de meilleures performances

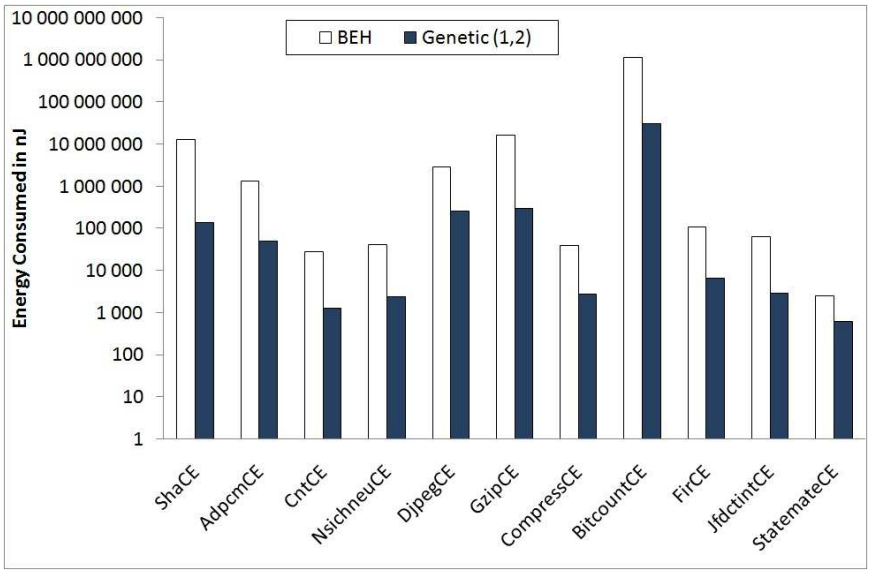

FIGURE 3.9 – Énergies consommées par nos algorithmes de type génétique.

énergétiques que BEH. En effet, ces résultats montrent que *Genetic (1,2)* consomme de 76.23% (StatemateCE) jusqu'à 98.92% (ShaCE) moins d'énergie que BEH.

3.5.3.2 Algorithme génétique avec croisement ou mutation

Dans cette sous-section, nous voulons connaître l'impact de la suppression de l'un des deux opérateurs génétiques sur la consommation d'énergie. D'abord, nous commençons par omettre l'opération de croisement. *Genetic 3* représente les résultats obtenus avec $P_m = 0.5$. La figure 3.10 donne les résultats obtenus en comparant BEH à notre algorithme génétique sans opérateur de croisement.

D'après cette figure, *Genetic 3* fait mieux que BEH et atteint les mêmes performances énergétiques en utilisant les deux opérateurs génétiques. En effet, ces résultats montrent que *Genetic 3* consomme de 76.23% (StatemateCE) jusqu'à 98.92% (ShaCE) moins d'énergie que BEH et cela sans effectuer aucun croisement.

Ensuite, nous poursuivons en omettant l'opération de mutation. *Genetic 4* représente les résultats obtenus avec $P_c = 0.5$ et un croisement en un point. *Genetic 5* représente les résultats obtenus avec $P_c = 0.5$ et un croisement en deux points. La figure 3.11 donne les résultats moyens obtenus en comparant BEH à nos algorithmes génétiques sans mutation.

D'après la figure 3.11, à la fois *Genetic 4* et *Genetic 5* produisent de meilleurs résultats en termes d'énergie que BEH. En effet, ces résultats montrent que, à la fois *Genetic 4* et *Genetic 5* consomment de 76.23% (StatemateCE) jusqu'à 98.92% (ShaCE) moins d'énergie que BEH et cela sans effectuer aucune mutation.

Ces expérimentations prouvent que, dans certains cas, il n'est pas nécessaire d'appliquer les deux opérateurs génétiques à la fois pour obtenir de bons résultats. Ainsi, pour certains problèmes, omettre l'un des deux opérateurs génétiques (croisement ou mutation) permet, quand même, à l'algorithme génétique de converger vers la solution optimale du problème, à condition toutefois de régler correctement chacun des différents paramètres.

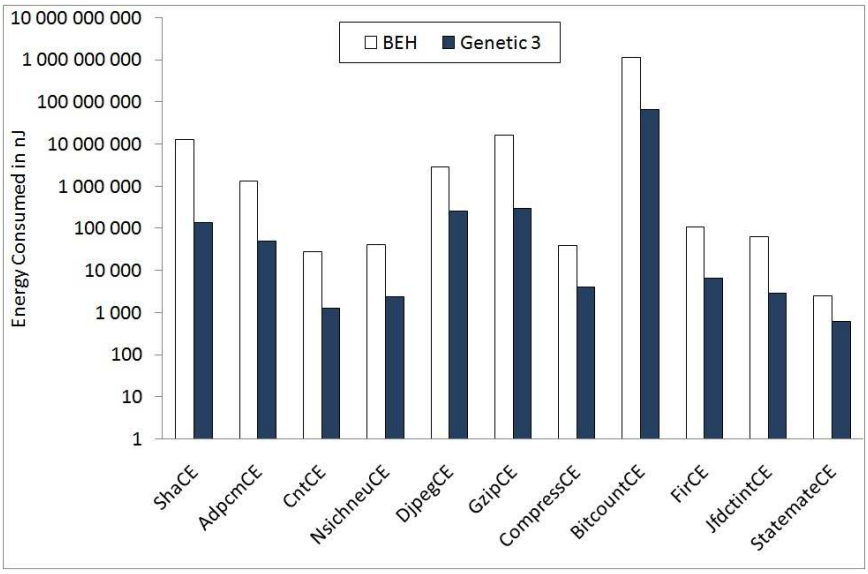

FIGURE 3.10 – Énergies consommées par notre algorithme de type génétique sans croisement.

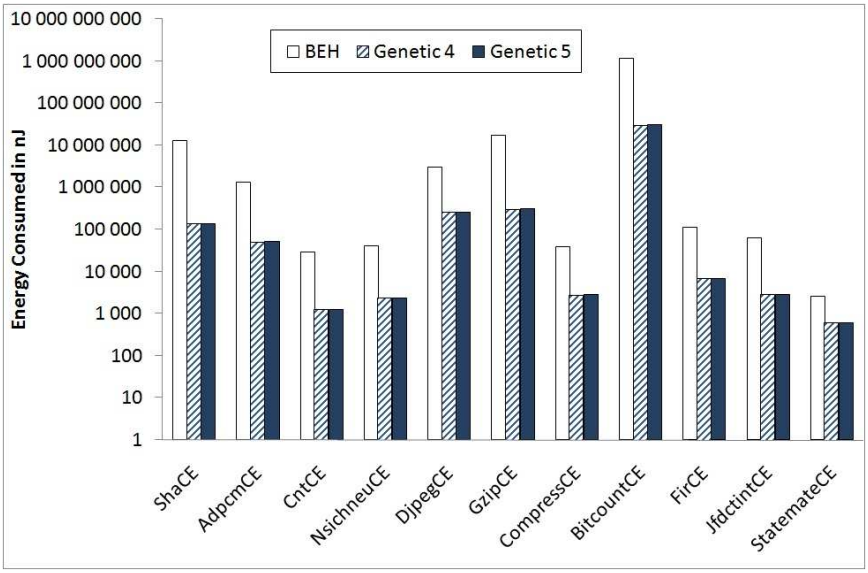

FIGURE 3.11 – Énergies consommées par nos algorithmes de type génétique sans mutation.

3.6 Comparaison entre métaheuristiques

Maintenant que nous avons expérimenté différentes métaheuristiques, il convient de les tester entre elles et cela afin de mieux se rendre compte de l'efficacité de chacune par rapport à l'autre en termes de consommation d'énergie en mémoire. Ainsi, nous avons comparé les résultats moyens obtenus sur 30 exécutions de chacune des métaheuristiques de type recherche avec tabous, recuit simulé et algorithmes génétiques avec BEH. La figure 3.12 récapitule les résultats obtenus.

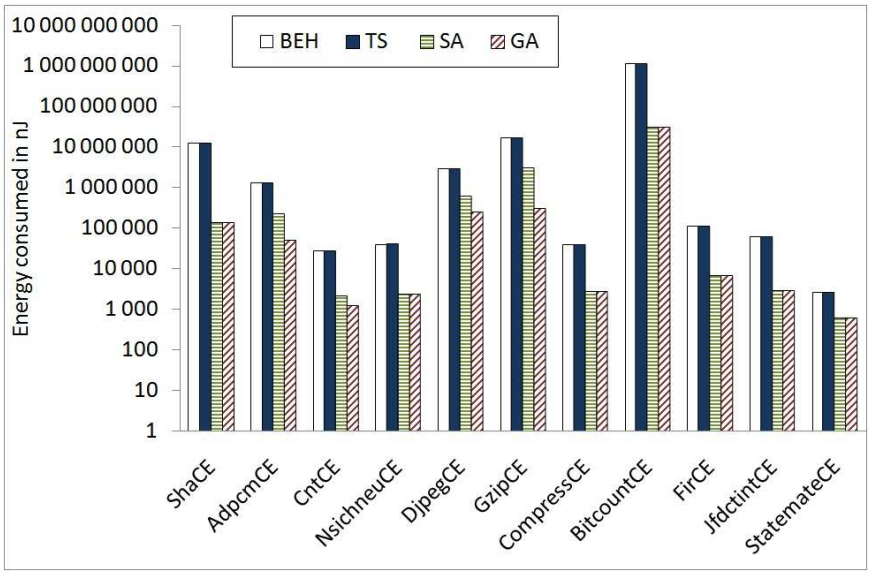

FIGURE 3.12 – Comparaison des énergies consommées par nos algorithmes de type métaheuristiques classiques.

Nous remarquons d'abord que TS atteint les mêmes performances énergétiques que BEH. Ensuite, SA arrive à faire mieux que BEH et TS pour tous les programmes tests. Enfin, GA est l'heuristique qui atteint les meilleurs gains énergétiques. En l'occurrence, ces résultats montrent que, pour certains programmes tests, GA réduit l'énergie de 3.18% pour CntCE, de 12.18% pour DjpegCE, de 12.82% pour AdpcmCE et de 16.36% pour GzipCE par rapport à SA. Grâce à cette figure, l'efficacité (en termes de consommation d'énergie) des méthodes évolutives et plus particulièrement des algorithmes génétiques par rapport aux méthodes de recherche locale devient ainsi évidente.

3.7 Conclusion

Dans ce chapitre, nous avons pu démontrer en section 3.6, que grâce aux algorithmes génétiques nous avons pu atteindre les meilleurs gains en consommation d'énergie mémoire obtenus en comparaison avec ceux qui sont obtenus en utilisant la méthode de la recherche avec tabous ou la méthode du recuit simulé. Effectivement, pour chacune de nos heuristiques basées sur le principe des algorithmes génétiques, le résultat moyen est égal au meilleur résultat ; cela veut dire que l'algorithme trouve à chaque fois le résultat optimal. Chose qui n'était pas vraie auparavant ni avec l'algorithme *SA_Seq*, ni avec l'algorithme de type *TS*, d'où l'intérêt des algorithmes génétiques et des méthodes évolutives en général.

Nous sommes d'une part, soucieux de savoir s'il est possible de réduire davantage la consommation d'énergie en mémoire dans les systèmes embarqués et de proposer des algorithmes plus robustes, qui peuvent être appliqués à d'autres problèmes encore plus complexes comme, en l'occurrence, celui de l'optimisation de fonctions tests multimodales. D'autre part, nous sommes curieux de tester l'efficacité des algorithmes hybrides, tant vantée par la littérature. C'est pourquoi, nous avons décidé de partir explorer cette voie. Ainsi, le chapitre suivant présentera de nouveaux algorithmes hybrides tentant de résoudre nos deux problèmes d'optimisation.

Chapitre 4

Algorithmes hybrides et distribués

Sommaire

4.1 Introduction

Nous avons pu constater en section 3.6 l'efficacité des algorithmes génétiques par rapport aux méthodes de recherche locale. Cependant, les algorithmes génétiques nécessitent un grand nombre d'évaluations de la fonction de performance (grand nombre d'itérations et une population importante) afin d'obtenir de bons résultats. Ainsi, pour des problèmes combinatoires complexes, les algorithmes génétiques sont désavantagés par la lourdeur des calculs. C'est pourquoi, l'idée est venue d'utiliser des algorithmes hybrides qui tirent profit de la complémentarité de plusieurs méthodes entre elles.

Dans ce chapitre, nous décrirons les différents algorithmes hybrides proposés pour résoudre les problèmes de l'optimisation de la consommation d'énergie et des fonctions multimodales et nous quantifierons leur impact. Ces problèmes ont été présentés au chapitre 1. Nous proposerons également des versions distribuées et coopératives de ces algorithmes hybrides. Précisons que pour chaque algorithme présenté, nous donnerons les résultats expérimentaux obtenus sur le premier problème. Concernant l'optimisation de fonctions tests multimodales, nous comparerons certains de nos algorithmes hybrides proposés avec ceux de travaux décrits dans la section 2.2 et cela afin de tester leur efficacité par rapport à ce qui a été déjà fait auparavant.

Les algorithmes hybrides associent souvent une métaheuristique évolutive et une méthode de recherche locale. De cette façon, la puissance de la méthode évolutive permet de balayer de manière globale d'espace des solutions. La méthode de recherche locale se concentre sur l'exploration d'une petite zone de cet espace afin d'affiner la solution. Cette coopération peut prendre la forme d'un simple passage de relais entre les méthodes utilisées. C'est une forme simple d'hybridation. On parle alors d'algorithmes de combinaisons comme ceux qui sont proposés en section 4.2 où nous proposons des heuristiques de combinaisons basées sur la recherche avec tabous et sur les algorithmes génétiques [Idrissi Aouad *et al.*, 2010f]. Mais les deux approches peuvent aussi être entremêlées de manière plus complexe. On parle alors de vrais algorithmes hybrides comme ceux qui sont décrits en section 4.3. Plus clairement, en hybridant deux algorithmes *A* et *B*, par exemple, nous obtenons un nouvel algorithme qui n'est plus complètement *A*, ni *B*, mais plutôt quelque chose qui est un mélange des deux algorithmes.

4.2 Algorithmes de combinaisons

4.2.1 TSGA

Pour cet algorithme de combinaison, que nous avons développé et noté *TSGA*, nous commençons d'abord par lancer l'algorithme de recherche avec tabous tel que c'est présenté en sous-section 3.4.1.2 sans changement des paramètres. Puis, nous appliquons l'algorithme génétique (décrit en sous-section 3.5.3) pour lequel la solution à améliorer devient la meilleure solution trouvée par TS. Ainsi, si l'algorithme génétique trouve une meilleure solution que celle qui est trouvée par la recherche avec tabous, il la retient. Sinon, il retient la solution trouvée par TS. Les résultats obtenus sont présentés dans la figure 4.1.

4.2.2 GATS

Pour cet algorithme de combinaison, que nous avons développé et noté *GATS*, nous commençons d'abord par lancer l'algorithme génétique tel qu'il est présenté en sous-section 3.5.3 sans changement des paramètres. Ensuite, nous appliquons l'algorithme de recherche avec tabous (décrit en sous-section 3.4.1.2) pour lequel la solution initiale à améliorer devient la meilleure solution trouvée par l'algorithme génétique et n'est donc plus choisie aléatoirement. Ainsi, si l'algorithme de recherche avec tabous trouve une meilleure solution que celle qui est trouvée par l'algorithme génétique, il la retient. Sinon, TS retient la solution trouvée par l'algorithme génétique. Les résultats obtenus sont présentés dans la figure 4.1.

4.3 Algorithmes hybrides

4.3.1 GA_Hybrid

Pour cet algorithme hybride, nous considérons l'approche génétique. Cette fois-ci, au lieu d'appliquer tous les opérateurs génétiques habituels (croisement et mutation), nous remplaçons l'opérateur de mutation par la méthode de recherche avec tabous. Cette heuristique est plus élaborée que les deux heuristiques précédentes dans la mesure où ce n'est pas une simple combinaison. En remplaçant la mutation par la recherche locale avec tabous, nous espérons améliorer la solution trouvée par l'algorithme génétique. En effet, TS est mieux armée pour pointer une solution intéressante que l'opérateur de mutation avec son mode de sélection aléatoire. Les principes de notre algorithme hybride, noté *GA_Hybrid*, sont décrits dans l'algorithme 5.

Algorithm 5 Algorithme *GA_Hybrid*.

Intitialize p_m, p_c, $p_i \in$]0,1] and $i \leftarrow 1$
$maxGen \leftarrow 10000$
Generate population P_0
Evaluate P_0 and find the best solution π^*
$\pi_{Elite} \leftarrow \pi^*$
$stop\_criterion \leftarrow$ false
while Not $stop\_criterion$ **do**
  $P_i \leftarrow \emptyset$
  for $j = 1$ to $PopSize/2$ **do**
    Select two parents p_1 and p_2 from P_{i-1}
    offspring $\leftarrow (p_1, p_2)$
    With probability p_c, perform offspring := crossover(p_1,p_2)
    With probability p_i, improve offspring by using TS (with $maxiter = 300$)
    Evaluate offspring and add it to P_i
  end for
  Add P_{i-1} to P_i
  Sort P_i
  Keep the $PopSize$ best solution in P_i
  Find the best solution π^* in P_i
  if π^* is better than π_{Elite} **then**
    $\pi_{Elite} \leftarrow \pi^*$
  end if
  if $fitness(\pi_{Elite}) = 0$ OR $i = maxGen$ **then**
    $stop\_criterion \leftarrow$ true
  end if
  Update(i)
end while

TSGA1, *GATS3* et *GA_Hybrid5* représentent les résultats obtenus pour $P_m = 0.1$, $P_c = 0.5$ et pour un croisement en un point. *TSGA2*, *GATS4* et *GA_Hybrid6* représentent les résultats obtenus pour $P_m = 0.1$, $P_c = 0.5$ et pour un croisement en deux points. 30 exécutions ont été générées pour chacun des algorithmes proposés. Pour chaque algorithme, le résultat moyen est égal au meilleur résultat ; cela veut dire que tous les algorithmes convergent vers le résultat optimal. De plus, les résultats obtenus avec

les couples (*TSGA1* et *TSGA2*), (*GATS3* et *GATS4*) et (*GA_Hybrid5* et *GA_Hybrid6*) sont égaux ; cela veut dire que quel que soit l'algorithme de combinaison ou hybride considéré, effectuer un croisement en un point ou un croisement en deux points n'influe pas sur la qualité de la solution obtenue. Ainsi, dans le but d'alléger le graphique, nous utilisons les notations *TSGA (1,2)*, *GATS (3,4)* et *GA_Hybrid (5,6)*. Étant donné que TS est une alternative à la méthode de tri BEH, comme c'est démontré en section 3.4.1, par la suite nous utiliserons les résultats obtenus par TS comme base de comparaison. La figure 4.1 donne les résultats moyens obtenus en comparant TS aux trois algorithmes hybrides *TSGA*, *GATS* et *GA_Hybrid*. Nous constatons que tous les algorithmes hybrides produisent de meilleurs résultats que TS. En effet, ces résultats montrent que *TSGA (1,2)*, *GATS (3,4)* et *GA_Hybrid (5,6)* consomment de 76.23% (StatemateCE) jusqu'à 98.92% (ShaCE) moins d'énergie que TS.

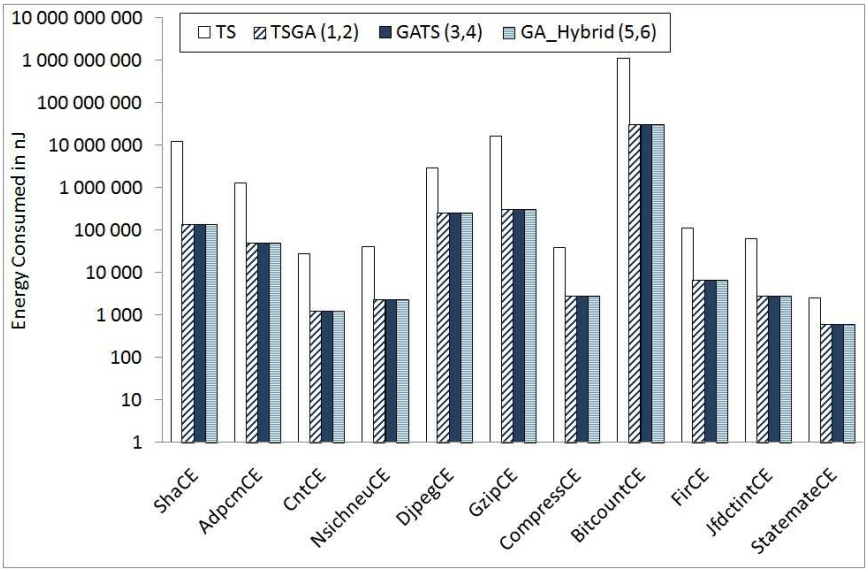

FIGURE 4.1 – Énergies consommées par *TSGA*, *GATS* et *GA_Hybrid*.

Étant donné que l'algorithme hybride *GA_Hybrid* et les deux algorithmes de combinaisons *TSGA* et *GATS* atteignent les mêmes gains énergétiques, nous avons calculé les temps d'exécution de chacun d'entre eux afin de les départager. Les temps d'exécution des algorithmes *TSGA* et *GATS* sont égaux, nous n'affichons donc que les résultats obtenus pour *TSGA*. Le tableau 4.1 donne le pourcentage en temps d'exécution nécessaire au déroulement de chaque algorithme.

TABLE 4.1 – Pourcentage des temps d'exécution des algorithmes *TSGA* et *GA_Hybrid*.

Programmes tests	**TSGA**	**GA_Hybrid**
ShaCE	100	**99.33**
AdpcmCE	**100**	119.08
CntCE	100	**98.78**
NsichneuCE	**100**	118.72
DjpegCE	100	**82.19**
GzipCE	100	**99.20**

Les valeurs écrites en caractères gras se réfèrent à l'algorithme nécessitant le moins de temps d'exécution. Par exemple, pour le programme test DjpegCE, l'algorithme *GA_Hybrid* est le plus rapide. En revanche, pour le programme test AdpcmCE, l'algorithme *TSGA* est cette fois le plus rapide. Il en résulte que l'algorithme *GA_Hybrid* est en général le plus rapide. Cela est dû au fait qu'il réduit le nombre de générations par rapport aux algorithmes de combinaisons, d'où l'intérêt d'utiliser des algorithmes hybrides.

4.3.2 GASA

Dans cette sous-section, nous présentons un algorithme hybride, proposé dans [Idrissi Aouad *et al.*, 2010b] et noté *GASA*, qui est basé sur le principe des algorithmes génétiques et de la méthode du recuit simulé. Dans cet algorithme, l'algorithme génétique utilise le recuit simulé comme un troisième opérateur, appelé *Improve operator*. Ses principes sont décrits dans l'algorithme 6. Les autres opérateurs sont définis comme suit :

Algorithm 6 Algorithme hybride *GASA*.

Intitialize p_m, p_c, $p_i \in$]0,1] and $i \leftarrow 1$
$maxGen \leftarrow 10000$
Generate population P_0
Evaluate P_0 and find the best solution π^*
$\pi_{Elite} \leftarrow \pi^*$
$stop\_criterion \leftarrow$ false
while Not $stop\_criterion$ **do**
  $P_i \leftarrow \emptyset$
  for $j = 1$ to $PopSize/2$ **do**
    Select two parents p_1 and p_2 from P_{i-1}
    offspring $\leftarrow (p_1, p_2)$
    With probability p_c, perform offspring := crossover(p_1,p_2)
    With probability p_m, mutate offspring
    With probability p_i, improve offspring by using SA (with $maxiter = 200$)
    Evaluate offspring and add it to P_i
  end for
  Add P_{i-1} to P_i
  Sort P_i
  Keep the $PopSize$ best solution in P_i
  Find the best solution π^* in P_i
  if π^* is better than π_{Elite} **then**
    $\pi_{Elite} \leftarrow \pi^*$
  end if
  if $fitness(\pi_{Elite}) = 0$ OR $i = maxGen$ **then**
    $stop\_criterion \leftarrow$ true
  end if
  Update(i)
end while

– **Sous-population initiale** : la population initiale P_0 est choisie aléatoirement. Dans l'algorithme 6, $PopSize = 30$ est la taille de chaque population P_0. Durant chacune des $maxGen$ générations,

les résultats de $PopSize$ sont générés à travers le croisement de parents sélectionnés à partir de la sous-population.

– **Fonction de performance** : la fonction *fitness* est définie dans l'équation 4.1 suivante :

$$Fitness(solution) = Total\_Number\_Access\_all\_data - Number\_Access(solution). \tag{4.1}$$

– **Opérateur de sélection** : ici, nous utilisons une sélection aléatoire proportionnelle à la fonction *fitness*. À chaque étape, un individu est sélectionné aléatoirement pour être placé dans la prochaine génération.
– **Opérateur de croisement** : il est défini par la probabilité $P_c = 0.6$ et est appliqué à deux parents sélectionnés aléatoirement parmi la population. Dans cette approche, nous avons utilisé le croisement en deux points et cela étant donné qu'il donne toujours les mêmes résultats que le croisement en un point (dans notre cas).
– **Opérateur de mutation** : il est défini par la probabilité $P_m = 0.06$ et change le résultat en passant les termes de la solution de 1 à 0 ou de 0 à 1 (tant que la solution satisfait les contraintes).
– **Opérateur *Improve*** : avec la probabilité $P_i = 0.01$, l'algorithme SA est appliqué au nouveau résultat.

La figure 4.2 donne les résultats moyens obtenus sur 30 exécutions indépendantes de TS et 30 exécutions indépendantes de l'algorithme hybride *GASA*. *GASA* trouve toujours la meilleure solution et dépasse les performances énergétiques de TS. En effet, *GASA* consomme de 76.23% (StatemateCE) jusqu'à 98.92% (ShaCE) moins d'énergie que TS.

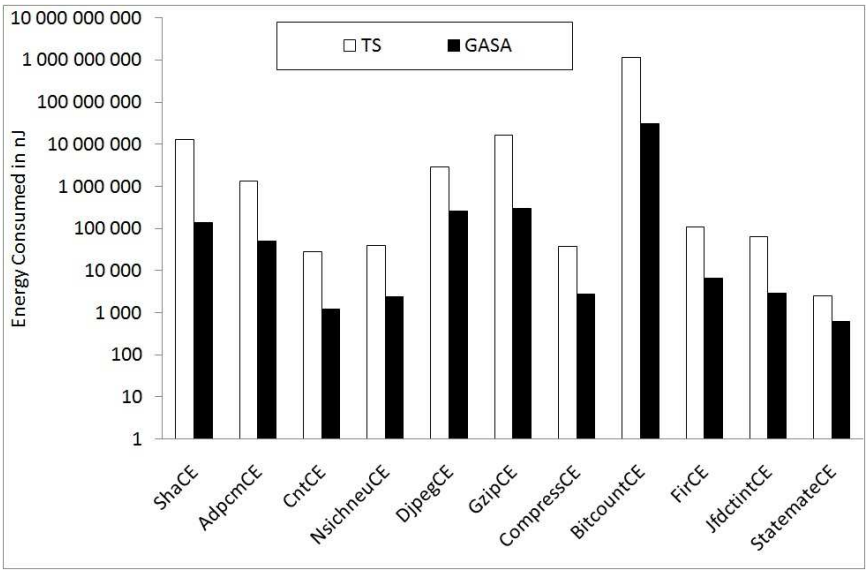

FIGURE 4.2 – Énergies consommées par l'algorithme hybride *GASA*.

4.3.3 PSOSA

4.3.3.1 Réduction de la consommation d'énergie

Dans cette sous-section, nous présentons l'algorithme hybride proposé dans [Idrissi Aouad *et al.*, 2010a] et noté *PSOSA*, qui combine les avantages de la méthode d'optimisation par essaims particulaires, présentée en section 3.5.2 (qui a une forte capacité de recherche globale) et la méthode du recuit simulé (qui a une forte capacité de recherche locale). Quelques travaux utilisant ce type d'hybridation pour résoudre d'autres problèmes peuvent être trouvés dans [Xia et Wu, 2006; Chaojun et Qiu, 2006; Fang *et al.*, 2007].

En hybridant SA avec PSO, l'algorithme est capable d'éviter les optima locaux. Toutefois, si SA est hybridé avec PSO à chaque itération, les coûts de calculs augmenteront et au même moment la rapidité de convergence de PSO pourra être affaiblie. C'est pourquoi, dans notre algorithme, SA est hybridé avec PSO avec une probabilité $p_i = 0.01$ ($i.e.$ comme un opérateur de mutation dans les algorithmes génétiques). Ainsi, l'algorithme *PSOSA* est en mesure de garder une convergence rapide (la plupart du temps) grâce à PSO et puis risque de sortir d'un optimum local avec l'aide de SA qui est appliqué à chaque particule courante dans l'essaim. Les principes de notre algorithme *PSOSA* sont détaillés dans l'algorithme 7, où :

- **Solution (*particule*)** : une solution peut être représentée par un tableau de taille égale au nombre de données. Chaque élément de ce tableau désigne si une donnée est incluse dans la SPM ('1') ou pas ('0'). L'algorithme *PSOSA* débute par un essaim initialisé aléatoirement.
- **Fonction *Evaluate*** : fonction objectif définie par l'équation 4.2 suivante :

$$\begin{aligned} Evaluate(solution) = Total\_Number\_Access\_all\_data \\ - Number\_Access(solution). \end{aligned} \tag{4.2}$$

- **Équation de mise à jour de la particule** : chaque dimension j de la particule i est mise à jour en utilisant l'équation 4.3.

$$x_{ij} = \begin{cases} 1 & \text{si} \quad rand < sigm(v_{ij}) \\ 0 & \text{sinon} \end{cases} \tag{4.3}$$

  où $sigm(v_{ij})$ est la fonction *sigmoid* utilisée pour mettre à l'échelle les vélocités comprises entre 0 et 1 et définie comme suit :

$$sigm(v_{ij}) = \frac{1}{1 + exp(-v_{ij})} \tag{4.4}$$

- **Fonction *Generate*** : SA est un algorithme de recherche aléatoire de l'espace des solutions faisables utilisant la notion de relation de voisinage. Soit S l'ensemble des solutions faisables et $f : S \rightarrow \Re$ la fonction objectif à minimiser. Une relation de voisinage est une relation binaire $N \subseteq S \times S$ avec quelques propriétés spécifiques. L'interprétation de $N(s, s')$ est que la solution s est voisine de la solution s' dans l'espace de recherche de toutes les solutions S. Une heuristique de voisinage commence sa recherche par une solution initiale s_0 et à chaque étape, elle passe d'une solution courante à une solution voisine selon certaines règles spécifiques à l'heuristique. À chaque itération, notre algorithme SA génère un voisin aléatoire de la solution courante $current\_solution$ (ligne 25). Nous définissons la relation de voisinage comme suit :

Algorithm 7 Algorithme hybride *PSOSA*.

$iter \leftarrow 0$, $pi \leftarrow 0.01$
Initialize $swarm\_size \leftarrow 30$ particles
maximum iterations $maxiter \leftarrow 4000$
while $iter < maxiter$ **do**
  for each particle $i \leftarrow 1$ to $swarm\_size$ **do**
    Evaluate ($particle(i)$)
    if the fitness value is better than the best fitness value ($pbest$) in history **then**
      Update current value as the new $pbest$
    end if
  end for
  Choose the particle with the best fitness value in the neighborhood ($gbest$)
  for each particle $i \leftarrow 1$ to $swarm\_size$ **do**
    Update particle velocity according to Equation (3.5)
    Enforce velocity bounds ($range[0, 4]$)
    Update particle position according to Equation (4.3)
    Validate particle position
  end for
  if $random(0, 1) < pi$ **then**
    {With probability pi, improve $current\_solution$ by using SA algorithm}
    $iterSA \leftarrow 0$
    Initialize (T)
    $best\_solution \leftarrow current\_solution$
    $Ccost \leftarrow$ Evaluate ($current\_solution$)
    while $iterSA < 400$ **do**
      $Neighbor \leftarrow$ Generate ($current\_solution$)
      $Ncost \leftarrow$ Evaluate ($Neighbor$)
      if Accept (Ccost, Ncost, T) **then**
        $current\_solution \leftarrow Neighbor$
        $Ccost \leftarrow Ncost$
      end if
      Update ($iterSA$, $best\_solution$)
      Update (T) according to Equation (3.2)
    end while
    $current\_solution \leftarrow best\_solution$
  end if
  Update ($iter$)
end while

- avec une probabilité égale à pi, la valeur de chaque élément de $current\_solution$ passe de 1 à 0 ou de 0 à 1.
- tant que *pas faisable*[28]($current\_solution$) retirer la donnée j ayant un faible nombre d'accès de $current\_solution$ ($current\_solution[j] \leftarrow 0$).

– **Fonction *Accept*** : cette fonction est l'idée principale dans l'approche SA. Elle spécifie la proba-

28. $current\_solution$ devant satisfaire la contrainte de ne pas dépasser la capacité maximale de la SPM.

bilité d'accepter un mouvement de $current\_solution$ à une solution voisine $Neighbor$, ce qui dépend également de la température (T). La fonction *Accept* doit satisfaire les conditions suivantes :

1. $p = 1$ si la solution $Neighbor$ est meilleure que $current\_solution$ en termes de fonction coût f ($i.e.$ $f(Neighbor) < f(current\_solution)$ dans un problème de minimisation).
2. si $Neighbor$ est pire que $current\_solution$, la valeur de p est positive ($i.e.$ elle permet le déplacement vers une solution pire) mais décroît avec $|f(Neighbor) - f(current\_solution)|$.
3. pour $current\_solution$ et $Neighbor$ fixes, lorsque $Neighbor$ est pire que $current\_solution$, la valeur de p décroît avec le temps et tend vers 0.

La fonction $Accept(c_{cost}, n_{cost}, T)$ est décidée par la probabilité d'accepter la configuration de la solution $Neighbor$. Cette probabilité est donnée par la formule suivante :

$$p = \begin{cases} 1 & \text{si} \quad N_{cost} < C_{cost} \\ \frac{1}{2} rand * (1 + e^{\frac{C_{cost} - N_{cost}}{T}}) & \text{sinon} \end{cases} \tag{4.5}$$

où T est la température et $rand$ un nombre aléatoire généré indépendamment dans l'intervalle $[0, 1]$.

La figure 4.3 présente les résultats moyens obtenus sur 30 exécutions de chacun des algorithmes TS et *PSOSA*. *PSOSA* fait mieux que TS en termes de consommation énergétique. En effet, ces résultats montrent que *PSOSA* consomme de 76.23% (StatemateCE) jusqu'à 98.92% (ShaCE) moins d'énergie que TS.

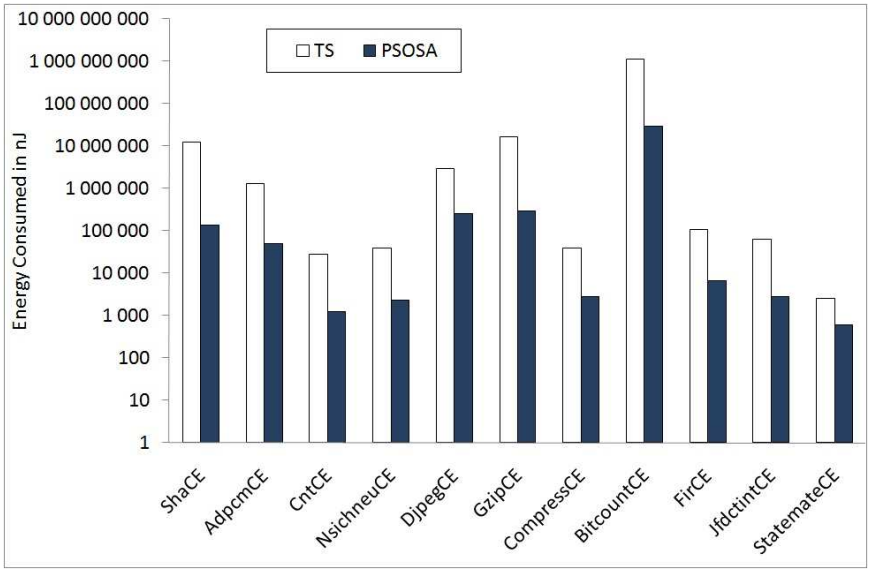

FIGURE 4.3 – Énergies consommées par l'algorithme hybride *PSOSA*.

Algorithm 8 Algorithme hybride *PSOSA* adapté au cas des fonctions tests mathématiques.

$iter \leftarrow 0$, $cpt \leftarrow 0$, Initialize $swarm\_size$ particles
$stop\_criterion \leftarrow$ maximum iterations or $Optimal\_solution$ is not attained
while Not stop_criterion **do**
  for each particle $i \leftarrow 1$ to $swarm\_size$ **do**
    Evaluate (particle(i))
    if the fitness value is better than the best fitness value ($cbest$) in history **then**
      Update current value as the new $cbest$.
    end if
  end for
  Choose the particle with the best fitness value in the neighborhood ($gbest$)
  for each particle $i \leftarrow 1$ to $swarm\_size$ **do**
    Update particle velocity according to Equation (3.5)
    Enforce velocity bounds
    Update particle position according to Equation (3.6)
    Enforce particle bounds
  end for
  if there is no improvement of global best solution **then**
    $cpt \leftarrow cpt + 1$
  end if
  Update global best solutions
  if $cpt = Kmax$ **then**
    $cpt \leftarrow 0$
    $iterSA \leftarrow 0$, Initialize (T) {Apply SA to global best solution }
    $stop\_criterion \leftarrow$ SA_maximum_iterations or $Optimal\_solution$ is not attained
    $current\_solution \leftarrow global\_best\_solution$
    $current\_cost \leftarrow$ Evaluate ($current\_solution$)
    while Not stop_criterion **do**
      while inner-loop stop criterion **do**
        $Neighbour \leftarrow$ Generate ($current\_solution$)
        $Neighbour\_cost \leftarrow$ Evaluate ($Neighbour$)
        if Accept (current_cost, Neighbour_cost, T) **then**
          $current\_solution \leftarrow Neighbour$
          $current\_cost \leftarrow Neighbour\_cost$
        end if
        $iterSA \leftarrow iterSA + 1$
        Update ($global\_best\_solution$)
      end while
      Update (T) according to Equation (3.2)
      Update ($stop\_criterion$)
    end while
  end if
  $iter \leftarrow iter + 1$
  Update ($stop\_criterion$)
end while

4.3.3.2 Optimisation de fonctions tests mathématiques

Dans cette sous-section, nous présentons l'algorithme hybride proposé dans [Idoumghar *et al.*,]. C'est une adaptation de l'algorithme 7 présenté en section 4.3.3.1. Dans le cas des fonctions tests multimodales, SA est hybridé avec PSO chaque K itérations, K étant prédéfini à $270 \sim 500$ selon nos expérimentations. Les principes de notre algorithme *PSOSA* sont détaillés dans l'algorithme 8, où :

- **Description d'une particule** : chaque particule (solution) $X \in S$ est représentée par ses $n > 0$ composants $i.e.$, $X = (x_1, x_2, ..., x_i, ..., x_n)$ où $i = 1, 2, ..., n$ et n représente la dimension du problème d'optimisation à résoudre.
- **Essaim initial** : l'essaim initial correspond à la population de particules qui vont évoluer. Chaque particule x_i est initialisée avec une valeur aléatoire uniforme entre les bornes inférieure et supérieure de l'intervalle définissant le problème d'optimisation.
- **Algorithme SA** : s'il n'y a pas d'améliorations de la meilleure solution globale durant les dernières K itérations, cela signifie que l'algorithme est coincé dans un optimum local. Afin de s'en échapper, SA est appliqué sur la meilleure solution globale. La performance de SA dépend de la définition de différents paramètres de contrôle :

  - **Température initiale** : [Kirkpatrick *et al.*, 1983] suggèrent qu'une température initiale convenable est celle qui conduit à une probabilité moyenne χ_0 d'une solution qui augmente f étant acceptée d'environ 0.8. La valeur de T_0 dépendra évidemment de la mise à l'échelle de f et, par conséquent, sera un problème spécifique. Elle peut être estimée en effectuant une recherche initiale (100 itérations dans les simulations suivantes) dans laquelle toutes les augmentations de f sont acceptées et en calculant l'augmentation moyenne observée de la fonction objectif δf. T_0 est alors donnée par :

$$T_0 = -\frac{\delta f}{ln(\chi_0)} \tag{4.6}$$

  - **Fonction *Accept*** : $Accept(current\_solution, Neighbour, T)$ est déterminée par la probabilité d'acceptation donnée par l'équation 4.7, qui est la probabilité d'accepter la configuration $Neighbour$.

$$p = \begin{cases} 1 & \text{si} \quad f(s_n) - f(s_c) < 0 \\ exp(\frac{-|f(s_n)-f(s_c)|}{T}) & \text{sinon} \end{cases} \tag{4.7}$$

  - **Fonction *Generate*** : le voisinage de chaque solution x est généré en utilisant l'équation suivante :

$$x \leftarrow x + d\sigma r \tag{4.8}$$

    où d est la direction du nouveau voisinage et prend comme valeur soit 1, soit -1, σ est un nombre aléatoire généré avec une distribution gaussienne $(0, 1)$ et r est une constante qui correspond au rayon du générateur de voisinage.
  - **Critère d'arrêt SA** : lorsque le système atteint 3000 évaluations de fonction ou bien, lorsque le nombre maximal d'évaluations de fonctions ou la solution optimale ne sont pas atteints.
  - **Décrémentation de la température** : la température réduisant la fonction objectif la plus couramment utilisée est géométrique (voir l'équation 3.2). Dans les simulations suivantes $\gamma = 0.99$.
  - ***inner-loop*** : la longueur de chaque niveau de température détermine le nombre $q = 150$ des solutions générées à chaque température T.

Nous comparons notre algorithme *PSOSA* avec, d'un côté, l'algorithme *TL-PSO* [Nakano *et al.*, 2008] et avec les algorithmes *PSO* (algorithme classique), *ATREPSO*, *QIPSO* et *GMPSO* [Pant *et al.*, 2008] d'un autre côté. Tous ces algorithmes ont été décrits en section 2.2. Nous exécutons notre algorithme hybride sur les fonctions tests mathématiques présentées en section 1.2. Ces fonctions contiennent beaucoup d'optima locaux dans leurs espaces de solution. La quantité d'optima locaux augmente avec la complexité croissante des fonctions, $i.e.$ avec l'augmentation de la dimension. Dans nos expérimentations, nous avons utilisé ces fonctions en dimension élevée (20 ou 30 selon l'algorithme considéré), sauf pour le cas des fonctions d'Himmelblau et de Shubert qui sont en deux dimensions par définition.

Comparaison avec l'algorithme *TL-PSO* : Comme décrit dans [Nakano *et al.*, 2008], nous exécutons notre algorithme *PSOSA* sur les quatre fonctions tests mathématiques suivantes : Rastrigin, Schwefel, Griewank et Rosenbrock. Ici, le nombre de particules dans l'essaim est de 30. La dimension de recherche est égale à $n = 20$ et le nombre d'évaluations de la fonction objectif est de 60000 ($i.e.$ 2000×30). Les résultats obtenus lors des simulations numériques sont donnés dans le tableau 4.2.

TABLE 4.2 – Comparaison entre *PSOSA* et *TL-PSO*.

	Résultats moyens		**Meilleurs résultats**		**Pires résultats**	
Fonction	TL-PSO	PSOSA	TL-PSO	PSOSA	TL-PSO	PSOSA
Rastrigin	8.7161	$5.23034e-05$	4.0589	$4.35994e-09$	17.053	$8.15835e-04$
Schwefel	445.8830	206.693255	0.6748	0	829.4431	355.318
Griewank	0.0228	$1.18294e-03$	$1.0507e-5$	$1.2299e-15$	0.0739	0.0172263
Rosenbrock	19.5580	0.58359742	0.1331	0.0442625	71.5439	1.36171

Ces résultats indiquent les valeurs moyennes, les meilleures valeurs ainsi que les pires valeurs obtenues sous la même condition de 50 essais. En analysant le tableau 4.2, nous pouvons constater que les résultats obtenus avec notre algorithme *PSOSA* sont excellents en comparaison avec ceux qui sont obtenus par l'algorithme *TL-PSO*.

Comparaison avec les algorithmes *PSO, ATREPSO, QIPSO* et *GMPSO* : Afin d'effectuer une comparaison équitable entre *PSO*, *ATREPSO*, *QIPSO*, *GMPSO* et notre algorithme *PSOSA*, nous avons fixé, comme indiqué dans [Pant *et al.*, 2008], la même méthode pour la génération aléatoire de nombres de telle sorte que la population initiale de l'essaim soit la même pour tous les cinq algorithmes. Le nombre de particules de l'essaim est de 30. Les cinq algorithmes utilisent un poids d'inertie ω, linéairement décroissant, commençant à 0.9 et se terminant à 0.4, avec des paramètres définis par l'utilisateur $c_1 = c_2 = 2.0$. Pour chaque algorithme, le nombre d'évaluations de la fonction objectif est de 300000. Pour chaque paramètre expérimental, 30 exécutions ont été effectuées et la moyenne des meilleures solutions trouvées a été enregistrée. La solution moyenne et l'écart-type [29] trouvés par les cinq algorithmes sont donnés dans le tableau 4.3.

Les résultats numériques donnés dans le tableau 4.3 montrent que :

– Tous les algorithmes dépassent l'algorithme *PSO* classique.
– *PSOSA* a de meilleures performances en comparaison avec *PSO*, *QIPSO*, *ATREPSO* et *GMPSO*, excepté pour les fonctions Sphère et Ackley.

29. Notons que l'écart-type donne une indication sur la stabilité des algorithmes.

TABLE 4.3 – Comparaison des moyennes/écart-types des solutions obtenues en utilisant *PSOSA*, *PSO*, *QIPSO*, *ATREPSO* et *GMPSO*.

Fonction	PSO	QIPSO	ATREPSO	GMPSO	PSOSA
Rastrigin	22.339158	11.946888	19.425979	20.079185	0
	15.932042	9.161526	14.349046	13.700202	0
Sphère	$1.167749e-45$	0.000000	$4.000289e-17$	$7.263579e-17$	$5.3656e-32$
	5.222331e-46	0.000000	0.000246	6.188854e-17	2.98492e-31
Griewank	0.031646	0.011580	0.025158	0.024462	$3.32255e-20$
	0.025322	0.012850	0.028140	0.039304	2.68415e-20
Rosenbrock	22.191725	8.939011	19.490820	14.159547	0.227048188
	1.615544e+04	3.106359	3.964335e+04	4.335439e+04	0.243978057
Noisy	8.681602	0.451109	8.046617	7.160675	0.002019998
	9.001534	0.328623	8.862385	7.665802	0.000650347
Schwefel	−6178.559896	−6355.586640	−6183.677600	−6047.670898	−8379.66
	4.893329e+02	477.532584	469.611104	482.926738	2.20425e-19
Ackley	$3.483903e-18$	$2.461811e-24$	0.018493	$1.474933e-18$	$7.43546e-16$
	8.359535e-19	0.014425	0.014747	1.153709e-08	1.09382e-15
Michalewizc	−18.159400	−18.469600	−18.982900	−18.399800	−19.62555806
	1.051050	0.092966	0.272579	0.403722	0.00926944
Himmelblau	−3.331488	−3.783961	−3.751458	−3.460233	−3.78396
	1.243290	0.190394	0.174460	0.457820	3.16001e-15
Shubert	−186.730941	−186.730942	−186.730941	−186.730942	−186.730942
	1.424154e-05	0.000000	1.424154e-05	1.525879e-05	8.66746e-14

– Pour la fonction Sphère, *QIPSO* obtient de meilleurs résultats que ceux qui sont obtenus par *PSOSA*. Mais, lorsque le nombre maximal d'itérations est fixé à 1.5×10^6, *PSOSA* trouve la valeur optimale.
– Pour la fonction d'Ackley, *QIPSO* obtient de meilleurs résultats que *PSOSA*. Mais, *PSOSA* a un écart-type plus petit que celui de *QIPSO*.

4.3.4 MPSOM

Dans cette sous-section, nous présentons l'algorithme hybride que nous avons proposé dans [Idoumghar *et al.*, 2010] et qui est noté *MPSOM*. Cet algorithme se base sur l'algorithme de PSO auquel il incorpore un opérateur de mutation et la règle de *Metropolis* afin de sortir d'un optimum local. Les détails sont donnés dans l'algorithme 9.

4.3.4.1 Réduction de la consommation d'énergie

Afin de réduire la consommation d'énergie en mémoire, quelques caractéristiques de l'algorithme 9 doivent être adaptés :

– **Solution (*particule*)** : une solution peut être représentée par un tableau ayant une taille égale au nombre de données. Chaque élément de ce tableau indique si la donnée est allouée en SPM ('1') ou pas ('0'). Dans ce cas, l'algorithme *MPSOM* débute avec un essaim initial aléatoire.
– **Fonction *Evaluate*** : fonction objectif définie par l'équation 4.9.

Algorithm 9 Algorithme hybride *MPSOM* adapté au cas des fonctions tests mathématiques.

```
1: Initialize swarm_size particles
2: Evaluate (Swarm)
3: stop_criterion ← maximum evaluation functions
4: noImprove ← 0
5: Initialize inertia factor w ← w_0
6: Initialize Initial Temperature T ← T_0
7: while Not stop_criterion do
8:    Sort (Swarm)
9:    if nonimprove < k then
10:      for each particle i ← 1 to swarm_size do
11:         Accept (X, cbest, T)
12:         Update velocity according to Equation (4.12)
13:         Enforce velocity bounds
14:         Update particle position according to Equation (4.15)
15:         Enforce position bounds
16:      end for
17:   else
18:      {Mutation operator}
19:      noImprove ← 0
20:      for each particle i ← 1 to swarm_size do
21:         Initialize its velocity to maximum velocity allowed
22:      end for
23:   end if
24:   Evaluate (Swarm)
25:   if there is no improvement of global best solution then
26:      noImprove ← noImprove + 1
27:   else
28:      Update global best solution
29:      noImprove ← 0
30:   end if
31:   Update inertia factor w by using Equation (4.14)
32:   Update (T)
33:   Update (stop_criterion)
34: end while
```

$$Evaluate(solution) = Total\_Number\_Access\_all\_data - Number\_Access(solution). \tag{4.9}$$

– **Équation de mise à jour de la particule** : chaque dimension j de la particule i est mise à jour en utilisant l'équation 4.10.

$$x_{ij} = \begin{cases} 1 & \text{si} \quad rand < sigm(v_{ij}) \\ 0 & \text{sinon} \end{cases} \tag{4.10}$$

La figure 4.4 présente les résultats obtenus en comparant TS à l'algorithme *MPSOM*. Dans ce cas, *MPSOM* trouve toujours la meilleure solution car les valeurs moyennes sont égales aux meilleures

valeurs. *MPSOM* fait mieux que TS en termes de consommation énergétique. En effet, ces résultats montrent que *MPSOM* consomme de 76.23% (StatemateCE) jusqu'à 98.92% (ShaCE) moins d'énergie que TS.

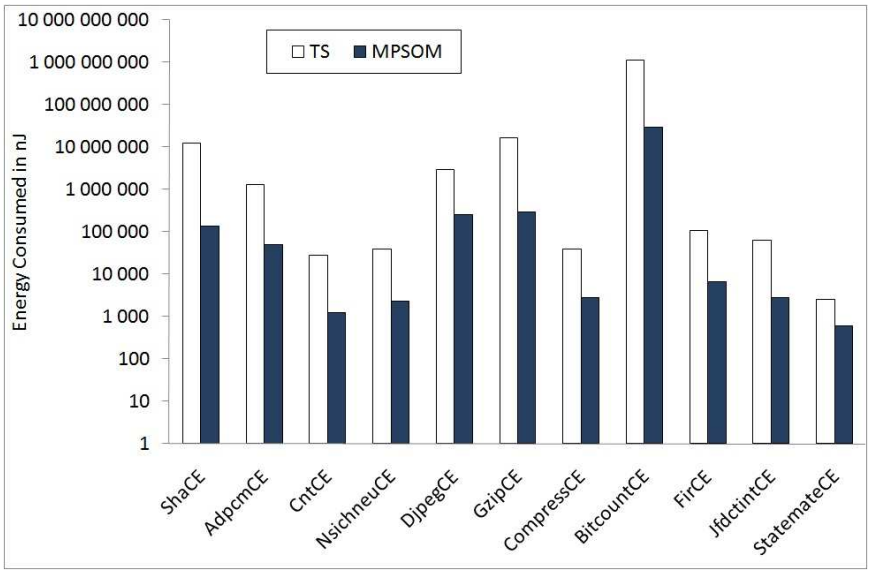

FIGURE 4.4 – Énergies consommées par l'algorithme hybride *MPSOM*.

4.3.4.2 Optimisation de fonctions tests mathématiques

Nous commençons par adapter l'algorithme *MPSOM* au cas des fonctions tests mathématiques. L'opérateur de mutation est hybridé avec PSO chaque K itérations si aucune amélioration de la meilleure solution globale n'est obtenue. $K = 60$ selon nos expérimentations. Le principe de *MPSOM* fonctionne comme illustré dans l'algorithme 9, où :

- **Description d'une particule** : chaque particule (solution) est représentée par sa :
  - position courante $X \in S$ représentée par ses $n > 0$ composants, $i.e.$ $X = (x_1, x_2, ..., x_i, ..., x_n)$ où $i = 1, 2, ..., n$ et n représente la dimension du problème d'optimisation à résoudre.
  - meilleure solution obtenue précédemment (appelée $cbest$).
  - vélocité qui correspond au taux de changement de la position.

- **Essaim initial** : correspond à la population de particules qui vont évoluer. Chaque particule x_i est initialisée avec une valeur uniforme aléatoire entre les bornes inférieure et supérieure de l'intervalle définissant le problème d'optimisation.
- **Fonction *Evaluate*** : fonction objectif ou *fitness* de l'algorithme *MPSOM* que nous voulons minimiser.
- **Tri** : toutes les particules de l'essaim sont triées dans l'ordre décroissant de leur fonction objectif.
- **Fonction *Accept*** : $Accept(cbest, X, T)$ est déterminée par la probabilité d'acceptation donnée par l'équation 4.11, qui est la probabilité d'accepter la position courante d'une particule comme étant son $cbest$.

$$p = \begin{cases} 1 & \text{si } f(X) \leq f(cbest) \text{ ou} \\ & \quad \frac{1}{2} rand \times (1 + e^{\frac{f(X) - f(cbest)}{T}}) < 1 \\ 0 & \text{sinon} \end{cases} \tag{4.11}$$

Dans l'équation 4.11, T est la température. Elle joue le même rôle que la température dans la méthode du recuit simulé (section 3.4.2). $rand$ est un nombre aléatoire indépendamment généré dans l'intervalle $[0, 1]$.

– **Mise à jour de la vélocité** : nous proposons ici, un voisinage topologique qui est déterminé dynamiquement durant le processus de recherche selon la fonction objectif de l'essaim. Après avoir trié la population de l'essaim pour chaque i^{ieme} particule, un ensemble N_i de ses voisins est défini comme : $N_i = \{particule\ k\ /\ k \geq i\ et\ k \leq swarm\_size\}$. Ensuite, afin d'ajuster la vélocité de la i^{ieme} particule, nous utilisons l'équation 4.12 suivante :

$$\begin{aligned} v_{ij} &= \omega \times v_{ij} + c_1 \times rand \times (cbest_{ij} - x_{ij}) \\ &\quad + min(V_{max}, \sum_{k=i}^{swarm\_size} \frac{cbest_{kj} - x_{ij}}{k}) \end{aligned} \tag{4.12}$$

Dans l'équation 4.12, le composant social de la i^{ieme} particule est calculé comme étant la moyenne pondérée des meilleures particules dans N_i. En procédant de la sorte, une particule n'est pas seulement influencée par la meilleure solution globale, mais elle ajuste sa vitesse en fonction de la moyenne pondérée des solutions qui sont meilleures que les siennes. Cela signifie simplement qu'il est mieux pour la particule de suivre un groupe de particules plutôt que de suivre une seule particule.

– **Mise à jour de la température** : afin d'éviter à l'algorithme d'être coincé dans un minimum local, le taux de réduction de la température doit être lent. Nous utilisons pour cela l'équation 4.13 où $i = 0, 1, ...$ et $\gamma = 0.99$.

$$T_{i+1} = \gamma\ T_i \tag{4.13}$$

.

– **Mise à jour du facteur d'inertie** : le facteur d'inertie, utilisé pour contrôler la vitesse de la relation entre la vitesse précédente et la vitesse courante de chaque particule, est défini par l'équation 4.14 où :

$$w_k = w_0 \times \left(1 - \frac{T_0 - T_k}{T_0}\right) \tag{4.14}$$

  – $w_0 = 0.9$ correspond à la valeur du poids au démarrage de l'algorithme. Notons qu'avec de faibles valeurs de w, la région de recherche de l'algorithme peut se trouver autour de la meilleure solution, alors qu'avec des valeurs élevées de w, l'algorithme peut améliorer l'exploration (recherche globale).
  – T_0 est une température initiale et T_k représente la température à la k^{ieme} itération.

– **Opérateur de mutation** : s'il n'y a pas d'améliorations de la meilleure solution globale pendant les dernières K itérations, cela signifie que l'algorithme est coincé dans un optimum local. Afin d'en échapper, notre algorithme *MPSOM* utilise l'opérateur de mutation en se basant sur l'idée suivante : en attribuant une vélocité maximale permise à chaque particule, l'équation 4.15 assure

que toutes les particules vont sortir de l'optimum local et *MPSOM* peut avoir une large capacité d'exploration.

$$x_{i+1} = x_i + v_{i+1} \tag{4.15}$$

Comparaison avec les algorithmes *GPSO-J, CLPSO* et *ATM-PSO* : Nous comparons notre algorithme *MPSOM* avec les algorithmes suivants : l'algorithme *Gaussian PSO with jumps* (*GPSO-J*) [Krohling, 2005], l'algorithme *Comprehensive Learning PSO* (*CLPSO*) [Liang *et al.*, 2006] et l'algorithme *Particle Swarm Optimizer with Adaptive Tabu and Mutation* (*ATM-PSO*) [Yu-Xuan *et al.*, 2010]. Tous ces algorithmes ont été décrits en section 2.2. Nous exécutons notre algorithme hybride sur sept des fonctions tests mathématiques présentées en section 1.2. Le tableau 4.4 récapitule leurs caractéristiques. Ces fonctions contiennent beaucoup d'optima locaux dans leurs espaces de solution. La quantité d'optima locaux augmente avec la complexité croissante des fonctions, $i.e.$ avec l'augmentation de la dimension. Dans nos expérimentations, nous avons utilisé ces fonctions en dimension élevée égale à 30.

TABLE 4.4 – Synthèse des fonctions tests mathématiques utilisées.

Fonction	**Problème**	**Rang**	$f(x^*)$	ϵ	**Classification**
Sphère	$\sum_{i=1}^{n} x_i^2$	$[-100; 100]$	0	0.01	Unimodale
Rastrigin	$\sum_{i=1}^{n}(x_i^2 - 10\cos(2\pi x_i) + 10)$	$[-5.12; 5.12]$	0	10	Multimodale
Griewank	$\frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n}\cos(\frac{x_i}{\sqrt{i}}) + 1$	$[-600; 600]$	0	0.1	Multimodale
Rosenbrock	$\sum_{i=1}^{n-1}\left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right]$	$[-2.048; 2.048]$	0	100	Unimodale
Schwefel	$420.9687\, n - \sum_{i=1}^{n}\left[x_i \sin(\sqrt{\lvert x_i\rvert})\right]$	$[-500; 500]$	0	2000	Multimodale
Ackley	$20 + e - 20\, e^{-0.2\,(\frac{1}{n}\sum_{i=1}^{n} x_i^2)^{\frac{1}{2}}} - e^{\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)}$	$[-30; 30]$	0	0.1	Multimodale
Noisy	$\left[\sum_{i=1}^{n}(i+1)x_i^4\right] + rand[0, 1]$	$[-1.28; 1.28]$	0	0.1	Bruitée

Nous fixons, comme c'est indiqué dans [Yu-Xuan *et al.*, 2010], le nombre de particules dans l'essaim à 20 et le nombre maximal d'évaluations de la fonction objectif à 5000 $D = 150000$. Pour chaque paramètre expérimental, 30 exécutions ont été effectuées et la moyenne des meilleures solutions trouvées a été enregistrée. Le tableau 4.5 donne les résultats obtenus.

L'analyse des résultats numériques donnés dans le tableau 4.5 montre que :

- Tous les algorithmes résolvent avec succès la fonction Sphère qui est la fonction unimodale la plus simple même si notre algorithme *MPSOM* obtient les meilleurs résultats.
- *MPSOM* dépasse les autres algorithmes quand il s'agit d'optimiser les problèmes multimodaux difficiles.
- *ATM-PSO* et *CLPSO* réussissent avec succès à résoudre la fonction de Schwefel. Par contre, les algorithmes *GPSO-J* et *MPSOM* échouent dans l'optimisation de cette fonction.

TABLE 4.5 – Résultats moyens et écart-types obtenus sur 30 exécutions indépendantes des algorithmes *CLPSO*, *GPSO-J*, *ATM-PSO* et *MPSOM* sur sept fonctions tests mathématiques (dimension $n = 30$ et taille de l'essaim $swarm\_size = 20$).

Fonction	CLPSO	GPSO-J	ATM-PSO	MPSOM
Sphère	$8.99e-14$	$3.85e-07$	$8.90e-104$	$9.4008e-113$
	± 4.66e-14	± 7.97e-07	± 3.18e-103	± 9.6549e-113
Rosenbrock	20.88	27.11	15.13	0.014441893
	± 2.58	± 16.91	± 8.71e-01	± 0.016312307
Schwefel	$1.76e-12$	1336.29	$2.36e-12$	4501.7
	± 3.27e-13	± 290.81	± 9.57e-13	± 1520.45862
Rastrigin	$1.34e-06$	15.86	0	0
	± 1.66e-06	± 4.52	± 0	± 0
Ackley	$8.45e-08$	$1.43e-03$	$2.59e-14$	$2.60232e-10$
	± 1.96e-08	± 1.36e-03	± 6.12e-15	± 5.27812e-11
Griewank	$1.95e-09$	$4.02e-02$	$2.22e-02$	0
	± 4.35e-09	± 4.02e-02	± 2.03e-02	± 0
Noisy	$8.18e-03$	$4.45e-03$	$9.77e-03$	$4.0721e-06$
	± 2.39e-03	±9.95e-04	±3.16e-03	± 1.055e-05

- Pour la fonction hautement multimodale de Rastrigin, *ATM-PSO* et *MPSOM* atteignent la solution optimale globale après 150000 évaluations de la fonction objectif.
- L'analyse des résultats obtenus pour la fonction d'Ackley montre que *ATM-PSO* et *MPSOM* obtiennent de meilleurs résultats moyens que *GPSO-J* et *CLPSO*.
- L'optimisation de la fonction de Griewank est un exemple de l'échec de tous les algorithmes (qui sont coincés dans des optima locaux), excepté notre algorithme *MPSOM* qui atteint la solution optimale globale.
- L'optimisation de la fonction bicarrée bruitée (*Noisy*) est un autre exemple du succès de *MPSOM* et illustre que les autres algorithmes sont adaptés à un environnement dynamique.

Taux de convergence $Q_{measure}$: Pour évaluer le taux de convergence de tous les algorithmes comparés, un seuil ϵ est positionné pour chaque fonction test comme cela est décrit dans [Yu-Xuan *et al.*, 2010]. Lorsque chaque algorithme atteint le seuil ϵ spécifié (voir le tableau 4.4) pour une certaine fonction test dans le k^{ieme} essai, le nombre d'évaluations de fonction FE_k nécessaire est enregistré et l'essai courant k est noté comme étant un essai réussi. [Feoktistov, 2006] propose un critère noté $Q_{measure}$, qui incorpore la mesure de la convergence et de la robustesse. Le $Q_{measure}$ utilisé afin d'évaluer les performances des algorithmes, est défini par l'équation 4.16 où n_t est le nombre total d'essais, n_s le nombre total d'essais réussis et le ratio de succès SR est défini avec $SR = \frac{n_s}{n_t}$.

$$Q_{measure} = \frac{n_t \sum_{i=1}^{n_s} FE_i}{n_s^2} \tag{4.16}$$

Les valeurs de $Q_{measure}$ des quatre algorithmes étudiés sont données pour les sept fonctions tests mathématiques dans le tableau 4.6. L'analyse des résultats de ce tableau montre que *MPSOM* converge plus rapidement que tous les autres algorithmes, excepté pour l'optimisation de la fonction de Schwefel

TABLE 4.6 – Valeurs de $Q_{measure}$ des algorithmes *CLPSO*, *GPSO-J*, *ATM-PSO* et *MPSOM* sur sept fonctions tests mathématiques. Les pourcentages en parenthèses sont les ratios de succès (SR).

Fonction	CLPSO	GPSO-J	ATM-PSO	MPSOM
Sphère	67977	19343	8101	216.66
	(100%)	(100%)	(100%)	(100%)
Rosenbrock	34654	6537	980	700.66
	(100%)	(100%)	(100%)	(100%)
Schwefel	24439	47987	9726	21271
	(100%)	(100%)	(100%)	(100%)
Rastrigin	87201	1795500	15187	3248
	(100%)	(7%)	(100%)	(100%)
Ackley	62659	42857	9664	4832
	(100%)	(100%)	(100%)	(100%)
Griewank	65437	18309	7140	5085
	(100%)	(90%)	(100%)	(100%)
Noisy	38094	11137	5988	1177.33
	(100%)	(100%)	(100%)	(100%)

où *ATM-PSO* fait mieux que *MPSOM*.

4.3.5 Comparaison entre algorithmes hybrides

Maintenant que nous avons expérimenté différents algorithmes hybrides, il convient de les tester entre eux et cela afin de mieux se rendre compte de l'efficacité de chacun par rapport à l'autre dans le cas du problème de la réduction de la consommation d'énergie en mémoire. Nous avions démontré au chapitre 3, que grâce aux algorithmes génétiques, nous avons pu atteindre les meilleurs gains en consommation d'énergie mémoire obtenus en comparaison avec ceux qui sont obtenus en utilisant les méthodes de la recherche avec tabous ou du recuit simulé. Nous avons donc comparé les résultats obtenus pour chacun des algorithmes hybrides avec les résultats obtenus en considérant les algorithmes génétiques. La figure 4.5 récapitule les résultats obtenus.

Nous remarquons que tous les algorithmes hybrides atteignent les mêmes performances énergétiques que les algorithmes génétiques. Ils convergent tous vers la solution optimale.

C'est pourquoi, afin de départager les différents algorithmes, nous avons calculé les temps d'exécutions moyens obtenus sur 30 exécutions de chacun des algorithmes. La figure 4.6 donne les résultats obtenus.

D'après cette figure, on constate que l'algorithme génétique est largement plus rapide que tous les algorithmes hybrides. Ainsi, en ce qui concerne le problème de la réduction de la consommation de l'énergie en mémoire dans les systèmes embarqués, il est suffisant d'appliquer une heuristique basée sur le principe des algorithmes génétiques seuls. En effet, à performances énergétiques égales, il vaut mieux choisir la solution la plus rapide sans forcément chercher à concevoir une solution hybride à tout prix. En revanche, pour des problèmes plus complexes comme celui de l'optimisation de fonctions tests mathématiques, il est nécessaire de recourir à des algorithmes hybrides plus robustes et plus performants

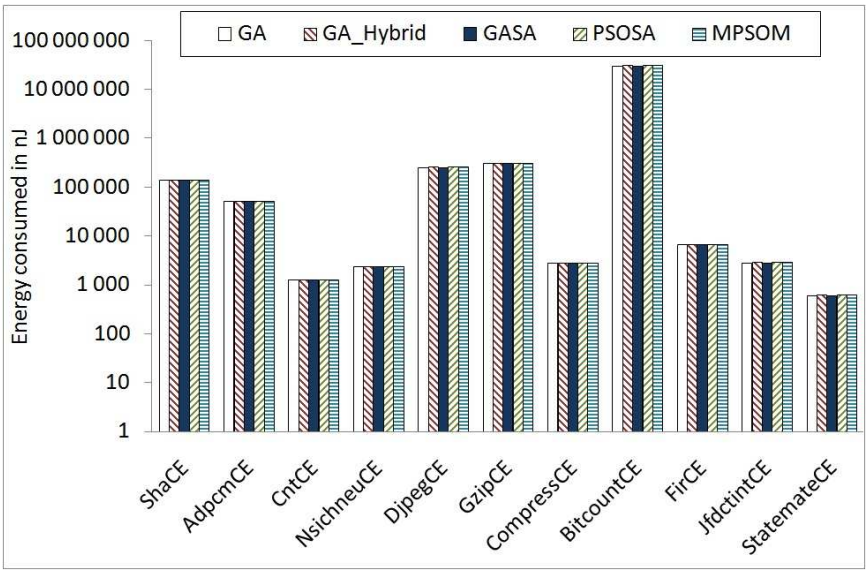

FIGURE 4.5 – Comparaison des énergies consommées par nos algorithmes hybrides.

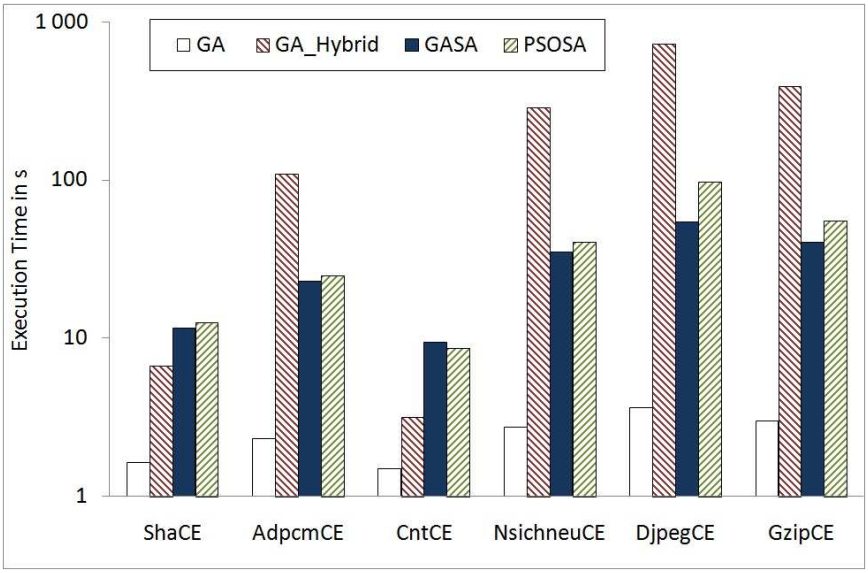

FIGURE 4.6 – Comparaison des temps d'exécution utilisés par nos algorithmes hybrides.

qu'un algorithme évolutif classique afin d'obtenir de meilleurs résultats. Ainsi, pour le problème de l'optimisation de fonctions tests mathématiques, les algorithmes hybrides *PSOSA* et *MPSOM* sont à préconiser.

4.4 Algorithmes distribués et coopératifs

Comme nous l'avons constaté, tous les algorithmes hybrides présentés en section 4.3 sont robustes et convergent tous vers la solution optimale. Cependant, tous ces algorithmes hybrides nécessitent un grand

temps de calcul pouvant aller à des jours d'exécution pour des problèmes complexes qui manipulent un très grand nombre de données. C'est pourquoi, dans le soucis de rendre nos algorithmes hybrides plus performants et plus rapides, nous proposons dans cette section des versions distribuées de nos algorithmes pouvant manipuler en parallèle toute une population de solutions. Ainsi, les tâches peuvent être réparties entre plusieurs processeurs : chacun d'eux est alors chargé d'optimiser une zone géographique donnée et des informations sont échangées périodiquement entre les processeurs voisins rendant les processeurs coopératifs entre eux. Pour les versions distribuées, nous utilisons un *cluster* d'un certain nombre de PCs (4 ou 6 selon l'algorithme distribué considéré) de mêmes caractéristiques ainsi que la bibliothèque MPICH2 (version 1.0.7) pour la communication entre les processeurs. Les processeurs sont connectés en réseau sous la forme d'une topologie en anneau et les machines exécutent nos algorithmes en plus de leur charge quotidienne normale.

4.4.1 SA distribué

Nous débutons par la version distribuée et coopérative de l'algorithme séquentiel *SA_Seq* présenté en sous-section 3.4.2. Notre algorithme SA distribué, proposé dans [Idrissi Aouad *et al.*, 2010c] et noté *SA_Dist*, est décrit dans l'algorithme 10. Pour la version distribuée *SA_Dist*, nous utilisons un *cluster* de 6 PCs. Chaque processeur, séparément et en parallèle, exécute *SA_Seq* (algorithme 3) sur sa solution courante. Après chaque $k = 300$ itérations, chacun des processeurs échange sa meilleure solution obtenue avec ses voisins en appelant la méthode *cooperation_send_receive*. À la k^{ieme} itération, chaque processeur envoie sa meilleure solution, puis continue d'améliorer sa solution courante et ensuite vérifie s'il n'a pas reçu une solution de ses voisins. Le processus continue avec l'amélioration séparée de chaque solution courante et cela pour un nombre maximum d'itérations. À la fin, la meilleure solution existante dans le réseau constitue le résultat final.

La figure 4.7 présente les résultats énergétiques moyens obtenus sur 30 exécutions de chaque algorithme. Les deux algorithmes *SA_Seq* et *SA_Dist* dépassent TS. Cependant, *SA_Dist* fait mieux que *SA_Seq* pour quelques programmes tests et réussit à consommer jusqu'à 16.36% (GzipCE) moins d'énergie. De plus, insistons sur le fait que *SA_Dist* trouve toujours la solution optimale sur les 30 exécutions contrairement à *SA_Seq*.

Nous avons également enregistré les temps d'exécution moyens nécessaires à chaque algorithme pour s'exécuter. La figure 4.8 donne les résultats obtenus en ne considérant que les plus grands programmes tests (en termes de taille). Nous notons que *SA_Dist* est clairement plus rapide que *SA_Seq*. En effet, *SA_Dist* nécessite de 65.63% (DjpegCE) jusqu'à 75.34% (CntCE) moins de temps d'exécution que *SA_Seq*.

4.4.2 GA_Hybrid distribué

Nous avons également implémenté deux versions distribuées et coopératives des algorithmes *TSGA* (présenté en section 4.2) et *GA_Hybrid* (algorithme 5). Ces versions distribuées sont notées *TSGA_Dist* et *GAHybrid_Dist* respectivement. Nous notons les versions séquentielles *TSGA_Seq* et *GAHybrid_Seq* respectivement. Les principes de *GAHybrid_Dist* sont décrits dans l'algorithme 11.

Pour les versions distribuées *TSGA_Dist* et *GAHybrid_Dist*, nous utilisons un *cluster* de 6 PCs. Chaque processeur, séparément et en parallèle, exécute soit *TSGA* soit *GA_Hybrid* sur sa solution courante. Après chaque $k = 300$ itérations, chacun des processeurs échange sa meilleure solution obtenue avec ses voisins en appelant la méthode *cooperation_send_receive*. À la k^{ieme} itération, chaque processeur envoie sa meilleure solution, continue d'améliorer sa solution courante et vérifie s'il n'a pas reçu

Algorithm 10 Algorithme distribué et coopératif *SA_Dist* s'exécutant sur chaque processeur.

```
1: maximum iterations maxiter ← 1667
2: Initialize T
3: stop_criterion ← maxiter, iter ← 0
4: Initialize current_solution
5: best_solution ← current_solution
6: while Not stop_criterion do
7:    cooperation_send_receive (received_solution)
8:    if received_solution is better than current_solution then
9:       current_solution ← received_solution
10:      if current_solution is better than best_solution then
11:         best_solution ← current_solution
12:      end if
13:      iter ← iter + 1
14:   else
15:      while inner-loop stop criterion do
16:         Neighbor ← Generate (X)
17:         Neighbor_cost ← Evaluate (Neighbor)
18:         if Accept (current_cost, Neighbor_cost, T) then
19:            current_solution ← Neighbor
20:            current_cost ← Neighbor_cost
21:         end if
22:         iter ← iter + 1
23:         if current_solution is better than best_solution then
24:            best_solution ← current_solution
25:         end if
26:      end while
27:      Update(T) according to T = 0.99 × T
28:   end if
29:   Update (stop_criterion)
30: end while
```

une solution de ses voisins. Le processus continue avec l'amélioration séparée de chaque solution courante et cela pour un nombre maximal d'itérations. À la fin, la meilleure solution existante dans le réseau constitue le résultat final.

Les deux versions séquentielle et distribuée produisent les mêmes résultats énergétiques pour chacun des deux algorithmes. Ces résultats sont les mêmes que ceux qui sont obtenus dans la figure 4.1 (page 72). Vu que *TSGA_Seq* et *GAHybrid_Seq* atteignent les mêmes gains énergétiques que *TSGA_Dist* et *GAHybrid_Dist* respectivement, nous avons calculé les temps d'exécution de chacun d'entre eux afin de les départager et cela pour les plus grands programmes tests (en termes de taille). Ainsi, la figure 4.9 donne les résultats moyens obtenus sur 30 exécutions de *TSGA_Seq* et de *TSGA_Dist*. *TSGA_Dist* nécessite de 82.61% (DjpegCE) jusqu'à 83.43% (ShaCE) moins de temps d'exécution que *TSGA_Seq*. Alors que la figure 4.10 donne les résultats moyens obtenus sur 30 exécutions de *GAHybrid_Seq* et de *GAHybrid_Dist*. *GAHybrid_Dist* nécessite de 78.54% (DjpegCE) jusqu'à 85.55% (AdpcmCE) moins de temps d'exécution que *GAHybrid_Seq*.

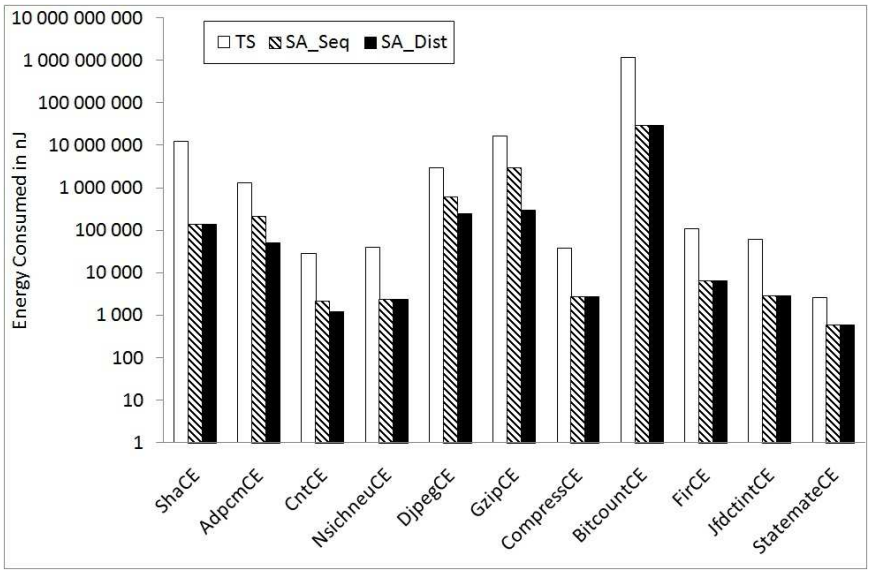

FIGURE 4.7 – Énergies consommées par *SA_Seq* et *SA_Dist*.

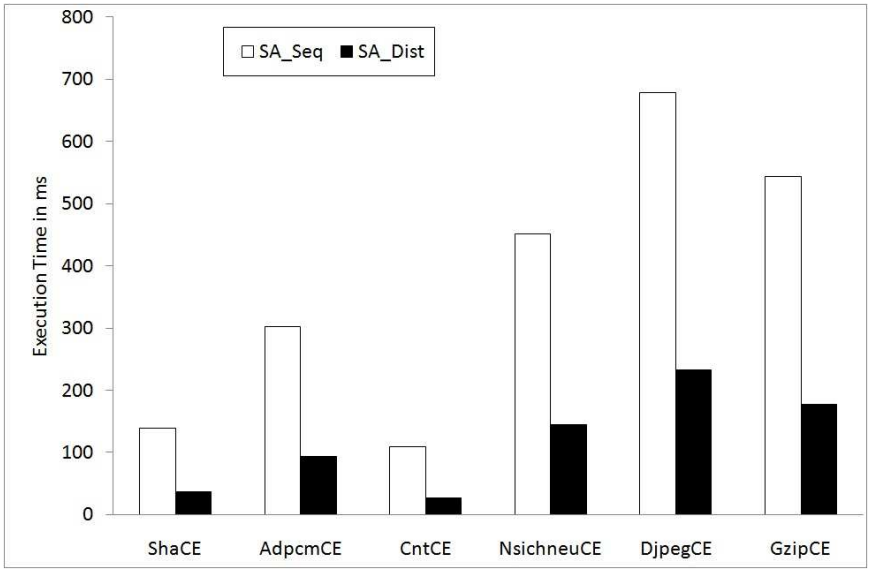

FIGURE 4.8 – Temps d'exécution utilisés par *SA_Seq* et *SA_Dist*.

Le tableau 4.7 donne le pourcentage en temps d'exécution nécessaire au déroulement de chaque algorithme distribué. Les valeurs écrites en caractères gras réfèrent à l'algorithme nécessitant le moins de temps d'exécution. Il en résulte que l'algorithme *TSGA_Dist* est le plus rapide. Cela diffère des résultats obtenus avec le tableau 4.1 où *GA_Hybrid* était, en général, le plus rapide.

4.4.3 GASA distribué

Ici, nous présentons une version distribuée de l'algorithme séquentiel hybride *GASA* décrit en sous-section 4.3.2. Cet algorithme est noté *GASA_Dist* et est proposé dans [Idrissi Aouad *et al.*, 2010b]. Ses

Algorithm 11 Algorithme coopératif et distribué *GAHybrid_Dist* s'exécutant sur chaque processeur.

Intitialize p_m, p_c, $p_i \in$]0,1] and $i \leftarrow 1$
$maxGen \leftarrow 1667$
Generate population P_0
Evaluate P_0 and find the best solution π^*
$\pi_{Elite} \leftarrow \pi^*$
$stop\_criterion \leftarrow$ false
while Not $stop\_criterion$ **do**
  $P_i \leftarrow \emptyset$
  for $j = 1$ to $PopSize/2$ **do**
    Select two parents p_1 and p_2 from P_{i-1}
    offspring $\leftarrow (p_1, p_2)$
    With probability p_c, perform offspring := crossover (p_1,p_2)
    With probability p_i, improve offspring by using TS (with $maxiter = 300$)
    Evaluate offspring and add it to P_i
  end for
  Add P_{i-1} to P_i
  if Migrate_condition **then**
    Receive $n = 5$ Individuals and add it to P_i
  end if
  Sort P_i
  Keep the $PopSize$ best solution in P_i
  Find the best solution π^* in P_i
  if π^* is better than π_{Elite} **then**
    $\pi_{Elite} \leftarrow \pi^*$
  end if
  if $fitness(\pi_{Elite}) = 0$ OR $i = maxGen$ **then**
    $stop\_criterion \leftarrow$ true
  end if
  Update (i)
end while

TABLE 4.7 – Pourcentage des temps d'exécution des algorithmes *TSGA_Dist* et *GAHybrid_Dist*.

Programmes tests	**TSGA_Dist**	**GAHybrid_Dist**
ShaCE	**97.40**	100
AdpcmCE	**97.89**	100
CntCE	**99.61**	100
NsichneuCE	**98.39**	100
DjpegCE	**98.56**	100
GzipCE	**97.50**	100

principes sont décrits dans l'algorithme 12.

Nous utilisons des sous-populations indépendantes d'individus avec leurs propres fonctions de performance, lesquelles évoluent séparément, sauf pour un échange de certains individus (migration). Un

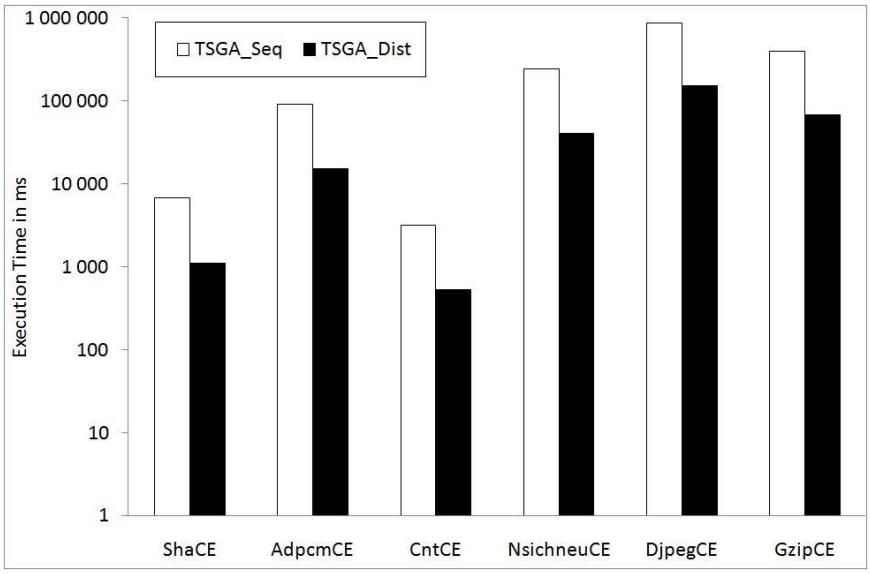

FIGURE 4.9 – Temps d'exécution utilisés par *TSGA_Seq* et *TSGA_Dist*.

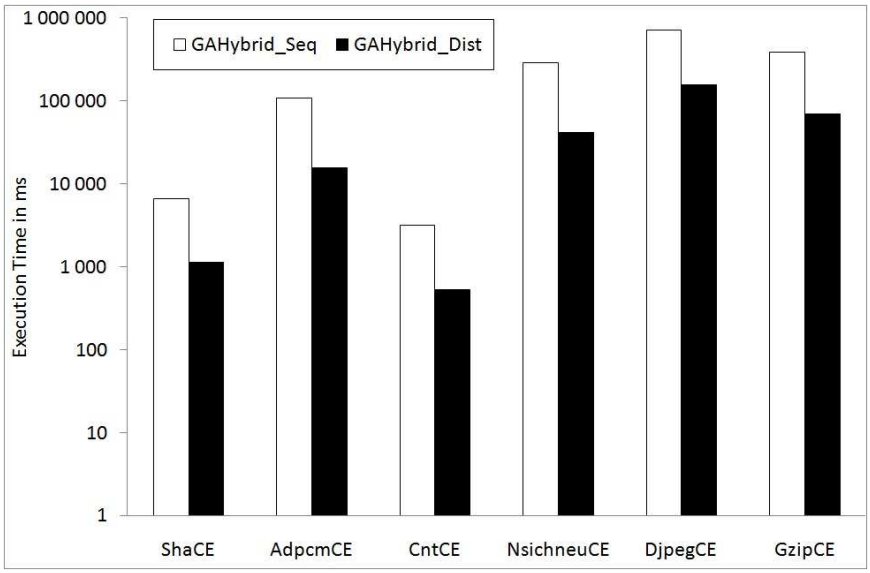

FIGURE 4.10 – Temps d'exécution utilisés par *GAHybrid_Seq* et *GAHybrid_Dist*.

ensemble de $m = 30$ individus est assigné à chacun des $P = 4$ processeurs (un *cluster* de 4 PCs) pour une taille totale de la population de $m \times P$. Cet ensemble assigné à chaque processeur est sa sous-population. La sous-population initiale est choisie aléatoirement pour chaque processeur. Chaque processeur, séparément et en parallèle, exécute *GASA_Seq* (algorithme 6) sur sa sous-population pour un certain nombre de générations. Par la suite, chaque sous-population échange un nombre spécifique d'individus (migrants) avec ses voisins ; $i.e.$ les migrants sont retirés d'une sous-population et sont ajoutés à une autre. De cette façon, la taille de la population reste la même après la migration. Le processus continue avec l'amélioration séparée de chaque sous-solution pour un nombre maximal de générations.

Algorithm 12 Algorithme coopératif et distribué *GASA_Dist* s'exécutant sur chaque processeur.

1: Intitialize p_m, p_c, $p_i \in]0, 1]$ and $i \leftarrow 1$
2: $maxGen \leftarrow 2500$
3: Generate population P_0
4: Evaluate P_0 and find the best solution π^*
5: $\pi_{Elite} \leftarrow \pi^*$
6: $stop\_criterion \leftarrow$ false
7: **while** Not $stop\_criterion$ **do**
8: $P_i \leftarrow \emptyset$
9: **for** j := 1 to $PopSize/2$ **do**
10: Select two parents p_1 and p_2 from P_{i-1}
11: offspring $\leftarrow (p_1, p_2)$
12: With probability p_c, perform offspring := crossover (p_1,p_2)
13: With probability p_m, mutate offspring
14: With probability p_i, improve offspring by using SA (with $maxiter = 200$)
15: Evaluate offspring and add it to P_i
16: **end for**
17: Add P_{i-1} to P_i
18: **if** Migrate_condtion **then**
19: Receive $n = 5$ individuals and add them to P_i
20: **end if**
21: Sort P_i
22: Keep the $PopSize$ best solution in P_i
23: Find the best solution π^* in P_i
24: **if** π^* is better than π_{Elite} **then**
25: $\pi_{Elite} \leftarrow \pi^*$
26: **end if**
27: **if** $fitness(\pi_{Elite}) = 0$ OR $i = maxGen$ **then**
28: $stop\_criterion \leftarrow$ true
29: **end if**
30: Update (i)
31: **end while**

À la fin, le meilleur individu existant constitue le résultat final.

Les deux versions séquentielle et distribuée produisent les mêmes résultats énergétiques pour chacune des deux heuristiques. Ces résultats sont les mêmes que ceux qui sont obtenus auparavant dans la figure 4.2. Vu que *GASA_Seq* atteint les mêmes gains énergétiques que *GASA_Dist*, nous avons calculé les temps d'exécution de chacun d'entre eux afin de les départager et cela pour les plus grands programmes tests (en termes de taille). Ainsi, la figure 4.11 donne les résultats moyens obtenus sur 30 exécutions de *GASA_Seq* et de *GASA_Dist*. Nous constatons que la version distribuée est plus rapide que la version séquentielle. En effet, *GASA_Dist* nécessite de 72.31% (GzipCE) jusqu'à 74.67% (CntCE) moins de temps d'exécution que *GASA_Seq*.

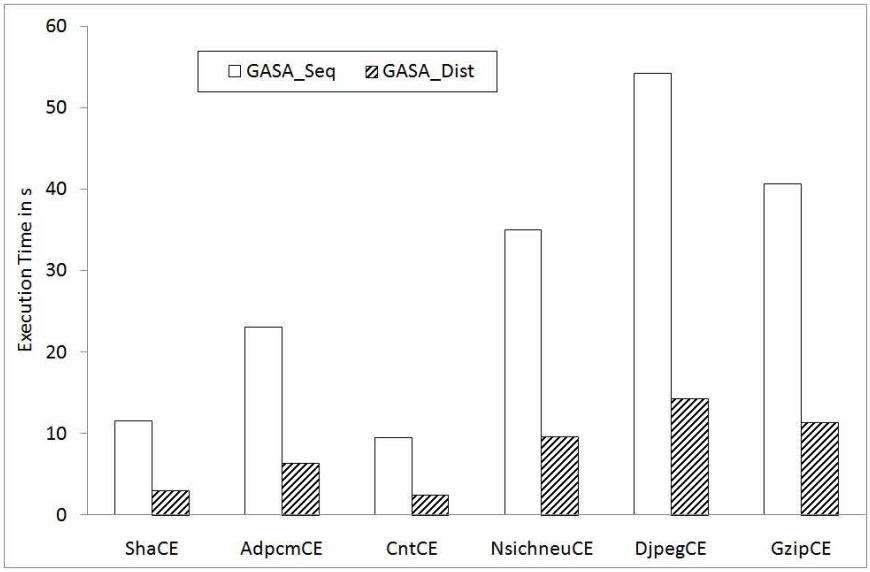

FIGURE 4.11 – Temps d'exécution utilisés par *GASA_Seq* et *GASA_Dist*.

4.4.4 PSOSA distribué

Dans cette sous-section, nous proposons une version distribuée de l'algorithme séquentiel *PSOSA* présenté en sous-section 4.3.3. Pour cet algorithme distribué, noté *PSOSA_Dist*, nous utilisons des sous-essaims indépendants de particules avec leurs propres fonctions de performance qui évoluent séparément, sauf pour un échange de certaines particules (migration). Un ensemble de $m = 30$ particules est assigné à chacun des $P = 4$ processeurs (un *cluster* de 4 PCs) pour une taille totale de la population de $m \times P$. Cet ensemble assigné à chaque processeur est son sous-essaim. Les sous-essaims initiaux sont choisis aléatoirement pour chaque processeur. Chaque processeur, séparément et en parallèle, exécute *PSOSA_Seq* (algorithme 7) sur son sous-essaim pour un certain nombre de générations. Par la suite, chaque sous-essaim échange [30] sa meilleure particule (migrant) avec ses voisins ; $i.e.$ le migrant est retiré d'un sous-essaim et est ajouté à un autre. De cette façon, la taille du sous-essaim reste la même après la migration (la pire particule est retirée). Le processus continue avec l'amélioration séparée de chaque solution courante pour un nombre maximum d'itérations. À la fin, la meilleure solution qui existe constitue le résultat final.

Les deux versions séquentielle et distribuée produisent les mêmes résultats énergétiques pour chacune des deux heuristiques. Ces résultats sont les mêmes que ceux qui sont obtenus dans la figure 4.3. Comme *PSOSA_Seq* et *PSOSA_Dist* donnent les mêmes résultats énergétiques, nous avons calculé les temps d'exécution de chacun d'entre eux afin de les départager et cela pour les plus grands programmes tests (en termes de taille). Ainsi, la figure 4.12 donne les résultats moyens obtenus sur 30 exécutions de *PSOSA_Seq* et de *PSOSA_Dist*. Nous constatons que la version distribuée est plus rapide que la version séquentielle. En effet, *PSOSA_Dist* nécessite de 73.16% (AdpcmCE) jusqu'à 84.65% (CntCE) moins de temps d'exécution que *PSOSA_Seq*.

30. $PSOSA\_Dist$ s'exécute sous le mode *asynchrone* : à la 100^{ieme} itération, chaque processeur envoie sa meilleure solution, continue d'améliorer son sous-essaim et vérifie s'il n'a pas reçu une solution de ses voisins.

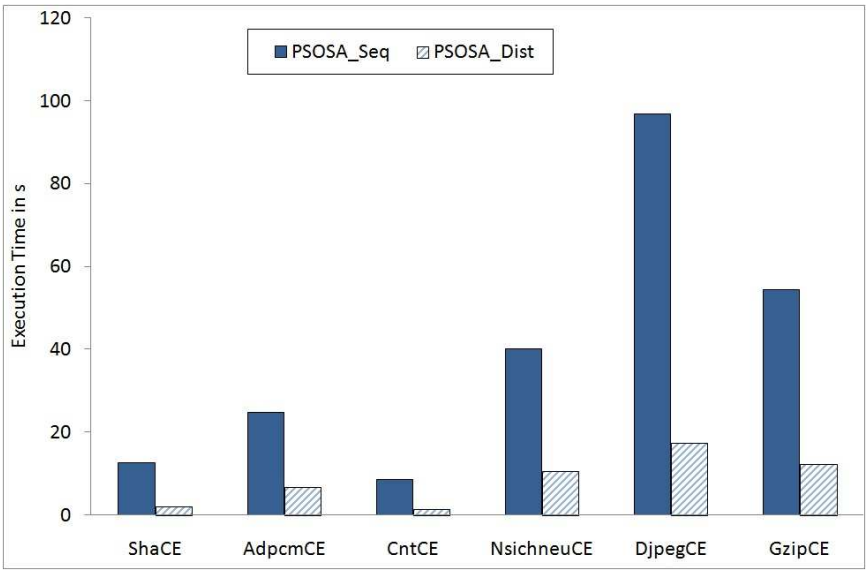

FIGURE 4.12 – Temps d'exécution utilisés par *PSOSA_Seq* et *PSOSA_Dist*.

4.5 Conclusion

En section 4.2, nous avons présenté deux algorithmes de combinaisons qui sont une forme simple d'hybridation. En revanche, en section 4.3, nous avons proposé quatre nouveaux algorithmes hybrides, à savoir : *GA_Hybrid*, *GASA*, *PSOSA* et *MPSOM*. Dans le cas de la réduction de la consommation d'énergie en mémoire dans les systèmes embarqués, ces algorithmes convergent tous vers la solution optimale, mais l'utilisation d'un algorithme génétique est plus rapide et suffit à résoudre ce problème comme nous avons pu le démontrer en section 4.3.5. De plus, nous avons expérimenté deux de nos algorithmes hybrides sur des fonctions tests mathématiques standards. Ainsi, dans la sous-section 4.3.3.2, nous avons comparé notre algorithme *PSOSA* aux algorithmes *QIPSO*, *ATREPSO* et *GMPSO* présentés dans [Pant *et al.*, 2008] et à l'algorithme *Tl-PSO* décrit dans [Nakano *et al.*, 2008] en considérant dix fonctions tests mathématiques connues. Il en est ressorti que notre algorithme *PSOSA* est plus performant. Alors que dans la sous-section 4.3.4.2, nous avons comparé notre algorithme *MPSOM* aux algorithmes *CLPSO*, *GPSO-J* et *ATM-PSO* décrits dans [Liang *et al.*, 2006; Yu-Xuan *et al.*, 2010; Krohling, 2005] en considérant sept des fonctions tests mathématiques. Là aussi, nous avons démontré l'efficacité de notre algorithme *MPSOM*.

De plus, en section 4.4 nous avons proposé des versions distribuées de nos algorithmes : *SA_Dist*, *GAHybrid_Dist*, *GASA_Dist* et *PSOSA_Dist*. Nous avons démontré que les versions distribuées sont plus rapides que les versions séquentielles. Les versions distribuées proposées sont également coopératives dans le sens où les processeurs échangent leurs meilleures solutions trouvées après un certain temps ou après un certain nombre d'itérations et en fonction de ces informations reçues, ils adaptent leurs parcours de recherche.

Conclusion générale et perspectives

Dans ce mémoire de doctorat, deux problématiques ont été traitées, à savoir : la réduction de la consommation d'énergie en mémoire dans les systèmes embarqués et l'optimisation de fonctions tests mathématiques. La prévalence des systèmes embarqués ainsi que leur très forte dépendance en énergie est un problème récurrent et primordial. La mémoire est considérée comme étant un principal consommateur d'énergie dans ce type de systèmes et les efforts se retournent vers elle. L'optimisation globale de fonctions tests mathématiques unimodales et multimodales est également un problème majeur. Ces fonctions possèdent des propriétés similaires aux problèmes du monde réel et fournissent une bonne base pour tester la crédibilité d'un algorithme d'optimisation, notamment du fait du nombre important d'optima locaux qu'elles contiennent. Le chapitre 1 décrit en détails ces deux problèmes d'optimisation.

Dans le chapitre 2, nous avons présenté différentes méthodes et solutions existantes afin de résoudre ces deux problèmes d'optimisation. Pour le problème de la réduction de la consommation d'énergie en mémoire, nous avons essayé de présenter un état de l'art assez large et général traitant de différents domaines s'appliquant à différents types de mémoire (mémoire cache, Scratch-Pad et mémoire principale) contrairement à ce qui s'est fait jusqu'à présent dans la littérature. En effet, sans avoir la prétention d'être exhaustif, ce travail offre une vue globale et précise de ce qui a été fait dans la plupart des domaines de la gestion mémoire : le choix optimal d'une allocation mémoire, la localité, le partitionnement mémoire, l'organisation de la mémoire en bancs ainsi que l'utilisation des modes de faible puissance. Certains problèmes, dans cette section, comme la réplication ou la consommation énergétique des systèmes multiprocesseurs n'ont pas été couverts en profondeur du fait que ce sont toujours des domaines de recherche active (ceci est également hors du contexte de cette thèse). Nous nous sommes également limités à l'étude des techniques d'optimisations logicielles économisant la consommation d'énergie dans tous les types de mémoire. Dans les différents travaux cités dans le chapitre 2, l'heuristique BEH est prédominante. C'est tout simplement une méthode de tri que nous essayons de dépasser. En ce qui concerne la problématique de l'optimisation globale de fonctions tests mathématiques, les solutions proposées dans la littérature se basent sur les principes des métaheuristiques simples ou hybridées entre elles. Ainsi, quelques travaux utilisent une seule métaheuristique à la fois parmi le recuit simulé, les algorithmes génétiques et l'optimisation par essaims particulaires et d'autres travaux proposent des méthodes hybrides basées sur différentes métaheuristiques.

La nouvelle approche dans ce mémoire, est la proposition d'algorithmes de type métaheuristiques pour la réduction de la consommation d'énergie en mémoire dans les systèmes embarqués. Ainsi, dans le chapitre 3, nous avons proposé différents algorithmes de type recherche avec tabous, recuit simulé et algorithmes génétiques dépassant tous la traditionnelle méthode de tri BEH. Toutefois, les algorithmes génétiques se sont révélés plus performants énergétiquement et font mieux que les autres méthodes. Effectivement, pour chacune de nos heuristiques basées sur le principe des algorithmes génétiques, le résultat moyen est égal au meilleur résultat ; cela veut dire que l'algorithme trouve à chaque fois le résultat optimal.

Afin d'essayer de réduire davantage la consommation d'énergie en mémoire dans les systèmes embarqués et de proposer des algorithmes plus robustes qui peuvent être appliqués au problème complexe de l'optimisation de fonctions tests multimodales, nous avons aussi proposé de nouveaux algorithmes hybrides tentant de résoudre nos deux problèmes d'optimisation. Ainsi, le chapitre 4 présente différents algorithmes hybrides : *TSGA*, *GATS*, *GA_Hybrid*, *GASA*, *PSOSA* et *MPSOM*. En ce qui concerne le problème de la réduction de la consommation d'énergie en mémoire, ces algorithmes convergent tous vers la solution optimale tout aussi bien que les algorithmes génétiques. Cependant, les algorithmes génétiques sont, de loin, les plus rapides comme c'est démontré dans ce chapitre. C'est pourquoi, dans le cadre de ce problème particulier, l'hybridation n'apporte pas d'améliorations supplémentaires (du point de vue énergie) et la proposition de notre algorithme de type génétique suffit amplement à résoudre ce problème. En revanche, dans le cas du problème de l'optimisation globale de fonctions tests mathématiques, les algorithmes hybrides proposés *PSOSA* et *MPSOM* sont plus à même d'être utilisés. Ainsi, en comparant notre algorithme *PSOSA* aux algorithmes *QIPSO*, *ATREPSO* et *GMPSO* présentés dans [Pant *et al.*, 2008] et à l'algorithme *Tl-PSO* décrit dans [Nakano *et al.*, 2008] sur dix fonctions tests mathématiques connues, nous avons prouvé que notre algorithme *PSOSA* est plus performant. Et en comparant notre algorithme *MPSOM* aux algorithmes *CLPSO*, *GPSO-J* et *ATM-PSO* décrits dans [Liang *et al.*, 2006; Yu-Xuan *et al.*, 2010; Krohling, 2005] en considérant sept des fonctions tests mathématiques, nous avons démontré l'efficacité de notre algorithme *MPSOM*. L'utilisation des fonctions tests mathématiques nous a permis d'avoir une bonne base pour tester la crédibilité de nos algorithmes d'optimisation. En effet, ces fonctions possèdent quelques propriétés similaires aux problèmes du monde réel et notamment elles contiennent beaucoup d'optima locaux dans leurs espaces de solution. La quantité d'optima locaux augmentant avec la complexité croissante des fonctions, $i.e.$ avec l'augmentation de la dimension, nous avons utilisé dans nos expérimentations ces fonctions en dimension élevée (20 ou 30 selon l'algorithme considéré). Il en ressort, que nos algorithmes *PSOSA* et *MPSOM* ont de meilleures performances par rapport aux algorithmes considérés en termes de précision, de vitesse de convergence, de stabilité et de robustesse.

De plus, nous avons également proposé des versions distribuées de nos algorithmes : *SA_Dist*, *GA-Hybrid_Dist*, *GASA_Dist* et *PSOSA_Dist*. Nous avons démontré que les versions distribuées sont plus rapides que leurs versions séquentielles respectives. Les versions distribuées proposées sont également coopératives dans le sens où les processeurs échangent leurs meilleures solutions trouvées après un certain temps ou après un certain nombre d'itérations et en fonction de ces informations reçues, ils adaptent leurs parcours de recherche.

De manière plus générale, la principale difficulté à laquelle est confronté un concepteur en présence d'un problème d'optimisation concret est celui du choix d'une méthode efficace, capable de produire une solution optimale (ou de qualité acceptable) au prix d'un temps de calcul raisonnable. Face au souci pragmatique de l'utilisateur, la théorie n'est pas encore d'un grand secours, car les théorèmes de convergence des métaheuristiques sont souvent inexistants ou applicables sous des hypothèses très restrictives. En outre, le réglage optimal des divers paramètres d'une métaheuristique, qui peut être préconisé par la théorie, est souvent inapplicable en pratique, car il induit un coût de calcul prohibitif. En conséquence, le choix d'une bonne méthode et le réglage des paramètres de celle-ci, font généralement appel, d'une part au savoir-faire et à l'expérience de l'utilisateur, plutôt qu'à l'application fidèle de règles bien établies. D'autre part, ils dépendent très fortement du problème d'optimisation considéré et diffèrent d'un problème à un autre.

Les algorithmes hybrides sont plus puissants et plus robustes que les algorithmes basés sur une seule

métaheuristique car ils combinent les avantages d'une méthode de recherche globale et d'une méthode de recherche locale. Préconiser telle hybridation plutôt que telle autre serait fortuit dans la mesure où l'efficacité de chaque méthode dépend du problème à résoudre, d'un côté et du bon réglage des différents paramètres, d'un autre côté. À titre d'exemple, dans le cas de l'économie d'énergie, tous les algorithmes hybrides atteignent les mêmes résultats ; nous pouvons en conclure qu'ils sont tous efficaces. Or, dans d'autres problèmes où la dimension du temps d'exécution est également prise en considération, les algorithmes hybrides *PSOSA* et *MPSOM* (plus performants dans le cas de l'optimisation globale de fonctions tests mathématiques) tireront leur épingle du jeu étant donné qu'ils apportent plus de rapidité et de robustesse que les autres algorithmes hybrides.

Dans le futur, nous envisageons d'appliquer les différentes méthodes hybrides proposées dans ce mémoire à d'autres problèmes d'ingénierie. En l'occurrence, l'optimisation d'une boîte à vitesse ou l'optimisation de circuits électroniques ainsi que d'autres problèmes multi-contraints relevant de l'ingénierie. En effet, ces algorithmes sont facilement adaptables à d'autres types de problèmes d'optimisation. D'autres directions peuvent également être suivies en comparant nos algorithmes proposés à de nouveaux algorithmes hybrides comme par exemple *PSO-GA*, *PSO-MDP*, *PSO-TS*, etc. La comparaison peut également être faite en utilisant d'autres fonctions tests mathématiques en considérant cette fois-ci une dimension supérieure à 30. Dans cette thèse, nous avons implémenté des versions distribuées en considérant 4 ou 6 processeurs. Il serait intéressant d'utiliser une machine du type *Grid5000*.

Annexe A

Stratégies de remplacement

Le chargement d'une ligne dans un cache se fait, en général, au détriment d'une ligne déjà présente. Un mécanisme matériel arbitre le remplacement des lignes dans un ensemble. Sur les caches associatifs, plusieurs stratégies de remplacement sont couramment utilisées :

- La stratégie de **choix aléatoire** (*Random*) : la ligne est choisie de manière aléatoire parmi les lignes possibles. Cette stratégie peu coûteuse en logique est peu fiable du point de vue des performances.
- La stratégie **LRU** (*Least Recently Used*) : la ligne choisie est celle qui a été la plus anciennement référencée. Cette stratégie donne, en général, de bons résultats, mais devient assez coûteuse lorsque l'associativité croît.
- La stratégie **FIFO** (*First In First Out*) : les lignes de la mémoire cache sont effacées dans l'ordre où elles sont arrivées dans la mémoire cache utilisant ainsi le principe de localité de la manière la plus simple possible. Cette stratégie est simple, mais le nombre de défauts de cache obtenus est entre 12 et 20% plus important que pour le LRU.

Ces stratégies sont assez simples, mais pas toujours satisfaisantes en terme de performances. De nombreuses autres stratégies ont été envisagées, mais le problème reste ouvert.

Annexe B

Fichier XML de description de la mémoire cache utilisé

```
< ?xml version="1.0" encoding="UTF-8" ?>
<cache-config>
    <icache ref="my_icache"/>
    <dcache ref="my_dcache"/>
    <cache id="my_icache" >
        <miss>
            <time>3</time>
            <power>5</power>
        </miss>
        <hit>
            <time>1</time>
            <power>2</power>
        </hit>
        <on_chip>true</on_chip>
        <block_bits>4</block_bits>
        <way_bits>1</way_bits>
        <row_bits>12</row_bits>
        <allocate>false</allocate>
        <replace>LRU</replace>
        <write_buffer_size>5</write_buffer_size>
        <read_port_size>5</read_port_size>
        <write_port_size>5</write_port_size>
        <write>WRITE_THROUGH</write>
    </cache>
    <cache id="my_dcache">
        <miss>
            <time>3</time>
            <power>5</power>
        </miss>
        <hit>
            <time>1</time>
            <power>2</power>
```

```
        </hit>
        <on_chip>true</on_chip>
        <block_bits>4</block_bits>
        <way_bits>0</way_bits>
        <row_bits>9</row_bits>
        <allocate>false</allocate>
        <replace>LRU</replace>
        <write_buffer_size>5</write_buffer_size>
        <read_port_size>5</read_port_size>
        <write_port_size>5</write_port_size>
        <write>WRITE_THROUGH</write>
    </cache>
</cache-config>
```

Annexe C

Fichier XML de description de la hiérarchie mémoire

```
< ?xml version="1.0" encoding="UTF-8" ?>
<memory>
    <banks>
        <bank>
            <name>non-volatile memory</name>
            <type>ROM</type>
            <address><offset>0</offset></address>
            <size>0x00080000</size>
            <writable>false</writable>
            <bus ref="local"/>
            <modes>
                <mode id="on">
                    <name>on</name>
                    <transitions>
                        <transition><dest ref="off"/><power>666</power></transition>
                        <transition><dest ref="on"/><power>111</power></transition>
                    </transitions>
                </mode>
                <mode id="off">
                    <name>off</name>
                    <transitions>
                        <transition><dest ref="off"/><power>111</power></transition>
                        <transition><dest ref="on"/><power>666</power></transition>
                    </transitions>
                </mode>
            </modes>
        </bank>
        <bank>
            <name>ON-CHIP STATIC RAM</name>
            <type>SPM</type>
            <address><offset>0x40000000</offset></address>
            <size>0x8000</size>
```

```
            <latency>10</latency>
            <block_bits>4</block_bits>
            <cached>false</cached>
            <writable>false</writable>
            <on_chip>true</on_chip>
            <port_num>1</port_num>
            <bus ref="Bus1"/>
        </bank>
    </banks>
    <buses>
        <bus id="Bus1">
            <name>Bus1</name>
            <type>LOCAL</type>
        </bus>
    </buses>
</memory>
```

Glossaire

Application-Specific Memory : ASM, petite mémoire placée *on-chip* à côté du processeur.

DMA : Direct Memory Access, procédé qui permet à un périphérique d'accéder directement à la mémoire vive sans occuper de ressources processeur (réduction de coût).

Fragmentation : le nombre total de cases mémoire disponibles est suffisant mais ces cases ne sont pas contiguës. Cela crée des cases vides non exploitables. C'est le problème de la fragmentation.

Localité spatiale : les éléments (données ou instructions) accédés de façon temporellement proche (successive) ont tendance à être proches géographiquement.

Localité temporelle : les éléments (données ou instructions) accédés récemment ont tendance à être à nouveau référencés dans un futur proche.

Mémoire Cache : petite zone mémoire SRAM rapide qui nécessite une circuiterie dédiée.

Mémoire Scratch-Pad : SPM, petite zone mémoire SRAM rapide qui ressemble à la mémoire cache mais qui est gérée au niveau logiciel, soit par le développeur soit par le compilateur.

Mapping : fonction de correspondance entre adresse physique (morceau de mémoire) et adresse logique (morceau de programme) (réalisée par le MMU).

Segment mémoire : morceau contigu de mémoire pouvant avoir toute taille (< 4 Go en adressage 32 bits) et pouvant commencer n'importe où en mémoire centrale.

Topologie en anneau : La topologie réseau en anneau se caractérise par une connexion circulaire de la ligne de communication. Le transfert des données entre 2 ordinateurs sur le réseau peut donc se faire suivant 2 directions.

Bibliographie

[Absar et Catthoor, 2005] M. J. Absar et F. Catthoor. Compiler-Based Approach for Exploiting Scratch-Pad in Presence of Irregular Array Access. In *DATE*, 2005.

[Absar et Catthoor, 2006] J. Absar et F. Catthoor. Analysis of Scratch-Pad and Data-Cache Performance Using Statistical Methods. In *ASP-DAC*, pages 820–825, 2006.

[Adiletta *et al.*, 2002] M. Adiletta, M. Rosenbluth, D. Bernstein, G. Wolrich, et H. Wilkinson. The Next Generation of Intel IXP Network Processors. *Intel Technology Journal*, 6(3) :6–18, August 2002.

[Al-khedhairi, 2008] A. Al-khedhairi. Simulated Annealing Metaheuristic for Solving P-Median Problem. In *Int. Journal Contemp. Math. Sciences*, volume 3(28), pages 1357–1365, 2008.

[Angiolini *et al.*, 2003] E. Angiolini, L. Benini, et A. Caprara. Polynomial-Time Algorithm for On-Chip Scratchpad Memory Partitioning. In *CASES*, 2003.

[Angiolini *et al.*, 2005] F. Angiolini, L. Benini, et A. Caprara. An Efficient Profile-Based Algorithm for Scratchpad Memory Partitioning. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 24(11) :1660–1676, 2005.

[Athavale *et al.*, 2001] R. Athavale, N. Vijaykrishnan, M. T. Kandemir, et M. J. Irwin. Influence of Array Allocation Mechanisms on Memory System Energy. In *IPDPS*, page 3, 2001.

[Atienza *et al.*, 2004] D. Atienza, S. Mamagkakis, F. Catthoor, J. M. Mendias, et D. Soudris. Dynamic Memory Management Design Methodology for Reduced Memory Footprint in Multimedia and Wireless Network Applications. In *DATE*, pages 532–537, 2004.

[Avissar *et al.*, 2001] O. Avissar, R. Barua, et D. Stewart. Heterogeneous Memory Management for Embedded Systems. In *CASES*, 2001.

[Avissar *et al.*, 2002] O. Avissar, R. Barua, et D. Stewart. An Optimal Memory Allocation Scheme for Scratch-pad-based Embedded Systems. *Transaction. on Embedded Computing Systems.*, 1(1) :6–26, 2002.

[Banakar *et al.*, 2002] R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, et P. Marwedel. Scratchpad Memory : Design Alternative for Cache On-Chip Memory in Embedded Systems. In *CODES*, pages 73–78, New York, NY, USA, 2002. ACM Press.

[Bäck, 1996] T. Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University, Press, 1996.

[Ben Fradj *et al.*, 2005] H. Ben Fradj, A. El Ouardighi, C. Belleudy, et M. Auguin. Energy Aware Memory Architecture Configuration. *SIGARCH Comput. Archit. News*, 33(3) :3–9, 2005.

[Benchmarks, 2010] Benchmarks, 2010. www.loria.fr/∼idrissma/benchs.zip.

[Benini *et al.*, 2000] L. Benini, A. Macii, E. Macii, et M. Poncino. Increasing Energy Efficiency of Embedded Systems by Application Specific Memory Hierarchy Generation. *IEEE Design and Test*, 17(2) :74–85, 2000.

[Benini et Micheli, 1999] L. Benini et G. De Micheli. System-Level Power Optimization : Techniques and Tools. In *ISLPED-99 :ACM/IEEE*, pages 288–293, San Diego, Calif., August 1999.

[Benini et Micheli, 2000] L. Benini et G. De Micheli. System-Level Power Optimization : Techniques and Tools. *IEEE Design and Test*, 17(2) :74–85, 2000.

[Brash, 2002] D. Brash. The ARM Architecture Version 6 (ARMv6). In *ARM Ltd.*, January 2002. White Paper.

[Cassé et Rochange, 2007] H. Cassé et C. Rochange. OTAWA, Open Tool for Adaptative WCET Analysis. In *DATE*, Nice, April 2007. Poster session.

[Chaojun et Qiu, 2006] D. Chaojun et Z. Qiu. Particle Swarm Optimization Algorithm Based on the Idea of Simulated Annealing. *IJCSNS International Journal of Computer Science and Network Security*, 6(10) :152– 157, October 2006.

[Clerc, 2005] M. Clerc. *L'optimisation par Essaims Particulaires*. Hermes Science Publications, 2005.

[Collette et Siarry, 2002] Y. Collette et P. Siarry. *Optimisation Multiobjectif*. Eyrolles, 2002.

[De La Luz *et al.*, 2000] V. De La Luz, M. T. Kandemir, N. Vijaykrishnan, et M. J. Irwin. Energy-Oriented Compiler Optimizations for Partitioned Memory Architectures. In *CASES*, pages 138–147, 2000.

[De La Luz *et al.*, 2002] V. M. De La Luz, M. T. Kandemir, N. Vijaykrishnan, M. J. Irwin, A. Sivasubramaniam, et I. Kolcu. Compiler-Directed Array Interleaving for Reducing Energy in Multi-Bank Memories. In *ASP-DAC*, pages 288–296, 2002.

[De La Luz *et al.*, 2006] V. De La Luz, M. T. Kandemir, et I. Kolcu. Reducing Memory Energy Consumption of Embedded Applications that Process Dynamically Allocated Data. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(9) :1855–1860, 2006.

[Dominguez *et al.*, 2005] A. Dominguez, S. Udayakumaran, et R. Barua. Heap Data Allocation to Scratch-Pad Memory in Embedded Systems. *ACM Transactions on Design Automation of Electronic Systems*, 5(2) :115–192, 2005.

[Dréo *et al.*, 2003] J. Dréo, A. Petrowski, E. Taillard, et P. Siarry. *Métaheuristiques pour l'Optimisation Difficile*. Eyrolles, 2003.

[Egger *et al.*, 2006] B. Egger, J. Lee, et H. Shin. Scratchpad Memory Management for Portable Systems with a Memory Management Unit. In *EMSOFT*, 2006.

[Engelbrecht, 2005] A. P. Engelbrecht. *Fundamentals of Computational Swarm Intelligence*. John Wiley & Sons, 2005.

[Fang *et al.*, 2007] L. Fang, P. Chen, et S. Liu. Particle Swarm Optimization with Simulated Annealing for TSP. In *Proceedings of the 6th WSEAS Int. Conf. on Artificial Intelligence, Knowledge Engineering and Data Bases*, pages 206–210, Stevens Point, Wisconsin, USA, 2007. World Scientific and Engineering Academy and Society (WSEAS).

[Feoktistov, 2006] V. Feoktistov. *Differential Evolution : In Search of Solutions (Springer Optimization and Its Applications)*, volume 5. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[Gendreau, 2003] M. Gendreau. *An Introduction to Tabu Search*, volume 57. Kluwer Academic Publishers, Boston, MA, 2003.

[Graybill et Melhem, 2002] R. Graybill et R. Melhem. *Power Aware Computing*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[Guthaus *et al.*, 2001] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, et R. B. Brown. MiBench : A free, Commercially Representative Embedded Benchmark Suite. In *WWC*

'01 : Proceedings of the Workload Characterization, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[H. Kellerer et Pisinger, 2004] U. Pferschy H. Kellerer et D. Pisinger. *Knapsack Problems*. Springer, Berlin, Germany, 2004.

[Hallnor et Reinhardt, 2000] G. Hallnor et S. K. Reinhardt. A fully associative software-managed cache design. In *27th International Symposium on Computer Architecture (ISCA)*, 2000.

[Hiser et Davidson, 2004] J. D. Hiser et J. W. Davidson. EMBARC : An Efficient Memory Bank Assignment Algorithm for Retargetable Compilers. In *ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 182–191. ACM Press, 2004.

[Idoumghar *et al.*,] L. Idoumghar, M. Idrissi Aouad, M. Melkemi, et R. Schott. Hybrid PSO-SA Type Algorithms for Multi-Modal Function Optimization and Reducing Energy Consumption in Embedded Systems. *Applied Computational Intelligence and Soft Computing*, 2011. Article ID 138078, 12 pages.

[Idoumghar *et al.*, 2009] L. Idoumghar, M. Melkemi, et R. Schott. A Novel Hybrid Evolutionary Algorithm for Multi-Modal Function Optimization and Engineering Applications. In *Proceedings of ASC'2009 : Artificial Intelligence and Soft Computing - 13th IASTED International Conference on Artificial Intelligence and Soft Computing*, volume 683, pages 87–93, Palma de Mallorca, Spain, September 2009. Acta Press.

[Idoumghar *et al.*, 2010] L. Idoumghar, M. Idrissi Aouad, M. Melkemi, et R. Schott. Metropolis Particle Swarm Optimization Algorithm with Mutation Operator For Global Optimization Problems. In *Proceedings of IEEE-ICTAI'2010 : 22th International Conference on Tools with Articial Intelligence*, pages 35–42, Arras, France, October 2010.

[Idrissi Aouad *et al.*, 2010a] M. Idrissi Aouad, L. Idoumghar, R. Schott, et O. Zendra. Reduction of Energy Consumption in Embedded Systems : A Hybrid Evolutionary Algorithm. In *Proceeding of META'10 : 3rd International Conference on Metaheuristics and Nature Inspired Computing*, volume 95, Djerba, Tunisia, October 2010.

[Idrissi Aouad *et al.*, 2010b] M. Idrissi Aouad, L. Idoumghar, R. Schott, et O. Zendra. Sequential and Distributed Hybrid GA-SA Algorithms for Energy Optimization in Embedded Systems. In *IADIS'2010 : Proceedings of the International Conference Applied Computing 2010*, pages 167–174, Timisoara, Romania, October 2010.

[Idrissi Aouad *et al.*, 2010c] M. Idrissi Aouad, L. Idoumghar, R. Schott, et O. Zendra. Sequential and Distributed SA-Type Algorithms for Energy Optimization in Embedded Systems. In *IEEE-CiSE'2010 : Proceedings of International Conference on Computational Intelligence and Software Engineering*, volume 1, Wuhan, China, December 2010.

[Idrissi Aouad *et al.*, 2010d] M. Idrissi Aouad, R. Schott, et O. Zendra. A Tabu Search Heuristic for Scratch-Pad Memory Management. In *ICSET'2010 : Proceedings of International Conference on Software Engineering and Technology*, volume 64, pages 386–390, Rome, Italy, April 2010. WASET Publisher.

[Idrissi Aouad *et al.*, 2010e] M. Idrissi Aouad, R. Schott, et O. Zendra. Genetic Heuristics for Reducing Memory Energy Consumption in Embedded Systems. In *ICSOFT'2010 : Proceedings of International Conference on Software and Data Technologies*, volume 2, pages 394–402, Athens, Greece, July 2010.

[Idrissi Aouad *et al.*, 2010f] M. Idrissi Aouad, R. Schott, et O. Zendra. Hybrid Heuristics for Optimizing Energy Consumption in Embedded Systems. In *ISCIS'2010 : Proceedings of the 25th International Symposium on Computer and Information Sciences*, volume 62, pages 409–414, London, UK, September 2010. Springer.

[Idrissi Aouad et Zendra, 2007] M. Idrissi Aouad et O. Zendra. A Survey of Scratch-Pad Memory Management Techniques for low-power and -energy. In *2nd ECOOP Workshop on ICOOOLPS'2007*, pages 31–38, Berlin, Germany, July 2007.

[Idrissi Aouad et Zendra, 2008] M. Idrissi Aouad et O. Zendra. Outils de Caractérisation du Comportement Mémoire et d'Estimation de la Consommation Energétique. Technical Report RT-0350, INRIA, 2008.

[Issenin *et al.*, 2004] I. Issenin, E. Brockmeyer, M. Miranda, et N. Dutt. Data Reuse Analysis Technique for Software-Controlled Memory Hierarchies. In *DATE*, pages 202–207, 2004.

[ITRS, 2007] ITRS. System Drivers, 2007. http ://www.itrs.net/Links/2007ITRS/2007_Chapters/2007_SystemDrivers.pdf.

[Kandemir *et al.*, 2001] M. T. Kandemir, I. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, et A. Parikh. Dynamic Management of Scratch-pad Memory Space. In *Pmc. DAC*, 2001.

[Kandemir *et al.*, 2002] M. T. Kandemir, J. Ramanujam, et A. N. Choudhary. Exploiting shared scratch pad memory space in embedded multiprocessor systems. In *DAC*, pages 219–224, 2002.

[Kandemir *et al.*, 2004] M. T. Kandemir, M. J. Irwin, G. Chen, et I. Kolcu. Banked scratch-pad memory management for reducing leakage energy consumption. In *ICCAD*, pages 120–124, 2004.

[Kandemir *et al.*, 2005] M. T. Kandemir, M. J. Irwin, G. Chen, et I. Kolcu. Compiler-guided leakage optimization for banked scratch-pad memories. *IEEE Trans. VLSI Syst.*, 13(10) :1136–1146, 2005.

[Kennedy et Eberhart, 1995] J. Kennedy et R. C. Eberhart. Particle Swarm Optimization. In *IEEE International Conference on Neural Networks*, pages 1942–1948. IEEE Computer Society, 1995.

[Kennedy et Russell, 2001] J. Kennedy et C. Eberhart. Russell. Swarm Intelligence. In *Morgan Kaufmann Academic Press*, 2001.

[Kim *et al.*, 2001] S. Kim, N. Vijaykrishnan, M. T. Kandemir, A. Sivasubramaniam, M. J. Irwin, et E. Geethanjali. Power-Aware Partitioned Cache Architectures. In *ISLPED*, pages 64–67, 2001.

[Kirkpatrick *et al.*, 1983] S. Kirkpatrick, C. D. Gelatt, Jr., et M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220 :671–680, 1983.

[Krohling, 2005] R. A. Krohling. Gaussian Particle Swarm with Jumps. In *Proceedings of the IEEE Comgress on Evolutionary Computat.*, pages 1226–1231, 2005.

[Lazarova, 2008] M. Lazarova. Parallel Simulated Annealing for Solving the Room Assignment Problem on Shared and Distributed Memory Platforms. In *CompSysTech '08*, pages II.13–1, NY, USA, 2008. ACM.

[LCTES, 2003] LCTES. Compilation Challenges for Network Processors. In *Compilers and Tools for Embedded Systems*. Industrial Panel, ACM Conference on Languages, June 2003.

[Li et Silva, 2008] H. Li et D. Landa Silva. Evolutionary Multi-objective Simulated Annealing with Adaptive and Competitive Search Direction. In *IEEE Congress on Evolutionary Computation*, pages 3311–3318, 2008.

[Liang *et al.*, 2006] J. J. Liang, A. K. Qin, P. N. Suganthan, et S. Baskar. Comprehensive Learning Particle Swarm Optimizer for Global Optimization of Multimodal Functions. In *IEEE Trans. Evol. Comput.*, volume 10, pages 281–295, 2006.

[Locatelli, 2002] M. Locatelli. Simulated Annealing Algorithms for Continuous Global Optimization. In *In : P.M. Pardalos, H.E. Romeijn (Eds.) Handbook of Global Optimization*, volume 2, pages 179–230. Kluwer Academic Pub., 2002.

[Mamagkakis *et al.*, 2006] S. Mamagkakis, D. Atienza, C. Poucet, F. Catthoor, et D. Soudris. Energy-Efficient Dynamic Memory Allocators at the Middleware Level of Embedded Systems. In *EMSOFT*, pages 215–222, 2006.

[Mamidipaka et Dutt, 2004] M. Mamidipaka et N. Dutt. eCACTI : An Enhanced Power Estimation Model for On-Chip Caches. In *Technical Report TR-04-28, CECS, UCI*, 2004.

[Marco et Stützle, 2008] A. M. Marco et T. Stützle. Convergence behavior of the fully informed particle swarm optimization algorithm. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 71–78, 2008.

[Matsui et Yamada, 2008] S. Matsui et S. Yamada. Optimizing Hierarchical Menus by Genetic Algorithm and Simulated Annealing. In *GECCO '08 : Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1587–1594, NY, USA, 2008. ACM.

[Molga et Smutnicki, 2005] M. Molga et C. Smutnicki. Test Functions for Optimization Needs. 2005. Available at http ://www.zsd.ict.pwr.wroc.pl/files/docs/functions.pdf.

[Montes de Oca et Stützle, 2008] M. A. Montes de Oca et T. Stützle. Convergence Behavior of the Fully Informed Particle Swarm Optimization Algorithm. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, GECCO '08, pages 71–78, New York, NY, USA, 2008. ACM.

[Nakano *et al.*, 2008] S. Nakano, A. Ishigame, et K. Yasuda. Consideration of Particle Swarm Optimization Combined with Tabu Search. In *Special Issue on "The Electronics, Information and Systems Conference Electronics, Information and Systems Society, I.E.E. of Japan"*, volume 128-C, pages 1162–1167, July 2008.

[Nguyen *et al.*, 2005] N. Nguyen, A. Dominguez, et R. Barua. Memory Allocation for Embedded Systems with a Compile-Time-Unknown Scratch-Pad Size. In *CASES*, 2005.

[Nguyen *et al.*, 2007a] Nghi Nguyen, Angel Dominguez, et Rajeev Barua. Memory Allocation for Embedded Systems with a Compile-Time-Unknown Scratch-Pad Size. In *To appear in the ACM Transactions on Embedded Computing Systems (TECS)*, 2007.

[Nguyen *et al.*, 2007b] Nghi Nguyen, Angel Dominguez, et Rajeev Barua. Scratch-Pad Memory Allocation without Compiler Support for Java Applications. In *CASES*, pages 85–94, 2007.

[Ortiz-Boyer *et al.*, 2005] D. Ortiz-Boyer, C. Herbaś-Martínez, et N. Garciá-Pedrajas. CIXL2 - A crossover operator for evolutionary algorithms based on population features. In *Journal of Artifical Intelligence Research*, volume 24, pages 1–48, 2005.

[Ozturk et Kandemir, 2005a] O. Ozturk et M. T. Kandemir. Integer linear programming based energy optimization for banked DRAMs. In *ACM Great Lakes Symposium on VLSI*, pages 92–95, 2005.

[Ozturk et Kandemir, 2005b] O. Ozturk et M. T. Kandemir. Nonuniform Banking for Reducing Memory Energy Consumption. In *DATE*, pages 814–819, 2005.

[Ozturk et Kandemir, 2006] O. Ozturk et M. T. Kandemir. Data Replication in Banked DRAMs for Reducing Energy Consumption. In *ISQED*, pages 551–556, 2006.

[Panda *et al.*, 1997] P. R. Panda, N. Dutt, et A. Nicolau. Efficient Utilization of Scratch-Pad Memory in Embedded Processor Applications. In *Proceedings of the 1997 European conference on Design and Test*, EDTC '97, page 7, Washington, DC, USA, 1997. IEEE Computer Society.

[Pant *et al.*, 2008] M. Pant, R. Thangaraj, et A. Abraham. Particle Swarm Based Meta-Heuristics for Function Optimization and Engineering Applications. In 7^{th} *Conf. Computer Information Systems and Industrial Management Applications*, volume 7, pages 84–90, Washington, DC, USA, 2008. IEEE Computer Society.

[Poletti *et al.*, 2004] F. Poletti, P. Marchal, D. Atienza, L. Benini, F. Catthoor, et J. M. Mendias. An Integrated Hardware/Software Approach for Run-Time Scratchpad Management. In *DAC*, pages 238–243, 2004.

[Poli *et al.*, 2007] R. Poli, J. Kennedy, et T. Blackwell. Particle swarm optimization. An overview. In *Swarm Intelligence*, volume 1, pages 33–57, 2007.

[Premalatha et Natarajan, 2010] K. Premalatha et A. M. Natarajan. Combined Heuristic Optimization Techniques for Global Minimization. *Journal of Advances in Soft Computing and Its Applications*, 2(1) :85–99, March 2010.

[Ratnaweera *et al.*, 2004] A. Ratnaweera, S. K. Halgamuge, et H. C. Watson. Self-Organizing Hierarchical Particle Swarm Optimizer With Time-Varying Acceleration Coefficients. *IEEE Trans. Evolutionary Computation*, 8(3) :240–255, 2004.

[Shi et Eberhart, 1998] Y. Shi et R. C. Eberhart. A Modified Particle Swarm Optimizer. In *IEEE Congress on Evolutionary Computation (CEC 1998)*, pages 69–73. IEEE Computer Society, 1998.

[Shi et Eberhart, 1999] Y. Shi et R. C. Eberhart. Empirical Study of Particle Swarm Optimization. In *IEEE Congress on Evolutionary Computation (CEC 1999)*, volume 3, pages 1945–1950. IEEE, 1999.

[Shiue et Chakrabarti, 1999] W-T. Shiue et C. Chakrabarti. Memory Exploration for Low Power, Embedded Systems. In *DAC*, 1999.

[Shrivastava *et al.*, 2005] A. Shrivastava, I.Issenin, et N. Dutt. Compilation Techniques for Energy Reduction in Horizontally Partitioned Cache Architectures. In *CASES*, pages 90–96, 2005.

[Sivanandam et Deepa, 2007] S. N. Sivanandam et S. N. Deepa. *Introduction to Genetic Algorithms*. Springer Publishing Company, Incorporated, 2007.

[Sjödin *et al.*, 1998] J. Sjödin, B. Fröderberg, et T. Lindgren. Allocation of Global Data Objects in On-Chip RAM. In *Workshop on Compiler and Architectural Support for Embedded Computer Systems*. ACM, December 1998.

[Steinke *et al.*, 2002] S. Steinke, L. Wehmeyer, B. Lee, et P. Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *DATE*, page 409. IEEE Computer Society, 2002.

[Suganthan *et al.*, 2005] P. N. Suganthan, N. Hansen, J. J. Liang, K. Deb, Y.-P. Chen, A. Auger, et S. Tiwari. Problem definitions and evaluation criteria for the CEC 2005 special session on real-parameter optimization. In *Technical Report 2005005 Nanyang Technological University, Singapore and IIT Kanpur, India*, May 2005.

[Sydow *et al.*, 2009] M. Sydow, F. Bonchi, C. Castillo, et D. Donato. Optimising Topical Query Decomposition. In *WSCD '09 : Proceedings of the 2009 workshop on Web Search Click Data*, pages 43–47, NY, USA, 2009. ACM.

[Tanenbaum, 2005] A. Tanenbaum. *Architecture de l'Ordinateur. 5e édition*. November 2005.

[Tarjan *et al.*, 2006] D. Tarjan, S. Thoziyoor, et N. P. Jouppi. *CACTI 4.0.* Technical Report HPL-2006-86. HP Laboratories Palo Alto, June 2006.

[Truong *et al.*, 1998] D. N. Truong, F. Bodin, et A. Seznec. Improving Cache Behavior of Dynamically Allocated Data Structures. In *IEEE PACT*, pages 322–329, 1998.

[Udayakumaran *et al.*, 2002] S. Udayakumaran, B. Narahari, et R. Simha. Application Specific Memory Partitioning for Low Power. In *ACM COLP 2002 (Compiler and Operating Systems for Low Power)*. ACM Press, 2002.

[Udayakumaran *et al.*, 2006] S. Udayakumaran, A. Dominguez, et R. Barua. Dynamic Allocation for Scratch-Pad Memory Using Compile-Time Decisions. *Embedded Comput. Syst*, 5(2) :472–511, 2006.

[Udayakumaran et Barua, 2003] S. Udayakumaran et R. Barua. Compiler-Decided Dynamic Memory Allocation for Scratch-Pad Based Embedded Systems. In *CASES*, pages 276–286. ACM Press, 2003.

[Udayakumaran et Barua, 2006] S. Udayakumaran et R. Barua. An Integrated Scratch-Pad Allocator for Affine and Non-Affine Code. In *DATE*, pages 925–930, 2006.

[Verma *et al.*, 2004] M. Verma, L. Wehmeyer, et P. Marwedel. Dynamic Overlay of Scratchpad Memory for Energy Minimization. In *CODES+ISSS*, 2004.

[Wang *et al.*, 2007] Y. Wang, Z. Zhao, et R. Ren. Hybrid Particle Swarm Optimizer with Tabu Strategy for Global Numerical Optimization. In *IEEE Congress on Evolutionary Computation*, pages 2310–2316, 2007.

[Wang et Li, 2004] X.H. Wang et J.J. Li. Hybrid Particle Swarm Optimization with Simulated Annealing. In *Proceedings of the Third International Conference on Machine Learning and Cybernetics*, volume 4, pages 2402–2405, 2004.

[Wehmeyer *et al.*, 2004] L. Wehmeyer, U. Helmig, et P. Marwedel. Compiler-Optimized Usage of Partitioned Memories. In *Proceedings of the 3rd workshop on Memory performance issues : in conjunction with the 31st international symposium on computer architecture*, WMPI '04, pages 114–120, New York, NY, USA, 2004. ACM.

[Weise, 2009] T. Weise. *Global Optimization Algorithms – Theory and Application*. it-weise.de (self-published) : Germany, 2009.

[Wilton et Jouppi, 1996] S.J.E. Wilton et N.P. Jouppi. Cacti : An Enhanced Cache Access and Cycle Time Model. *IEEE Journal of Solid-State Circuits*, 31 :677–688, 1996.

[Xia et Wu, 2006] W.J. Xia et Z.M. Wu. A Hybrid Particle Swarm Optimization Approach for the Job-shop Scheduling Problem. *International Journal of Advanced Manufacturing Technology*, 29(3-4) :360–366, 2006.

[Yu-Xuan *et al.*, 2010] W. Yu-Xuan, X. Qiao-Liang, et Z. Zhen-Dong. Particle Swarm Optimizer with Adaptive Tabu and Mutation : A Unified Framework for Efficient Mutation Operators. 5 :1 :1–1 :27, February 2010.

[Zendra, 2006] O. Zendra. Memory and Compiler Optimizations for Low-Power and -Energy. In *ICOOOLPS*, 2006.

[Zhang *et al.*, 2003] W. Zhang, M. Karaköy, M. T. Kandemir, et G. Chen. A Compiler Approach for Reducing Data Cache Energy. In *ICS*, pages 76–85, 2003.

[Zheng et Kiyooka, 1999] Y. Zheng et S. Kiyooka. Genetic Algorithm Applications, 1999. www.me.uvic.ca/∼zdong/courses/mech620/GA_App.PDF.

Résumé

La mémoire est considérée comme étant gloutonne en consommation d'énergie, un problème sensible, particulièrement dans les systèmes embarqués. L'optimisation globale de fonctions multimodales est également un problème délicat à résoudre du fait de la grande quantité d'optima locaux de ces fonctions. Dans ce mémoire, je présente différents nouveaux algorithmes hybrides et distribués afin de résoudre ces deux problèmes d'optimisation. Ces algorithmes sont comparés avec les méthodes classiques utilisées dans la littérature et les résultats obtenus sont encourageants. En effet, ces résultats montrent une réduction de la consommation d'énergie en mémoire d'environ 76% jusqu'à plus de 98% sur nos programmes tests, d'une part. D'autre part, dans le cas de l'optimisation globale de fonctions multimodales, nos algorithmes hybrides convergent plus souvent vers la solution optimale globale. Des versions distribuées et coopératives de ces nouveaux algorithmes hybrides sont également proposées. Elles sont, par ailleurs, plus rapides que leurs versions séquentielles respectives.

Mots-clés : algorithmes distribués, algorithmes hybrides, basse énergie, fonctions multimodales, gestion mémoire, métaheuristiques, optimisations, systèmes embarqués autonomes.

Abstract

Memory is considered to be greedy in energy consumption, a sensitive issue, especially in embedded systems. The global optimization of multimodal functions is also a difficult problem because of the large number of local optima of these functions. In this thesis report, I present various new hybrid and distributed algorithms to solve these two optimization problems. These algorithms are compared with conventional methods used in the literature and the results obtained are encouraging. Indeed, these results show a reduction in memory energy consumption by about 76% to more than 98% on our benchmarks on one hand. On the other hand, in the case of global optimization of multimodal functions, our hybrid algorithms converge more often to the global optimum solution. Distributed and cooperative versions of these new hybrid algorithms are also proposed. They are more faster than their respective sequential versions.

Keywords: autonomus embedded systems, distributed algorithms, hybrid algorithms, low-energy, memory management, metaheuristics, multimodal functions, optimizations.