Ordonnancement de liste dans les systèmes embarqués sous contrainte de mémoire

Paul-Antoine Arras, Didier Fuin, Emmanuel Jeannot, Arthur Stoutchinin, Samuel Thibault

ComPAS'2013 : RenPar'21 / SympA'15/ CFSE'9
Grenoble, France, du 16 au 18 janvier 2013

Ordonnancement de liste dans les systèmes embarqués sous contrainte de mémoire

Paul-Antoine ARRAS*,†, Didier FUIN†, Emmanuel JEANNOT*, Arthur STOUTCHININ† & Samuel THIBAULT*

*Inria Bordeaux Sud-Ouest
200 avenue de la Vieille Tour
33405 Talence Cedex - France
prenom.nom@inria.fr

†STMicroelectronics
12 rue Jules Horowitz - B.P. 217
38019 Grenoble - France
prenom.nom@st.com

Résumé
Le décodage vidéo et le traitement d'image dans les systèmes embarqués sont soumis à de fortes contraintes de ressources, notamment en termes de mémoire. Les heuristiques d'ordonnancement de liste à priorités statiques (HEFT, SDC, etc.) étant souvent citées pour à la fois leurs bonnes performances et leur faible complexité, nous proposons une méthode visant à y introduire la notion de mémoire. De plus, nous montrons que par un ajustement adéquat des priorités des tâches et un recours judicieux à l'insertion, il est possible d'obtenir des accélérations au-delà de 20 %. Enfin, nous proposons une méthodologie visant à évaluer l'opportunité de recourir à un ordonnancement dynamique dans ce cadre.

Mots-clés : graphes de tâches, ordonnancement, mémoire, systèmes sur puces, décodage vidéo

1. Introduction

À l'heure où la convergence des terminaux numériques repousse les limites de l'intégration de fonctionnalités multimédias autrefois dévolues à des appareils *ad hoc*, il n'est pas rare de rencontrer des téléphones mobiles capables de reproduire des contenus vidéo en diffusion continue émanant d'un réseau sans fil. Aussi cela laisserait-il à penser que l'opération consistant à décoder un flux d'images serait devenue une tâche anodine sujette à un traitement séquentiel à la portée de n'importe quel processeur généraliste embarqué bon marché ; il n'en est rien. En réalité, les algorithmes d'encodage et de décodage (codecs) récents destinés au grand public, tels que le très répandu H.264 [18] et son successeur HEVC [15], sont d'une complexité [12] qui rend inconcevable l'utilisation isolée de ce type d'unité de calcul, sauf à vouloir se satisfaire d'une restitution de piètre qualité.
Dans ces conditions, la seule solution viable consiste à avoir recours à un traitement parallèle avec notamment des accélérateurs matériels dont chacun est spécialisé dans l'exécution d'une tâche bien déterminée. Se pose alors la question de l'ordonnancement : sur quel processeur et dans quel ordre exécuter chaque tâche ? Pour y répondre, de nombreux algorithmes existent déjà, mais ils sont en grande majorité destinés au calcul haute performance sur des (ensembles de) stations où la mémoire n'est pas une contrainte, contrairement au monde de l'embarqué dans lequel réduire l'empreinte d'un programme est une préoccupation majeure. Néanmoins, les solutions traditionnellement issues de l'informatique haut niveau, telles que les heuristiques d'ordonnancement dites « de liste », démontrent pour certaines de bonnes performances alliées à une faible complexité, qui se prêtent donc bien à des systèmes légers à l'instar de ceux visés par notre étude.
C'est pourquoi nous proposons une adaptation d'algorithmes d'ordonnancement de liste en y introduisant la notion de mémoire en vue de procéder au décodage de flux vidéo sur des systèmes embarqués. La suite de l'article développe comment y parvenir : la section 2 décrit le modèle de calcul utilisé ; la section 3 établit le contexte de travail et énumère les travaux antérieurs sur lesquels se fonde cette étude ; la section 4 décrit les changements du modèle pour prendre en compte les contraintes mémoire ;

la section 5 expose le cœur de la contribution, à savoir la méthode permettant de prendre en compte la mémoire dans le processus d'ordonnancement ; la section 6 illustre la technique développée par des simulations mettant en œuvre un cas d'application concret ; la section 7 relate les autres travaux ayant trait au problème étudié mais n'ayant pu être exploités ici ; enfin la section 8 résume et conclut cet article.

2. Définitions et modèles

2.1. Modèle de calcul

L'application parallèle à ordonnancer comprend un ensemble de tâches interdépendantes, ce qui se modélise par un graphe orienté acyclique (*directed acyclic graph*, DAG) dont les sommets représentent les tâches à exécuter et les arêtes les dépendances. Celles-ci sont souvent dues à des transferts de données, mais peuvent aussi servir à synchroniser les tâches entre elles.Les durées des tâches et des communications dépendent de la ressource qui les effectue. Elles sont spécifiées par un tableau à double entrée (modèle *unrelated*). La consommation mémoire de certaines tâches sera expliquée et modélisée à la section 4. Le modèle de calcul (*computation model*) du graphe de tâches est appelé *macro dataflow*. Chaque tâche doit d'abord recevoir les données dont elle a besoin, puis elle effectue son calcul sans interruption et enfin envoie ses données à tous ses successeurs.

2.2. Modèle de la plateforme cible

Dans le contexte du décodage vidéo et du traitement d'image, la solution homogène sur processeurs généralistes se révèle coûteuse et peu performante ; quant à l'implémentation sous forme de circuit intégré spécialisé (*application-specific integrated circuit*, ASIC), elle pèche par manque de flexibilité. La solution intermédiaire communément admise [8, 17, 9] consiste en une plateforme hétérogène de type système-sur-puce (*system on chip*, SoC) comportant à la fois un ou plusieurs processeurs généralistes et des accélérateurs matériels spécialisés ; c'est dans ce cadre-là que nous nous plaçons pour cette étude.

Dans le détail, les hypothèses suivantes sont faites :

- les processeurs sont pour certains généralistes, pour d'autres des accélérateurs qui ne peuvent exécuter qu'un seul type de tâche ;
- tous les processeurs sont interconnectés et les coûts de communication entre chaque paire sont modélisés par une bande passante et une latence constantes ;
- les communications et les calculs peuvent s'effectuer en parallèle ;
- les communications interprocesseurs se font sans contention et les communications intraprocesseurs ont un coût nul.

Ces hypothèses sont celles classiquement retenues pour les heuristiques principalement destinées à l'informatique telles que HEFT (cf. section 3.2). Implicitement, elles supposent également que de la mémoire est présente en quantité suffisante, de telle sorte qu'il ne soit pas nécessaire d'en tenir compte dans le processus d'ordonnancement. Dans un contexte embarqué où la mémoire est une ressource critique, il est indéniable que cette dernière hypothèse est caduque. D'autres hypothèses (p. ex. l'absence de contention) pourront être levées dans des travaux futurs.

Dans le cadre de cette étude, nous proposons un modèle de mémoire *partagée* à deux niveaux suffisamment général pour pouvoir être appliqué à la plupart des architectures embarquées réelles. Le premier (L1), le plus proche des unités de calcul, le plus performant et donc le plus coûteux, est présent en quantité réduite exprimée en nombre d'*emplacements* : il accueille uniquement les données vidéo en cours de traitement ; ainsi, pour H.264, il s'agira des macroblocs en cours de décodage. Le second est une mémoire externe, plus lointaine et donc souffrant d'une plus grande latence, en contrepartie d'une taille suffisante pour contenir à la fois les données n'ayant pas encore été traitées et celles l'ayant déjà été. Les transferts entre ces deux niveaux sont assurés par un processeur spécifique dédié aux accès directs à la mémoire (*direct memory access*, DMA) ; du point de vue de l'ordonnanceur, il s'agit d'une ressource de calcul spécialisée capable d'exécuter uniquement les tâches gérant ce type de transferts.

3. Ordonnancement

3.1. Taxinomie

Traditionnellement, on répartit les algorithmes d'ordonnancement de tâches en deux catégories selon qu'ils sont statiques – ils s'exécutent avant le chargement du programme – ou dynamiques – ils s'exé-

	Assignation	Ordre	Date de début
Totalement dynamique	Exécution	Exécution	Exécution
Assignation statique	Compilation	Exécution	Exécution
Auto-séquencé	Compilation	Compilation	Exécution
Totalement statique	Compilation	Compilation	Compilation

TABLE 1 – Répartition des activités d'un ordonnanceur (en haut) en fonction de la stratégie (à gauche).

cutent en même temps que le programme. Lee & Ha [11] notent qu'en réalité le travail d'un ordonnanceur peut se décomposer en trois opérations distinctes :

1. l'assignation des tâches aux processeurs ;
2. la spécification de l'ordre d'exécution sur chaque processeur ;
3. la détermination de la date de début de chaque tâche.

Il en résulte que la classification peut s'étoffer jusqu'à comprendre quatre catégories (cf. tableau 1) :
– totalement dynamique : les trois opérations ont lieu à l'exécution ;
– assignation statique : seule l'allocation est faite à la compilation ;
– auto-séquencé : le programme détermine lui-même à l'exécution la date où il peut lancer la tâche suivante ;
– totalement statique : les trois activités se font à la compilation.
Le choix de la stratégie s'effectue selon le compromis suivant : plus un ordonnanceur sera statique, moins il induira de coûts dans l'implémentation, mais moins il sera efficace dans le cas de tâches à durées non déterministes ; au contraire, plus un ordonnanceur sera dynamique, plus il sera à même de prendre en compte d'éventuelles variations dans la durée des tâches, mais plus il sera coûteux à mettre en œuvre. Dans le cas des codecs vidéo tels que H.264 [18] et HEVC [15], un ordonnanceur totalement statique est d'emblée exclu car la durée des tâches est soumise à de fortes variations ; à l'opposé, un ordonnanceur totalement dynamique aurait un coût relativement lourd pour un système embarqué et n'est donc guère envisageable. L'assignation et le choix de la date de début doivent donc nécessairement être statique et dynamique, respectivement ; la discussion sur l'opportunité de définir l'ordre des tâches à l'exécution est quant à elle reportée à la section 6.
En outre, un paramètre supplémentaire est à prendre en compte dans la mise au point d'un ordonnanceur : le type d'algorithme. Pour cette étude, nous retenons uniquement l'ordonnancement de liste à priorités statiques pour sa faible complexité et ses relativement bonnes performances [1]. Le principe est d'assigner des priorités aux tâches à exécuter et de les placer dans une liste ordonnée par priorité décroissante ; parmi les tâches disponibles, la première à être traitée sera toujours celle ayant la plus haute priorité, c'est-à-dire la première de la liste. En cas d'égalité, les *ex æquo* sont départagés aléatoirement.

3.2. HEFT

Parmi les ordonnancements de liste les plus répandus, celui ayant un des meilleurs rapports performances-complexité est HEFT (*Heterogeneous Earliest Finish Time*). Cette heuristique proposée par Topcuoglu *et al.* [16] détermine un ordonnancement totalement statique d'un DAG sur un environnement hétérogène de manière à minimiser la durée totale d'exécution de l'application (*makespan*). Il est possible de le convertir en ordonnancement auto-séquencé, voire à assignation statique, en éliminant les informations que l'on souhaite déterminer à l'exécution [11].
Dans le détail, l'algorithme est constitué de deux phases principales :

1. détermination des priorités ;
2. sélection du processeur.

La première étape consiste donc à affecter lesdites priorités aux tâches à ordonnancer. La priorité d'une tâche est son *bottom-level*, c'est-à-dire la longueur du chemin critique de cette tâche au puits du graphe lorsque l'on considère la moyenne des durées des communications et des calculs sur les ressources hétérogènes.

La seconde étape peut alors s'effectuer comme suit : lorsqu'une tâche est retirée de la liste, l'algorithme recherche le processeur capable de minimiser sa date de fin – y compris en tirant parti des temps d'inactivité laissés à ce stade par l'ordonnanceur (*insertion*) – et la lui alloue dans la mesure où toutes les dépendances de données sont satisfaites.
HEFT est particulièrement adapté dans le cadre de cette étude sous plusieurs aspects : au plan embarqué, il s'agit d'un algorithme à faible complexité, facile à implémenter et qui, pour cette raison, se prête bien à une stratégie partiellement dynamique ; au plan hétérogène, il prend en compte les variations de vitesse d'exécution entre les ressources de calcul en fonction des tâches ; au plan applicatif, les algorithmes de décodage vidéo sont facilement représentables sous forme de graphes de tâches. Néanmoins, il présente un certain nombre de limites : l'hypothèse est faite qu'un processeur quelconque peut exécuter une tâche quelconque, ce qui n'est pas le cas dans le modèle retenu (cf. section 2.2) ; lorsque la quantité de données échangées entre deux tâches est importante et que celles-ci sont assignées à des ressources de calcul « éloignées », le coût peut être significativement plus élevé que si les deux étaient allouées sur le même processeur en dépit d'une vitesse d'exécution inférieure ; et, surtout, la notion de mémoire en est absente.
Les deux premiers points ayant déjà été soulevés et résolus par l'algorithme SDC de Shi & Dongarra [13], celui-ci servira de point de départ à notre étude. En revanche, le dernier point n'ayant, à notre connaissance, pas été traité et la gestion de la mémoire étant l'une des clés de voûte de la conception des systèmes embarqués, il fait l'objet de cette étude.

4. Vers un modèle prenant en compte la mémoire

Pour prendre en compte les contraintes de mémoire, nous proposons les améliorations suivantes au modèle de calcul décrit section 2. Les tâches du DAG se répartissent en deux catégories :
– celles effectuant des calculs ;
– celles effectuant des transferts mémoire.
Les premières peuvent être exécutées par un processeur généraliste et éventuellement par un accélérateur, tandis que les secondes sont exclusivement affectées au DMA. Ces dernières ont des propriétés particulières qui complexifient leur ordonnancement.
En effet, chacune de ces tâches va, lors de son exécution, soit consommer soit libérer une quantité constante de mémoire locale déterminée par ce que nous appelons des *unités de données* (UD) [1] à transférer, 1 UD occupant un emplacement tel que défini précédemment ; cette quantité s'exprime comme un coût algébrique : positif si la mémoire est consommée – c'est-à-dire si le transfert va de la mémoire externe à la L1 – ou négatif dans le cas contraire. Le nombre d'emplacements disponibles est mis à jour à chaque exécution de tâche en y soustrayant algébriquement son coût ; il ne peut jamais être négatif : lorsqu'il s'annule, il faut d'abord ordonnancer des tâches libératrices avant d'en exécuter d'autres consommatrices.
La figure 1 illustre le modèle décrit ci-dessus par un DAG représentant un algorithme d'amélioration de la qualité d'image appliquant une réduction du bruit (*temporal noise reduction*, TNR) à chaque ligne de pixels. Une flèche indique une dépendance de données (i.e. une relation d'antériorité à l'exécution). Chaque tâche effectuant le même traitement parallèle sur toutes les lignes incluses dans les images constituant une séquence vidéo, le graphe ne comprend qu'un exemplaire de chaque. Les nœuds à simple suffixe (p. ex. `frameController_0`) sont exécutés une fois par image et ceux à double suffixe (p. ex. `tempUV_0_0`) une fois par ligne ; les chiffres indiquent respectivement les numéros d'image et de ligne. Les tâches gérant la mémoire apparaissent en traits non pleins : discontinus pour la consommation, pointillés pour la libération ; les autres se chargent du contrôle et des calculs. `hostController` est exécuté par le processeur hôte du SoC pour introduire une image en mémoire externe ; `frameController` lance son traitement depuis un processeur généraliste ; `lineController0` et `1` programment le DMA pour respectivement lire et écrire les données en mémoire externe *via* `src` et `dst` qui transfèrent chacun 1 UD. `estimateLineNoise` et `estimateFrameNoise` estiment le niveau de bruit de l'image n afin de calibrer le traitement à appliquer à l'image $n+1$. Enfin, `spaY`, `tempUV`, `tempY` et `motionDetect` analysent l'image afin de permettre à `fading` d'appliquer la correction adéquate.

1. L'idée des unités de données est de modéliser la consommation mémoire d'une manière similaire à ce qui est fait dans les modèles *dataflow* avec les jetons (*tokens*).

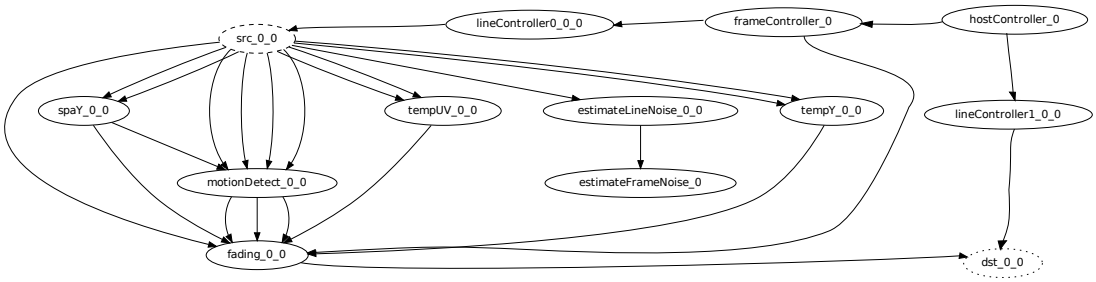

FIGURE 1 – Exemple de graphe de tâches pour une application d'amélioration de la qualité d'image (TNR). Une seule ligne est traitée. Pour n lignes, il faut exécuter les tâches à double suffixe n fois. `src_0_0` consomme de la mémoire, `dst_0_0` en libère. Le successeur de `estimateFrameNoise_0` est `frameController_1` et n'est donc pas représenté sur ce schéma.

5. Adaptation de l'heuristique d'ordonnancement

Pour faire face à ces nouvelles contraintes, le processus d'ordonnancement doit être adapté ; en effet, le simple fait de comptabiliser la mémoire introduit des dépendances implicites qui n'apparaissent pas dans le graphe de tâches initial et ne peuvent par conséquent pas être prises en compte par l'ordonnanceur. Pour y remédier, on construit un nouveau graphe de tâches, non orienté, comprenant uniquement celles affectées à la gestion mémoire et reliant toutes les paires n'ayant pas de dépendance de données, c'est-à-dire les nœuds entre lesquels il n'existe pas de chemin dans le DAG original ; cette représentation est appelée *graphe d'indépendance*.

L'idée est de représenter les liens entre tâches effectuant des transferts mémoire qui n'apparaissent pas comme des dépendances de données. L'exploitation de ce graphe permet alors un ajustement des priorités en vue d'anticiper l'exécution des tâches libératrices, celles-ci constituant le principal point de blocage de l'ordonnancement.

Pour ce faire, une hypothèse supplémentaire doit être introduite : les tâches gérant la mémoire sont appariées de telle sorte que la somme des coûts de chaque paire soit nulle [2] ; on note $C(v_i)$ le compagnon de la tâche v_i. Alors, chaque tâche libératrice v_r peut bénéficier d'un bonus de priorité égal à la somme des priorités des tâches v_c répondant aux critères suivants :

– v_c est adjacente à v_r dans le graphe d'indépendance ;
– v_c a un coût positif, c'est-à-dire elle consomme de la mémoire ;
– $C(v_r)$ a une priorité supérieure à v_c.

Le premier critère assure de ne prendre en compte que les tâches n'ayant pas de lien d'antériorité préexistant ; le second empêche que les tâches libératrices s'influencent mutuellement ; le dernier garantit que les sections critiques – c'est-à-dire les ensembles de tâches de calcul comprises entre une paire de tâches affectant la mémoire – ne se chevauchent pas au moment de l'ordonnancement. Par exemple, si l'on reprend l'application de la figure 1, la section critique associée au traitement de la ligne 0 de l'image 0 est constituée des cinq tâches comprises entre `src_0_0` et `dst_0_0` ; leurs priorités ne doivent pas s'intercaler avec celles des autres lignes.

Par ailleurs, le processus d'insertion doit également être adapté pour les tâches gérant la mémoire. On suppose que l'état de la mémoire – c'est-à-dire le nombre d'emplacements disponibles – à chaque étape d'ordonnancement peut être connu rétrospectivement ; alors une tâche consommatrice peut être insérée si les conditions suivantes sont réunies :

– le créneau considéré dispose de suffisamment de mémoire ;
– l'insertion n'affectera pas les tâches suivantes.

La seconde exigence consiste à s'assurer que la somme partielle des coûts des tâches suivantes ajoutée algébriquement à l'état de la mémoire provisoire – c'est-à-dire le nombre d'emplacements disponibles après que la tâche ait été ordonnancée – reste toujours positif.

2. En pratique, cette hypothèse n'est pas contraignante car une application réelle finit toujours par libérer la mémoire qu'elle consomme.

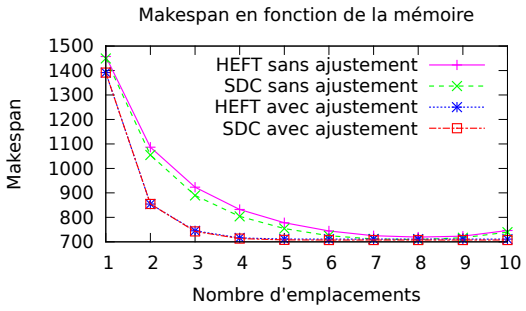

FIGURE 2 – Comparaison des *makespans* obtenus lors d'ordonnancements avec et sans ajustement des priorités en fonction du nombre d'emplacements mémoire. $a = 1,7$.

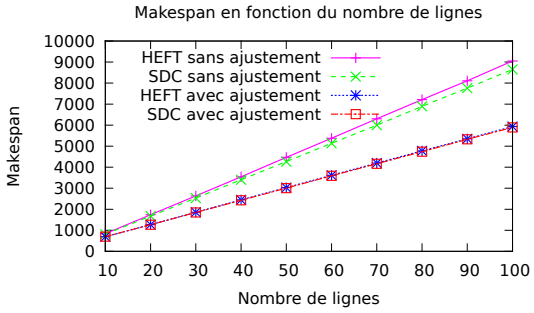

FIGURE 3 – Comparaison des *makespans* obtenus lors d'ordonnancements avec et sans ajustement des priorités en fonction du nombre de lignes de pixels à traiter. $a = 1,2$.

6. Expérimentations

Nous avons implanté l'ajustement des priorités dans HEFT et SDC, et nous l'avons évalué sur l'application présentée à la section 4. L'expérience a consisté à ordonnancer (sans exécuter) les tâches correspondant à l'analyse et au traitement de paquets de lignes extraites d'images contenues dans une séquence vidéo hypothétique. Pour modéliser cela, les durées des tâches ont été tirées aléatoirement pour chaque type de processeur en suivant le protocole suivant :

1. on définit pour chaque *type* de tâche (`src`, `fading`, etc.) une durée *unitaire* par nombre de pixels ;
2. on détermine la durée de *référence* w^r de chaque type de tâche en multipliant la durée unitaire par le nombre de pixels par ligne ;
3. pour que toutes les instances d'un type de tâche donné aient une variation similaire, on commence par déterminer aléatoirement la durée moyenne w^m de ce type de tâche. On fixe un paramètre de dispersion $a \geq 1$, et le tirage aléatoire est réalisé de manière à ce que w^m soit dans $[\frac{w^r}{a}, aw^r]$. Pour ce faire, on utilise une loi bêta[3] de paramètres $(\alpha, \beta) = (2,2) : w^m = w^r\,(\mathrm{Beta}(\alpha,\beta)(a-1/a) + 1/a)$. De plus, on impose : $\forall i,\ a \leq \sqrt{w^r_{i,j_s}/w^r_{i,j_h}}$ où w^r_{i,j_s} et w^r_{i,j_h} sont les durées de référence de la tâche v_i respectivement sur un processeur généraliste et un accélérateur, ce qui permet de garantir qu'un processeur généraliste ne puisse pas être, en moyenne, plus rapide qu'un accélérateur pour un type de tâche donné ;
4. la durée finale de chaque instance de tâche s'obtient de manière analogue à partir de la durée moyenne ainsi déterminée avec le même facteur de dispersion a.

Le premier jeu de simulations porte sur 10 lignes de 1000 pixels et comporte 10 000 tirages avec un générateur pseudo-aléatoire de type `mt19937` ; on compare les *makespans* obtenus en utilisant HEFT et SDC, avec et sans ajustement des priorités, en fonction du nombre d'emplacements mémoire. La figure 2 illustre les résultats obtenus. Que ce soit pour HEFT ou SDC, l'ordonnancement avec ajustement des priorités est toujours meilleur : l'accélération va de 4 % pour 1 emplacement à 20 % pour 2 emplacements, en passant par une moyenne de 10,6 %. La faible accélération pour 1 emplacement s'explique par l'impossibilité de tirer parti de l'insertion puisque le traitement d'une nouvelle ligne ne peut commencer sans que la précédente ne soit achevée ; en revanche, elle est maximale pour 2 emplacements car la forte contrainte mémoire induit des chevauchements dans l'ordonnancement sans ajustement qui dégradent considérablement le *makespan* ; cependant, cette dégradation s'érode avec la croissance du nombre d'emplacements.

La seconde expérience consiste à fixer le nombre d'emplacements mémoire à 2 et à faire varier le nombre de lignes de pixels de 10 à 100, les autres paramètres restant les mêmes que précédemment ; ceci permet

3. On rappelle que, contrairement à la loi de Gauss qui a un support sur $]-\infty,+\infty[$, la loi bêta a un support sur $[0,1]$ et que quand $\alpha = \beta$ l'espérance est 0,5.

d'évaluer l'impact de la croissance de la taille de l'application à ordonnancer sur le *makespan*. La figure 3 présente les résultats recueillis. On observe que le *makespan* croît linéairement avec le nombre de lignes et que la pente est inférieure de l'ordre de 21 % dans le cas ajusté, conformément à nos attentes, aussi bien pour HEFT que pour SDC.

Suite à la discussion de la section 3.1, nous proposons une méthodologie visant à déterminer l'opportunité de définir l'ordre des tâches à l'exécution. Pour ce faire, nous considérons trois ordonnancements :

– un ordonnancement de référence totalement statique utilisant les durées des tâches de référence [4] ;
– un ordonnancement auto-séquencé dont l'assignation et l'ordre sont issus de l'ordonnancement de référence mais dont les dates de départ sont déterminées à partir des durées finales des tâches ; cet ordonnancement modélise l'exécution d'une application dont on ne connaîtrait à la compilation que les durées moyennes des tâches ;
– un « oracle » connaissant les durées finales exactes des tâches et effectuant un ordonnancement complet dynamiquement.

L'écart en termes de *makespan* entre les deux derniers permet de mesurer le gain potentiel qu'apporterait un ordonnanceur partiellement dynamique dans le cas d'une exécution réelle.

Cette méthodologie est appliquée à l'application TNR : les paramètres de simulation sont les mêmes que précédemment et l'on compare les *makespans* obtenus avec l'ordonnancement auto-séquencé et l'oracle en fonction du paramètre de dispersion α ; les résultats sont présentés figure 4. On observe que l'écart croît avec la variation des durées des tâches jusqu'à atteindre 10,5 %, ce qui est cohérent puisque l'oracle est capable de compenser ces variations en changeant l'assignation d'une tâche qui prendrait trop de temps ou de l'insérer en tirant parti du temps d'inactivité antérieur. Dans ce cas, on conclut que si l'on observe des variations correspondant à un paramètre de dispersion supérieur à 1,2 alors un ajustement dynamique de l'ordonnancement peut être envisagé.

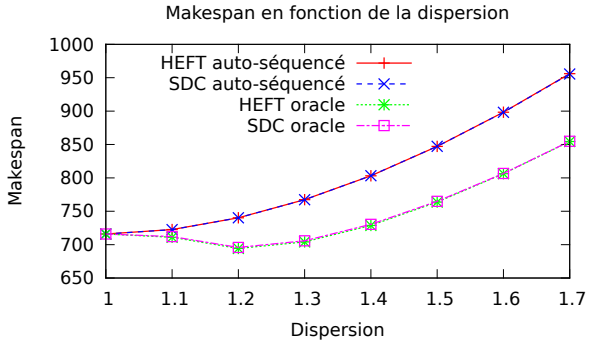

FIGURE 4 – Comparaison des *makespans* obtenus lors d'ordonnancements de types oracle et auto-séquencé en fonction du paramètre de dispersion α.

7. Travaux connexes

L'ordonnancement de graphes de tâches est montré NP-complet pour le cas homogène dans [7]. Ceci justifie d'utiliser des heuristiques pour résoudre le problème. Dans le cas hétérogène, de nombreuses heuristiques ont été proposées (voir [5] pour une étude sur une vingtaine d'entre elles). Les contraintes de mémoire sont, entre autres, prises en compte dans les ordonnancements statiques de nids de boucles par les compilateurs, pour les graphes *dataflow* [4] pour les applications et pour les *batch-scheduler* [3]. Cependant, nous n'avons pas trouvé d'études sur l'ordonnancement de graphe de tâches pour machine hétérogène à l'aide d'heuristique de liste visant à la fois la minimisation du *makespan* et le respect de contraintes mémoire de certaines tâches. Dans le domaine de l'embarqué, le problème d'exécuter une application multimédia sur un SoC est souvent modélisé par un ordonnancement de graphe *dataflow*. Cependant, les modèles récents dérivés du *synchronous dataflow* (SDF [10]) comme le SPDF (*Schedulable Parametric Dataflow* [6]), ne prennent pas en compte toute la dynamique de l'application, et en particulier la variation de la durée des tâches, comme c'est le cas dans cette étude.

Par ailleurs, il existe des ordonnancements de liste à priorités non statiques : c'est le cas par exemple du *dynamic level scheduling* (DLS) [14] où les priorités varient au cours du processus d'ordonnancement. Cette classe d'heuristiques a été exclue de notre étude car, bien que pouvant se prévaloir de bons résultats, elles souffrent de temps d'exécution conséquents [16]. Enfin, dans le domaine du temps réel, [2] présente une technique d'ordonnancement prenant en compte une capacité mémoire limitée, mais elle s'applique à des tâches préemptibles avec *deadlines*, ce qui n'est pas compatible avec notre modèle.

4. Cette ordonnancement de référence est uniquement utilisé comme base pour l'ordonnancement auto-séquencé. Les *makespans* qu'il produit n'ont pas d'intérêt direct dans le cadre de notre étude et ne sont donc pas représentés.

8. Conclusion

Nous avons présenté une méthode visant à introduire la notion de mémoire dans la plupart des algorithmes d'ordonnancement de liste en procédant à un ajustement de priorités. Nous l'avons implantée dans HEFT et SDC. De plus, nous avons montré qu'il était encore possible de recourir à un mécanisme d'insertion de tâches. Les expériences menées ont révélé que ces deux contributions rendent possible une amélioration de plus de 20 % par rapport à une implémentation non optimisée. En outre, nous avons proposé une méthodologie reprenant les techniques mises au point en vue de mesurer les gains potentiels apportés par un ordonnancement semi-dynamique dans le cadre d'environnements sous contrainte de mémoire, tels que les systèmes embarqués utilisés pour le décodage vidéo ou le traitement d'image. Nos prochains travaux s'intéresseront aux problèmes spécifiques posés par les algorithmes de décodage vidéo dynamiques tels que H.264 et HEVC. En effet, le modèle actuel fait l'hypothèse que le graphe de l'application à ordonnancer est statique ; il faudra donc y introduire les aspects de reconfiguration.

Bibliographie

1. Adam (T. L.), Chandy (K.) et Dickson (J.). – Comparison of list schedules for parallel processing systems. *Communications of the ACM*, vol. 17, n12, 1974.
2. Baker (T. P.). – Stack-based scheduling for realtime processes. *Real-Time Syst.*, vol. 3, n1, 1991.
3. Batat (A.) et Feitelson (D.). – Gang scheduling with memory considerations. *In : Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*. IEEE.
4. Buck (J.) et Lee (E.). – Scheduling dynamic dataflow graphs with bounded memory using the token flow model. *In : 1993 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP-93)*.
5. Canon (L.-C.), Jeannot (E.), Sakellariou (R.) et Zheng (W.). – Comparative Evaluation of the Robustness of DAG Scheduling Heuristics. *In : Integration Research in Grid Computing, CoreGRID integration workshop, 2008*.
6. Fradet (P.), Girault (A.), Poplavko (P.) et al. – Spdf : A schedulable parametric data-flow moc (extended version), 2011. Inria RR7828.
7. Garey (M. R.) et Johnson (D. S.). – *Computers and Intractability*. – Freeman, San Francisco, 1979.
8. Geng (T.) et al. – Parallelization of computing-intensive tasks of the h.264 high profile decoding algorithm on a reconfigurable multimedia system. *IEICE Transactions on Information and Systems*, vol. E93-D, n12, 2010.
9. Jian (G.-A.) et al. – A system architecture exploration on the configurable hw/sw co-design for h.264 video decoder.
10. Lee (E.) et Messerschmitt (D.). – Synchronous data flow. *Proceedings of the IEEE*, vol. 75, n9, 1987.
11. Lee (E. A.) et Ha (S.). – Scheduling strategies for multiprocessor real-time dsp. *IEEE Global Telecommunications inproceedings and Exhibition*, vol. 2, 1989.
12. Saponara (S.) et al. – Performance and complexity co-evaluation of the advanced video coding standard for cost-effective multimedia communications. *Eurasip Journal on Applied Signal Processing*, vol. 2004, n2, 2004.
13. Shi (Z.) et Dongarra (J. J.). – Scheduling workflow applications on processors with different capabilities. *Future Generation Computer Systems*, vol. 22, n6, 2006.
14. Sih (G. C.) et Lee (E. A.). – Compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, n2, 1993.
15. Sullivan (G.) et Ohm (J.-R.). – Recent developments in standardization of high efficiency video coding (hevc). *Proceedings of SPIE - The International Society for Optical Engineering*, vol. 7798, 2010.
16. Topcuoglu (H.), Hariri (S.) et Wu (M.-Y.). – Task scheduling algorithms for heterogeneous processors. *In : 8th IEEE Heterogeneous Computing Workshop (HCW'99)*.
17. Wang (S.-H.) et al. – A software-hardware co-implementation of mpeg-4 advanced video coding (avc) decoder with block level pipelining. *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 41, n1, 2005.
18. Wiegand (T.) et al. – Overview of the h.264/avc video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, n7, 2003.