The Data-Flow Equations of Checkpointing in reverse Automatic Differentiation

Benjamin Dauvergne[1] and Laurent Hascoët[1]

INRIA Sophia-Antipolis, TROPICS team,
2004 Route des lucioles, BP 93, 06902 Sophia-Antipolis, France

Résumé Le checkpointing est une technique pour réduire la consommation en mémoire des programmes adjoints produits par la Différentiation Automatique. Cependant, le checkpointing utilise aussi un espace mémoire non négligeable pour ce qu'on appelle des "snapshots". Nous analysons le flot d'informations le flux de données du checkpointing fournissant une caractérisation précise des choix optimaux en mémoire pour faire un snapshot. Cette caractérisation est dérivé formellement de la structure des checkpoints et d'équation classique de flux de donnée. En particulier, nous sélectionnons deux choix très différents et étudions leurs comportements sur un cerains nombres de codes réels. Bien qu'aucun choix est uniformément le meilleur, le choix nommé "snapshot-retardé" apparait comme préférable en général.

1 Introduction

2 Reverse Automatic Differentiation

Les dérivées mathématiques sont des ingrédients essentiels en Calcul Scientifique. En particulier, les gradients sont primordiaux en optimisation et pour les problèmes inverses. Les méthodes pour calculer des gradients peuvent être classifiés en deux catégories. Dans la première catégorie, les méthodes utilisent le processeurs de manière plus intensive parce que de nombreuses opérations sont dupliqués. Cela peut arriver via l'usage répété de dérivés tangents ou via l'usage de la Différentiation Automatique en utilisant la stratégie du "Tout-Recalcul". Ce n'est pas le contexte de ce papier. Dans la seconde catégorie, les méthodes économisent la duplications des opériations via une utilisation plus importante de mémoire. Cette catégorie englobe la résolution codé-à-la-main des équations adjointes ou la Différentiation Automatique via la stratégie du "Tout-Mémorisation", qui est le contexte de ce papier.

Étant une technique de transformation de programmes, la DA inverse peut et doit prendre avantage des analyses de programmes et des technologies de compilation [1] pour réduire ses problèmes de performances. Dans ce papier, nous analyserons le checkpointing une technique de DA qui permet d'échanger du recalcul contre de la mémorisation, en utilisant les outils d'analyse du flux de donnée en compilation. Le checkpointing offre un champ d'options qui influencent le code différentié résultant. Notre but est de formaliser ces options et de trouver

lesquels sont optimals. Cette étude fait partie d'un effort général pour formaliser toutes les techniques de compilation utilie à la différentiation automatique, de sorte que les outils de différentiation automatique puissent faire les bons choix sur des bases fermement établies.

3 La Différentiation Automatique Inverse

La différentiation automatique par transformation de programme se base sur un programme P calculant une fonction différentiable F et crée un nouveau programme qui calculent des dérivées de F. À partir de la règle de différentiation des fonctions composées, la DA inserts dans une copie de P des instructions, en quelques sortes dérivées, chacune correspondant à une instruction d'origine de P. En particulier, la DA inverse crée un programme $\overline{P}$ qui calculent des gradients. Dans $\overline{P}$, les instructions dites dérivées sont exécutés dans un ordre inverse de celui des instructions originelles de P. Les instructions dérivées utilisent certains valeurs produites par les instructions originels, et par conséquent les instructions originelles doivent être éxécutées dans une première passe, désigné par *passe avant* ou bien $\overrightarrow{P}$, qui produit toutes les valeurs originelles qui seront utilisé par les instructions originelles qui elles forme une seconde passe, désigné par *passe arrière* ou bien $\overleftarrow{P}$. Ceci est illustré par la figure . 1, dans laquelle nous avons présentement partager P en trois parties successives, U l'amont du code, C le centre du code et D l'aval du code. Dans notre contexte les valeurs originelles sont préservés pour la *passe arrière* via l'utisation de routines `PUSH` et `POP`, utilisant une pile que l'on appelle généralement la *bande*. Toutes les valeurs originelles ne

Fig. 1. Structure de base d'un programme produit par différentiation automatique inverse

sont pas requises par la *passe arrière*. Par la nature de la différentiations, les valeurs employés de manière linéaire par une expression ne sont pas requise par son expression dérivée. L'analyse "To Be Recorded" [2,6] (TBR ou "À Préserver") trouve l'ensemble des valeurs requises dénoté par *Req*. L'ensemble *Req* évolue à mesure de l'exécution de la *passe avant*. Par exemple dans la figure 1, l'analyse TBR de U trouve les variables requises pour $\overleftarrow{U}$ (i.e. l'ensemble $\mathbf{use}(\overleftarrow{U})$, qui doit être préservé entre la fin de l'exécution de $\overrightarrow{U}$ et le début de l'exécution de $\overleftarrow{U}$. À cette fin, chaque fois qu'une valeur requise est sur le point d'être écrasé par la prochaine instruction, elle est empilé par `PUSH` auparavant et ensuite dépilé par `POP` juste avant l'exécution de la dérivée de l'instruction d'écrasement.

Bien que complexe, la DA inverse peut être facilement appliqué par un outil automatique et elle a des avantages énormes concernant le coût calculatoire pour obtenir le gradient [4, chapter 3].

Dans [5], nous étudions les propriétés en terme de flux de donnée des programmes produits par différentiation automatique inverse, dans le cas simple

Fig. 1, i.e. sans checkpointing. Nous formalisions la structure de ces programmes et dérivions des équations de flux de donnée pour l'analyse d'"adjoint liveness" ou de "durée de vie adjointe", qui repère les instructions originelles qui sont inutiles dans le programme différentié et pour l'analyse TBR. Dans ce papier nous nous concentrerons sur les conséquences de l'introduction du checkpointing, de ce point de vue ce papier, bien qu'autonome, est la suite de [5].

4 The equations of Checkpointing Snapshots

Le checkpointing modifie la structure du code différentié pour réduire le pic de consommation mémoire. Quand un fragment de code C est *checkpointé* — notation $[C]$ — l'adjoint est formellement défini par les règles de réécriture récursives suivantes :

$$\begin{aligned} \boxed{Req \vdash \overline{[C];D}} = \ & \texttt{PUSH}(Sbk); \\ & \texttt{PUSH}(Snp); \\ & C; \\ & \boxed{Req_D \vdash \overline{D}} \\ & \texttt{POP}(Snp); \\ & \boxed{Req_C \vdash \overline{C}} \\ & \texttt{POP}(Sbk); \end{aligned} \tag{1}$$

Les boites désignent les termes à réécrire considérant les termes hors des boites comme le code qui reste tel quel. Cette nouvelle structure du code est esquissé dans la figure.2, à comparé à la figure 1. Désormais, C est d'abord exécuté dans

Fig. 2. Checkpointing in reverse AD

sa version originel — c'est une simple copie du programme d'origine — de sorte que la consommation de pile de $\overline{D} \doteqdot \overrightarrow{D}; \overleftarrow{D}$ — ; dénoté séquentialité du code — est libéré préalablement à l'exécution de $\overline{C} \doteqdot \overrightarrow{C}; \overleftarrow{C}$. Le pic de consommation mémoire par la pile durant $\overline{[C];D}$ est donc réduit au niveau du maximum du pic de consommation après $\overrightarrow{C}$ et du pic de consommation après $\overrightarrow{D}$. Cependant l'exécution dupliqué de C requièrent qu'un nombre suffisant de valeurs — le *snapshot*, photographie mémoire, ou l'ensemble Snp — soient préservées pour restaurer le contexte d'exécution de la prmière exécution de C. Cet usage consomme aussi de la mémoire sur la *bande*, bien que — généralement — que l'usage de la bande dans $\overrightarrow{C}$. Pour ne pas perdre les bénéfices du checkpointing, il est par conséquent essentiel de trouver l'ensemble *snapshot* le plus petit pour un fragment de programme C et de plus de trouver le choix du fragment C qui engrengera le plus grand bénéfice en terme de réduction de la consommation, sans trop aggraver le coût des recalculs.

Cela se révèle difficile, en effet un *snapshot* agrandit peut signifier un usage de la pile réduit dans les *passes avants* et inversement. Par conséquent, contrairement à ce qui arrive dans le cas sans *checkpoints*, il n'y a pas un meilleur choix unique évident. Il semble qu'il y ait plusieurs choix "optimaux", parmis lesquels aucun n'est meilleur ou pire que les autres. Notre but est ici d'établir les contraintes qui définissent et lient les ensembles de *snapshot* et ceux définissant l'usage de la *bande*, et de caractériser les éventuels choix optimaux. Pour notre outil de DA TAPENADE, nous avons arrêté notre choix sur une solution — *cf* Sect. 5 — que nos évaluations ont désigné comme un bon choix en moyenne.

Quatres ensembles indéterminés de variables : Éxaminons la définition (1) du checkpointing en détail. Le contexte de réécriture *Req* est l'ensemble requis entrant de variables imposé par U, qui doivent être préservées au travers de l'exécution de $Req \vdash \overline{[C];D}$. De même, Req_D et Req_C sont les ensembles de variables que $\vdash \overline{C}$ et $\vdash \overline{D}$ devront préserver, respectivement. Pour nous, Req_D et Req_C sont inconnus et devront être déterminer ensemble avec le *snapshot*. Concernant le *snapshot*, de par la strcuture de pile, il y a deux instants ouà les variables puissent être restaurées depuis la *bande* : avant $\overrightarrow{C}$ ou après $\overleftarrow{U}$. Par conséquent nous introduisons deux ensembles de *snapshot* :

- *Snp*, l'ensemble de *snapshot usuel*, contient les variables à restaurer avant $\overline{C}$, assurant ainsi que leurs valeurs seront identiques lors des deux exécutions de C et $\overrightarrow{C}$;
- *Sbk*, le *snapshot arrière*, contient les variablesà restaurer avant $\overleftarrow{U}$, assurant ainsi que quelquesoit leur sort durant $Req \vdash \overline{[C];D}$, leurs valeurs seront préservées pour l'exécution de $\overleftarrow{U}$.

L'usage de *Sbk* en lieu et place de *Snp* et Req_C peut probablement améliorer le traffic mémoire. En tout, nous avons quatres ensemble à déterminer : Req_D, Req_C, *Sbk* et *Snp*. Ces ensembles doivent respecter des contraintes paramétrées par *Req*, Req_D, Req_C, *Sbk*, *Snp*, et par des ensembles issue d'autres analyse du flux de donnée tels que **use**, les variables lues, et **out**, les variables au moins partiellement écrasées du code C, D, $\overline{C}$, et $\overline{D}$. Ces contraintes garantissent que le *checkpointing* préserve la sémantique du code différentié, i.e. la valeur des dérivés.

Deux conditions nécessaires et suffisantes : La figure 1 montre le programme différentié dans le cas de référence sans checkpoints. Ce programme de référence est supposé correct. Tout ce que nous devons garantir c'est que le résultats du programme différentié — les dérivées partielles — sont inaltérés par l'emploi du *checkpointing*. Par conséquent les dérivées sont préservées si et seulement si les variables originelles — i.e. ne contenant pas les valeurs des dérivées partielles intermédiaires — qui sont utilisés lors de la passe arrière contiennent les même valeurs que sans l'utilisation du checkpointing. En d'autres mots le *snapshot* et la *bande* doivent préserver l'ensemble **use** de $\overline{C}$ entre les instants t_1

et t_3 i.e.

$$\mathbf{out}\begin{pmatrix}\texttt{PUSH}(Sbk);\\ \texttt{PUSH}(Snp);\\ C;\\ Req_D \vdash \overline{D};\\ \texttt{POP}(Snp);\end{pmatrix} \bigcap \mathbf{use}(Req_C \vdash \overline{C}) = \emptyset \tag{2}$$

et l'ensemble **use** de $\overleftarrow{U}$ qui par définission est aussi Req entre les instants t_1 et t_4 i.e.

$$\mathbf{out}\begin{pmatrix}\texttt{PUSH}(Sbk);\\ \texttt{PUSH}(Snp);\\ C;\\ Req_D \vdash \overline{D};\\ \texttt{POP}(Snp);\\ Req_C \vdash \overline{C};\\ \texttt{POP}(Sbk);\end{pmatrix} \bigcap Req = \emptyset \ . \tag{3}$$

Ensuite tout devient purement mécanique. L'emsemble **out** d'une séquence de code est classiquement :

$$\mathbf{out}(A; B) = \mathbf{out}(A) \cup \mathbf{out}(B) \ ,$$

except in the special case of a PUSH/POP pair, which restore their argument :

$$\mathbf{out}(\texttt{PUSH}(v); A; \texttt{POP}(v)) = \mathbf{out}(A) \setminus \{v\} \ .$$

Le mécanisme de la DA inverse assure aussi que les variables requises par le contexte de réécriture sont effectivement préservées et cela n'affecte pas l'ensemble des variables lues. En écrivant court $\overline{A} \doteqdot \emptyset \vdash \overline{A}$, nous avons :

$$\begin{aligned}\mathbf{out}(Req \vdash \overline{A}) &= \mathbf{out}(\overline{A}) \setminus Req\\ \mathbf{use}(Req \vdash \overline{A}) &= \mathbf{use}(\overline{A}) \ .\end{aligned}$$

De plus, un PUSH seul ne peut écraser aucune variable, par conséquent l'équation (2) se réduit en :

$$\big(\mathbf{out}(C) \cup (\mathbf{out}(\overline{D}) \setminus Req_D)\big) \setminus Snp \bigcap \mathbf{use}(\overline{C}) = \emptyset \tag{4}$$

et l'équation (3) en :

$$\big(\big(\mathbf{out}(C) \cup (\mathbf{out}(\overline{D}) \setminus Req_D)\big) \setminus Snp \cup \big(\mathbf{out}(\overline{C}) \setminus Req_C\big)\big) \setminus Sbk \bigcap Req = \emptyset \ . \tag{5}$$

Des équations (4) et (5), nous obtenons des conditions équivalentes concernant Sbk, Snp, Req_D et Req_C :

$$\begin{aligned}Sbk &\supseteq \big((\mathbf{out}(C) \cup (\mathbf{out}(\overline{D}) \setminus Req_D)) \setminus Snp\\ &\qquad\qquad \cup\ (\mathbf{out}(\overline{C}) \setminus Req_C)\big) \cap Req\\ Snp &\supseteq \big(\mathbf{out}(C) \cup (\mathbf{out}(\overline{D}) \setminus Req_D)\big) \cap \big(\mathbf{use}(\overline{C}) \cup (Req \setminus Sbk)\big)\\ Req_D &\supseteq (\mathbf{out}(\overline{D}) \setminus Snp) \cap \big(\mathbf{use}(\overline{C}) \cup (Req \setminus Sbk)\big)\\ Req_C &\supseteq (\mathbf{out}(\overline{C}) \setminus Sbk) \cap Req \ .\end{aligned}$$

Remarquez le cycle dans ces équations, grâce à lui si nous ajoutons une variables à l'ensemble *Snp*, nous pourrions l'enlever de l'ensemble Req_D, et vice versa : comme nous l'avions posés, il ne peut y avoir de solution unique. Intéressons nous aux solutions minimales de ces équations, i.e. celle où nous remplaçons le symbole "$\supseteq$" par une simple égalité "$=$".

Résolution des Ensembles Indéterminés : La manipulation de ces équations fastidieuse et sujette aux erreurs. Par conséquent nous avons utilisé un logiciel de manipulations symboliques d'expressions — e.g. Maple [8]. En fait nous avons remplacé l'équation, de par exemple *Snp* dans les autres équations, et ainsi de suite pour obtenir des équations définissant une point fixe avec une seule inconnue X de la forme

$$X = A \cup (X \cap B) \quad ,$$

dont les solutions sont de la form "A plus un sous ensemble de B". Les solutions s'expriment comme fonction des ensembles suivants :

$$\begin{array}{l} Snp_0 = \mathbf{out}(C) \cap (\mathbf{use}(\overline{C}) \cup (Req \setminus \mathbf{out}(\overline{C}))) \\ Opt_1 = Req \cap \mathbf{out}(\overline{C}) \cap \mathbf{use}(\overline{C}) \\ Opt_2 = Req \cap \mathbf{out}(\overline{C}) \setminus \mathbf{use}(\overline{C}) \\ Opt_3 = \mathbf{out}(\overline{D}) \cap (\mathbf{use}(\overline{C}) \cup Req) \setminus \mathbf{out}(C) \quad . \end{array} \tag{6}$$

Pour chaque partition de Opt_1 en deux ensembles Opt_1^+ et Opt_1^-, et de même pour Opt_2 et Opt_3, ce qui suit est un solution minimale de notre problème :

$$\begin{array}{lllll} Sbk & = Opt_1^+ \cup Opt_2^+ & & \\ Snp & = Snp_0 \cup Opt_2^- & \cup Opt_3^+ \\ Req_D & = & Opt_3^- \\ Req_C & = Opt_1^- \cup Opt_2^- \quad . \end{array} \tag{7}$$

Tout quadruplet d'ensembles *(Sbk, Snp,* Req_D*,* Req_C*)* qui préserve les dérivées — celle du code différentié sans checkpoints — est supérieur ou égal à une de ces solutions minimales. Remarquez que $Opt_1 \subseteq Snp_0$, Snp_0, Opt_2 et Opt_3 sont disjoints.

5 Discussion and Experimental Results

La décision finale pour les ensembles *Sbk*, *Snp*, Req_D et Req_C dépend de chaque contexte particuliers. Aucune stratégie n'est systématiquement la meilleure. Nous avons testé deux options.

Nous avons d'abord examiné l'option que nous avions implémenté par défaut dans notre outil de DA TAPENADE [7]. Nous l'appelons désormais le *snapshot immédiat* — "eager snapshot". Cette option préserve suffisamment de variable via le *snapshot* pour réduire les ensembles Req_D et Req_C autant que possible, réduisant par conséquent le nombre de `PUSH`/`POP` subséquents dans $\overline{D}$ et $\overline{C}$. Les

équations (7) montrent que nous pouvons même rendre ces ensembles vides mais l'expérience nous a montré que rendre Req_D vide peut éventuellement coûter trop de mémoire.

Comme toujours le problème qui se cache ici est l'indécidabilité des indexes de tableaux : puisque qu'on ne peut pas toujours dire si deux accés de tableaux désignent la même zone mémoire ou pas, la stratégie du *snapshot immédiat* peut consommer trop de mémoire pour les *snapshots* dans certains cas — l'ensemble **out** n'est pas suffisament circonscrit.

Par conséquent la stratégie du *snapshot immédiat* force Opt_1^- et Opt_2^- à être vides mais

$$\begin{aligned} Opt_3^+ &= \mathbf{out}(\overline{D}) \cap (\mathbf{use}(\overline{C}) \setminus Req) \setminus \mathbf{out}(C) \\ Opt_3^- &= \mathbf{out}(\overline{D}) \cap Req \setminus \mathbf{out}(C) \end{aligned}$$

ce qui donne :

$$\begin{aligned} Sbk &= Req \cap \mathbf{out}(\overline{C}) \\ Snp &= (\mathbf{out}(C) \cap (\mathbf{use}(\overline{C}) \cup Req \setminus \mathbf{out}(\overline{C}))) \cup \\ &\quad (\mathbf{out}(\overline{D}) \cap \mathbf{use}(\overline{C}) \setminus Req \setminus \mathbf{out}(C)) \\ Req_D &= \mathbf{out}(\overline{D}) \cap Req \setminus \mathbf{out}(C) \\ Req_C &= \emptyset \ . \end{aligned} \tag{8}$$

Remarquez que l'intersection entre *Sbk* et *Snp* non-vide ce qui requiert une mécanisme spécial lors de la génération des `PUSH`/`POP` pour les *snapshots* pour éviter une sauvegarde en double des valeurs concernées.

L'autre options que nous avons examiné en détail est celle de conserver un snapshot aussi petit que possible, laissant par conséquent le plus gros du travail de préservation des valeurs en *passe avant* et *arrière* au mécanisme de TBR dans $\overline{D}$ et $\overline{C}$. Nous l'avon appelé la stratégie du *snapshot retardé*, et c'est aussi la nouvelles stratégie par défaut dans TAPENADE. Sous-jacent est l'idée que le mécanisme utilisant l'analyse TBR est efficace pour les tableaux car quand une cellule d'un tableau est écrasé, seul celle-ci est sauvé car l'instruction de PUSH dispose de l'addresse de la cellule à cet instant.

Ainsi les *snapshot retardé* forcent les ensembles Opt_1^+, Opt_2^+, et Opt_3^+ à être vide. Il en résulte les équations suivantes :

$$\begin{aligned} Sbk &= \emptyset \\ Snp &= \mathbf{out}(C) \cap (Req \cup \mathbf{use}(\overline{C})) \\ Req_D &= \mathbf{out}(\overline{D}) \cap (Req \cup \mathbf{use}(\overline{C})) \setminus \mathbf{out}(C) \\ Req_C &= \mathbf{out}(\overline{C}) \cap Req \ . \end{aligned} \tag{9}$$

Nous avons essayé TAPENADE, configuré avec chacune des deux options précédentes, sur notre suite de d'applications de validation. Les résultats sont résumés dans la table 1. On voit que expérimentalement la stratégie du *snapshot retardé* se comporte mieux en général. En fait nous n'avons pu montré l'intérêt de la stratégie de *snapshot immédiat* que sur un exemple artificiel que nous avons écris nous mêmes dans lequel le fragment checkpointé C, écrase de manière répétitive les mêmes éléments d'un tableau, induisant une sauvegarde répété

des mêmes cellules par le mécanisme de préservation dans la *passe avant*, celle-ci devenant plus coûteuse qu'une sauvegarde unique du tableau entier dans le *snapshot*. Néanmoins, dans des applications réelles ce cas est semble-t-il rare et la stratégie retardée est donc meilleure.

Tab. 1. Comparaison des approches *snapshots immédiats* et *retardés* sur un certain nombres d'applications de tailles variées

Code	*Domain*	*Orig. time*	*Adj. time*	*Eager (8)*	*Lazy (9)*
OPA	oceanography	110 s	780 s	480 Mb	479 Mb
STICS	agronomy	1.8 s	35 s	229 Mb	229 Mb
UNS2D	CFD	2.7 s	23 s	248 Mb	185 Mb
SAIL	agronomy	5.6 s	17 s	1.6 Mb	1.5 Mb
THYC	thermodynamics	2.7 s	12 s	33.7 Mb	18.3 Mb
LIDAR	optics	4.3 s	10 s	14.6 Mb	14.6 Mb
CURVE	shape optim	0.7 s	2.7 s	1.44 Mb	0.59 Mb
SONIC	CFD	0.03 s	0.2 s	3.55 Mb	2.02 Mb
Contrived example		0.02 s	0.1 s	8.20 Mb	11.72 Mb

Quel que soit l'option choisie, les équations (7) capturent l'étendue des interactions entre *snapshots* possibles. Par example, si plusieurs *checkpoints* se suivent et tous utilisent un tableau `A`, mais seulement le dernier *checkpoint* — on confond le fragment de code avec le checkpoint qui le délimite — *snapshot* écrase `A`, il est bien connu que seul le dernier *snapshot* devrait contenir une référence à `A`. Pourtant, quand un outil de DA ne repose pas sur une formalisation du checkpointing tel que celle que nous venons d'introduire ici, il se pourrait bien que `A` soit sauvé par tous les snapshots.

6 Conclusion

Nous venons de formaliser les techniques de checkpointing dans le contexte de la DA inverse par transformation de programme. Le checkpoitning repose sur la sauvegarde d'un certain nombres de variables and plusieurs options s'offrent à nous quant à savoir quels variables doivent être sauvé et quand. En utilisant notre formalisation et avec l'aide d'un logiciel de calcul symbolique, nous trouvions qu'aucune configuration de *snapshot* n'est strictement supérieur aux autres mais nous arrivions à spécifier l'ensemble des configurations optimales. Les implémentations des outils de DA sont ainsi plus sûrs et plus fiables.

Nous sélectionnons deux configurations optimales particulières et les implémentions dans un outil de DA, TAPENADE. L'expérience montre que le choix appelé *snapshot retardé* se comporte le mieux dans la plupart des cas.

Néanmoins, nous penson que pour la DA inverse d'un programme donné, le choix de méthode de calcul des *snapshot* n'a pas à être identique pour tous les

checkpoints. Cette description formelle de toutes les configurations possibles nous permet d'examiner la meilleur pour chaque checkpoint basé sur des propriétés statiques de cet emplacement particulier du code. À cet égard nous avons utilisé le calcul symbolique une nouvelle fois et nous avons trouvé une propriété très plaisante : pour un checkpoint doné, quelquesoit la configuration optimale choisie pour le snapshot, l'ensemble **out** de ce fragment de code se révèle être constant :

$$\mathbf{out}(\overline{C;D}) = \big(\mathbf{out}(\overline{C}) \cup ((\mathbf{out}(\overline{D}) \cup \mathbf{out}(C)) \setminus \mathbf{use}(\overline{C}))\big) \setminus \mathit{Req} \ ,$$

Si les *checkpoints* sont imbriqués, cet ensemble **out** est ce qui influence les *checkpoints* intérieurs. Par conséquent le choix d'une configuration optimale est local à chaque checkpoint.

Un des grand challenge pour la DA inverse est de trouver le meilleur placement pour l'imbrication des checkpoints. Cet imbrication a déjà été trouvé [3] dans le cas d'un flot de contrôle simple avec des performances uniformes. Pour des programmes arbitraires, nous pensons avoir montré par nos formules que la segmentation du code en sous-sections U, C, and D a un impact substantiel sur la consommation mémoire et que ces mêmes formules sont un pas pour trouver de bonnes segmentations.

Références

1. A. Aho, R. Sethi, and J. Ullman. *Compilers : Principles, Techniques and Tools.* Addison-Wesley, 1986.
2. C. Faure and U. Naumann. Minimizing the tape size. In G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors, *Automatic Differentiation of Algorithms : From Simulation to Optimization*, Computer and Information Science, chapter 34, pages 293–298. Springer, New York, NY, 2001.
3. Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1 :35–54, 1992.
4. Andreas Griewank. *Evaluating Derivatives : Principles and Techniques of Algorithmic Differentiation.* Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, PA, 2000.
5. L. Hascoët and M. Araya-Polo. The adjoint data-flow analyses : Formalization, properties, and applications. In H. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *Automatic Differentiation : Applications, Theory, and Tools*, Lecture Notes in Computational Science and Engineering. Springer, 2005.
6. L. Hascoët, U. Naumann, and V. Pascual. "to be recorded" analysis in reverse-mode automatic differentiation. *Future Generation Computer Systems*, 21(8), 2004.
7. L. Hascoët and V Pascual. Tapenade 2.1 user's guide. Technical report 0300, INRIA, 2004. http ://www.inria.fr/rrrt/rt-0300.html.
8. Darren Redfern. *The Maple handbook, Maple V, release 4.* Springer, 1996.