Documentation de l'Extension

Projet Génie Logiciel — ENSIMAG

Équipe : Groupe 8, équipe 43
Members :
Fatima Azzahra Ardan,
Mohammed EL ARABI,
Faical EL GOUIJ,
Hamza MOUNTASSIR,
Ayoub TOUATI

Tuteur : M. NICOLLIN Xavier

Date : 22 Janvier 2026

Contents

1 Introduction

Cette documentation détaille l'extension OPTIM ajoutée au DecacCompiler, réalisée dans le cadre du projet de génie logiciel. Elle vise à expliquer les optimisations appliquées au code généré par le compilateur Deca, afin d'améliorer l'efficacité du programme sans altérer son comportement.

Le document présente les techniques d'optimisation mises en œuvre, notamment la propagation des constantes, la propagation des copies, l'élimination du code mort, des optimisations avancées de type peepholes, des simplifications algébriques, le constant folding et la dead store elimination. Ces optimisations visent à réduire la taille du code généré, à accélérer son exécution et à mieux utiliser les ressources du système.

Cette documentation aborde également les choix d'architecture et d'algorithmes qui ont guidé la mise en place de ces optimisations, ainsi que la méthode de validation utilisée pour en évaluer l'efficacité. Enfin, les résultats obtenus après l'application des optimisations seront analysés, en mettant en évidence l'impact sur les performances du compilateur DecacCompiler.

2 Objectif

L'objectif de l'extension `OPTIM` est d'améliorer la performance du compilateur `DecacCompiler` en appliquant diverses techniques d'optimisation sur le code intermédiaire généré. Ces optimisations ont pour but de rendre le code plus compact, plus rapide à exécuter et, en conséquence, plus efficace en termes de consommation de ressources. Elles incluent :

- Les **optimisations peephole** permettant d'éliminer les séquences de code inutiles, notamment celles impliquant des instructions `LOAD` redondantes ou superflues.
- La **propagation des constantes** pour éliminer les calculs redondants et éviter des opérations inutiles.
- La **propagation des copies** pour réduire les écritures et les lectures inutiles en mémoire.
- L'**élimination du code mort** en supprimant les instructions qui n'ont pas d'impact sur le résultat final du programme.
- Le **constant folding** qui permet de simplifier les expressions arithmétiques en des valeurs constantes avant l'exécution.
- La **dead store elimination** pour supprimer les écritures inutiles dans la mémoire, optimisant ainsi l'utilisation de la mémoire.
- Des **simplifications algébriques** pour réduire la complexité des calculs, rendant le code plus rapide à exécuter.

En plus de l'amélioration des performances en termes de vitesse et de taille du code, ces optimisations visent également à réduire la consommation énergétique du programme lors de son exécution. En effet, des programmes plus légers et plus efficaces demandent moins de puissance de calcul et de ressources matérielles, ce qui contribue à une réduction de l'empreinte énergétique. Cette approche est particulièrement pertinente dans un contexte de développement logiciel de plus en plus axé sur la durabilité et l'efficacité énergétique.

3 Spécification et Algorithmes

Cette partie présente la spécification détaillée de l'extension OPTIM du compilateur Deca, en mettant en évidence les différences notables entre la vision initiale et l'implémentation actuelle. Dès le début du projet, l'objectif était de concevoir un ensemble d'optimisations ambitieuses qui incluaient notamment la propagation de constantes, la détection de code mort, et la mise en œuvre d'une analyse complète du flot de contrôle (CFG). Cependant, à mesure que le projet avançait et que les contraintes liées au cadre pédagogique du compilateur Deca se faisaient plus claires, la portée de certaines optimisations a été ajustée pour rester pragmatique, tout en garantissant une sémantique correcte et une efficacité mesurable.

Ainsi, une grande importance a été accordée à la spécification du temps d'exécution des instructions de base.

Instruction	**Cycles**	**Instruction**	**Cycles**
LOAD	2	DIV	40
STORE	2	INT	4
LEA	0	BRA	5
PEA	4	Bcc	5 (vrai) 4 (faux)
PUSH	4	BSR	9
POP	2	RTS	8
NEW	16	DEL	16
ADD	2	RINT	16
SUB	2	RFLOAT	16
SHL	2	WINT	16
SHR	2	WFLOAT/X	16
OPP	2	R/WUTF8	16
MUL	20	WSTR	16
CMP	2	WNL	14
QUO	40	ADDSP	4
REM	40	SUBSP	4
FLOAT	4	TSTO	4
Scc	3 (v) 2 (f)	HALT	1
ERROR	1	SCLK	2
SETROUND_m	20	CLK	16
FMA	21		

Modes d'adressage	**Temps supplémentaires**
Rm	0
d(XX)	4
d(XX, Rm)	5
#d	2
etiq	2
"..."	2 × longueur

3.1 Vision initiale vs. Réalisation actuelle

Initialement, la spécification d'OPTIM prévoyait une approche globale des optimisations, avec un focus sur des techniques classiques telles que l'optimisation du flot de contrôle à l'aide d'un CFG complet,

l'application d'un SSA (Static Single Assignment) formel, et l'analyse de données interprocédurale. Cependant, plusieurs limitations ont été rencontrées lors de l'implémentation de ces techniques :

- **Analyse du flot de contrôle (CFG)** : La construction d'un CFG complet s'est avérée trop complexe et coûteuse, surtout dans un contexte pédagogique, avec des risques d'erreurs sémantiques importants. Par conséquent, une stratégie simplifiée a été adoptée, basée sur une analyse partielle de l'AST et l'élimination de structures triviales comme `if(false)` ou `while(false)`.
- **SSA complet** : L'idée d'utiliser un SSA complet a été abandonnée au profit d'une version simplifiée ("SSA-lite"), se concentrant uniquement sur la propagation de copies dans des séquences linéaires. Cette approche limite les calculs de dominance et évite un renommage global des variables.
- **Optimisations sur le code IMA** : Bien que le peephole optimization ait été envisagé sous une forme simple, l'implémentation finale est plus agressive, appliquant des optimisations sur les instructions de bas niveau et permettant des simplifications supplémentaires des séquences IMA.

Ainsi, cette spécification reflète l'évolution de l'extension, en répondant à des besoins pragmatiques tout en restant fidèle à l'objectif d'améliorer les performances du compilateur à travers des transformations optimisées mais sûres.

3.2 Techniques mises en œuvre

3.2.1 Peephole Optimization

La **Peephole Optimization** est une technique locale appliquée au code intermédiaire (IMA) qui, contrairement à l'approche classique limitée à une fenêtre fixe d'instructions consécutives, adopte ici une stratégie plus agressive. Grâce à une boucle while dans la méthode optimize, l'analyse explore le flux d'instructions bien au-delà de la contiguïté immédiate. Elle identifie des opportunités de simplification même lorsque les instructions cibles sont séparées par des opérations neutres n'interférant ni avec les registres, ni avec les adresses concernées. Cette capacité à ignorer le "bruit" intermédiaire permet de détecter des motifs redondants ou triviaux (comme un LOAD #0 suivi plus loin d'un ADD) qui échapperaient à une analyse de voisinage standard, garantissant ainsi un code généré nettement plus compact et performant.

```
public void optimize(IMAProgram imaProg){
    LinkedList<AbstractLine> lines = imaProg.getLines();
    do {
        // ... Veuillez consulter le projet pour le code complet et corrigé
        changed = false; // si non boucle while infinie
        ListIterator<AbstractLine> iterator = lines.listIterator();
        while(iterator.hasNext()){
        ...
        }
        }while(changed);}
```

Dans cette section, nous décrivons les principales techniques implémentées dans la passe de `Peephole Optimization`, accompagnées d'exemples concrets et d'une présentation du code Java responsable de leur mise en œuvre.

3.2.1.1 Simplifications algébriques

Une optimisation courante consiste à éliminer des opérations triviales telles que `ADD #0`, `SUB #0`, `MUL #1`, `DIV #1`, qui ne modifient pas la valeur des registres. Par exemple, les séquences suivantes :

```
ADD #0, R2
BOV overflow_error
```

```
SUB #0, R2
BOV overflow_error
```

peuvent être éliminées sans affecter la sémantique du code.

Code Java correspondant :

```
private void addSubCase(ListIterator<AbstractLine> iterator, Line lineL, Instruction instructior
    // Remplacer l'instruction d'addition par une instruction de copie
    BinaryInstructionDValToReg addIsnt = (BinaryInstructionDValToReg) instruction;
    Operand operand1 = addIsnt.getOperand1();

    if (isImmediateZero(operand1)){
        //System.out.println("Found ADD(SUB) #0 Ri, removing it.");
        iterator.remove();
        changed = true; // Supprimer l'instruction ADD(SUB) #0 Ri
        // Après avoir supprimé ADD(SUB) #0 Ri, on peut vérifier si l'instruction suivante est
        removeBOV(iterator);
    }
}
```

De même, pour le reste des opérations arithmétiques :

```
MUL #1, R1
BOV overflow_error
```

qui peut être éliminée, et :

```
MUL #0, R1
BOV overflow_error
```

qui peut être remplacée par:

```
LOAD #0 Ri
```

Code Java correspondant :

```
private void mulCase(ListIterator<AbstractLine> iterator, Line lineL, Instruction instruction){
    // Remplacer l'instruction de multiplication par une instruction de copie
    MUL mulInst = (MUL) instruction;
    Operand operand1 = mulInst.getOperand1();

    if (operand1 instanceof ImmediateInteger ){
        int value = ((ImmediateInteger)operand1).getValue();
        if (value == 1){
```

```
            iterator.remove(); // Supprimer l'instruction MUL #1 Ri
            // Après avoir supprimé MUL #1 Ri, on peut vérifier si l'instruction suivante est u
            changed = true;
            removeBOV(iterator);
        }
        else if (value == 0){
            // Remplacer MUL #0 Ri par LOAD #0 Ri ceci va permettre de detecter les peephole de
            Operand registerOp = mulInst.getOperand2();
            GPRegister register = (GPRegister) registerOp;

            Instruction loadZero = new LOAD(new ImmediateInteger(0), register);
            lineL.setInstruction(loadZero); // Remplacer l'instruction MUL #0 Ri par LOAD #0 Ri
            changed = true;
            // Après avoir remplacé par LOAD #0 Ri, on peut vérifier si l'instruction suivante
            removeBOV(iterator);
        }
    }
}
```

et

```
DIV #1, R1
BOV overflow_error
```

qui peut être élimnée aussi.

Code Java correspondant :

```
private void divCase(ListIterator<AbstractLine> iterator, Line lineL, Instruction instruction){
    // Remplacer l'instruction de division par une instruction de copie
    Operand operand1 = ((DIV)instruction).getOperand1();

    if (isImmediateOne(operand1)){
        iterator.remove(); // Supprimer l'instruction DIV #1 Ri
        // Après avoir supprimé DIV #1 Ri, on peut vérifier si l'instruction suivante est une B
        changed = true;
        removeBOV(iterator);
}
}
```

Code Java responsable de la suppression des BOV :

```
private void removeBOV(ListIterator<AbstractLine> iterator){
    if (!this.check){
        return; // Si les vérifications sont désactivées, ne rien faire
    }
    if (iterator.hasNext()){
        AbstractLine nextLine = iterator.next();
        if (nextLine.isLine()){
            Line nextLineL = (Line) nextLine;
```

```
            Instruction nextInstruction = nextLineL.getInstruction();
            // Vérifier si l'instruction suivante est une BOV
            if (nextInstruction instanceof BOV){
                iterator.remove(); // Supprimer l'instruction BOV inutile après LOAD #0 Ri
                changed = true;
            }
        }

        iterator.previous(); // Revenir à la position initiale
    }
}
```

3.2.1.2 Optimisation des instructions LOAD
L'Optimisation Globale des LOAD (Orchestration)

```
private void loadOptimize(ListIterator<AbstractLine> iterator, Line lineL, Instruction instructi
```

La méthode loadOptimize constitue le point d'entrée principal pour toutes les transformations impliquant l'instruction LOAD. Plutôt que de se limiter à une analyse locale immédiate, elle orchestre les différentes optimisations spécifiques en explorant le flux d'instructions à la recherche de motifs redondants.

Fonctionnement et Logique d'Exploration

Contrairement à une analyse rigide, cet orchestrateur parcourt les instructions suivantes tout en gérant intelligemment les "obstacles" potentiels. Voici les règles de gestion du flux qu'il applique :

- **Sauts et Étiquettes (Labels)** : Si l'optimiseur rencontre une étiquette (`label`), il interrompt immédiatement sa recherche. En effet, une étiquette signifie qu'un saut peut arriver de n'importe où dans le programme, rendant l'état des registres incertain à ce point précis.
- **Barrières de contrôle** : Toute instruction de branchement ou de contrôle de flot arrête l'exploration pour garantir la stabilité du programme.
- **Transparence des instructions neutres** : L'orchestrateur est capable de "sauter" certaines instructions (comme des lignes vides ou des portions de code spécifiques) pour trouver des optimisations plus loin dans le code.

Priorités et Délégation aux Sous-Modules

La méthode appelle successivement les modules spécialisés selon une hiérarchie logique :

1. **Élimination pure** : Elle vérifie d'abord si deux `LOAD` strictement identiques se suivent (`LOAD X, Ri` suivi de `LOAD X, Ri`). Le second est alors supprimé car redondant.
2. **Gestion des STORE intermédiaires** : Elle délègue à `storeLoadOptimize` si un `STORE` est rencontré. Si ce `STORE` n'utilise pas le registre en cours de chargement, l'optimiseur peut continuer sa recherche.

3. **Fusion et Substitution** : Elle tente ensuite de déléguer la transformation aux modules `loadRedundantOptimize` (substitution de registres) ou `loadRiRjOptimize` (transfert direct), permettant ainsi de simplifier des chaînes de dépendances complexes.

4. **Simplification Arithmétique** : Enfin, elle vérifie si le chargement concerne la valeur zéro pour potentiellement simplifier une addition future via `loadZeroAddOptimize`.

Garantie de sûreté (Rollback)

Un aspect crucial de cet orchestrateur est sa capacité de "rollback". Si l'exploration ne mène à aucune optimisation valide ou si elle rencontre une instruction qui modifie de manière inattendue les registres suivis, l'itérateur revient exactement à sa position initiale via une boucle de compensation (`iterator.previous()`). Cela assure que le reste du processus de compilation continue sur une base saine et non corrompue.

En résumé, loadOptimize ne se contente pas de nettoyer le code ; il analyse les relations de dépendance entre les registres et la mémoire sur une fenêtre glissante, maximisant ainsi les gains de performance sans jamais compromettre la sémantique du programme original. On explicite ci-dessous les différents piste d'optimisation orchestrées par cette méthode:

Élimination des LOAD redondants

Il s'agit de la suppression des instructions de type:

```
LOAD X, R1
...
LOAD X, R1
```

qui peut être factorisée en

```
LOAD X, R1 # deuxième LOAD
```

Cela permet de se débarrasser des LOAD redondants, qu'ils soient consécutifs ou non. Cette optimisation a été généralisée pour s'appliquer à tous les types de LOAD, sans se limiter aux valeurs immédiates, comme ça a été le cas dans une première implémentation. En effet, il n'est plus nécessaire que les instructions soient successives. Toutefois, l'optimisation repose sur l'hypothèse que les registres ou les adresses utilisés ne sont pas "touchés" entre les deux LOAD, ce qui garantit la sécurité de l'optimisation. Cette approche permet ainsi de réduire efficacement le nombre d'instructions tout en évitant les risques d'introduire des erreurs dues à des dépendances entre instructions.

Code Java correspondant :

```java
private boolean loadRedundantOptimize(ListIterator<AbstractLine> iterator, Line lineL, Line next
    Instruction nextInstruction = nextLineL.getInstruction();
   // ... Veuillez consulter le projet pour un le script complèt
    if ((nextValueOp instanceof GPRegister)) {return false;}
    if (valueOp instanceof GPRegister){ return false;}
    if (!nextValueOp.equals(valueOp)){ return false;}
    if (nextRegister.getNumber() != register.getNumber()) {return false; }
    int back = 0;
    while (iterator.hasPrevious() && back < consumedInst) {
        AbstractLine prev = iterator.previous();
        back++;
```

```
        if (prev == lineL) { continue;}
        if (!prev.isLine()) {continue;}
        Instruction inst = ((Line) prev).getInstruction();
        if (inst != null && instructionTouchesRegister(inst, register)) {
            for (int i = 0; i < back; i++) {iterator.next();}
            return false;
        }
    }
    boolean removed = false;
    while (iterator.hasPrevious()) {
        AbstractLine prev = iterator.previous();
        if (prev == lineL) {
            removed = true;
            iterator.remove();
            break;
        }}
    if (!removed){return false;}
    changed = true;
     return true;
}
```

Auto LOAD

Il s'agit de la suppression des instruction de type:

```
LOAD Ri, Ri
```

L'optimisation "Auto LOAD" consiste à supprimer les instructions redondantes où un registre est chargé dans lui-même. Ces instructions sont inutiles car elles ne modifient pas l'état du programme et n'ont aucun effet sur les registres. Par conséquent, elles peuvent être supprimées pour réduire la taille du code et améliorer les performances sans altérer la sémantique du programme. **Code Java correspondant** :

```
private boolean loadRiRiOptimize(ListIterator<AbstractLine> iterator, Line lineL, Instruction ir
    if (valueOp instanceof GPRegister){
        GPRegister registerValue = (GPRegister) valueOp;
        if (register.getNumber() == registerValue.getNumber()){
            iterator.remove();
            changed = true;
            return true;
        }
    }
    return false;
}
```

Fusion de LOAD

Il s'agit du remplacement d'une séquence d'instructions de type :

```
LOAD X, Ri
....
LOAD Ri, Rj
```

par une seule instruction :

```
LOAD X, Rj
```

Cette optimisation consiste à détecter un chargement d'une valeur (mémoire ou constante) dans un registre R_i, suivi immédiatement (ou après quelques instructions neutres) par un transfert de ce registre R_i vers un second registre R_j.

Le code vérifie rigoureusement qu'aucune instruction intermédiaire entre le premier LOAD et le second ne modifie ou n'utilise le registre R_i afin de garantir la sûreté du remplacement. En fusionnant ces deux étapes, on élimine une instruction de transfert de registre à registre. Cela permet de réduire le nombre de cycles d'exécution (gain de 2 cycles selon votre table de référence) et de libérer potentiellement le registre R_i plus tôt dans le flux du programme.

Code Java correspondant : Le principe de cet algorithme s'apparente à celui de la détection de charges redondantes (*Load Redundant Optimization*). Par souci de concision, nous ne présentons ici que la logique d'optimisation :

```java
private boolean loadRiRjOptimize( ListIterator<AbstractLine> iterator, Line lineL, Line nextLine
   //... Veuillez consulter le projet pour le script complet
    ....
    DVal dst = (DVal) valueOp;
    nextLineL.setInstruction(new LOAD(dst, nextRegister));
    // Suppression du LOAD initial
    while (iterator.hasPrevious()) {
        AbstractLine prev = iterator.previous();
        if (prev == lineL) {
            iterator.remove();
            break;
        }
    }
    changed = true;
    return true;
}
```

Simplification d'addition

Il s'agit du remplacement d'une séquence d'instructions de type :

```
LOAD #0, Ri
...
ADD X, Ri
```

par une seule instruction :

```
LOAD X, Ri
```

L'optimisation "LOAD Zero ADD" détecte les cas où un registre R_i est initialisé à zéro juste avant de se voir ajouter une valeur X (qu'il s'agisse d'une constante, d'une adresse ou d'un autre registre). Mathématiquement, l'opération revient à effectuer 0+X, ce qui est équivalent à un simple chargement de X dans R_i.

En remplaçant l'addition par un LOAD, on gagne en efficacité de deux manières :

1. **Réduction du cycle d'horloge** : On supprime totalement l'instruction d'initialisation (LOAD #0).

2. **Suppression des effets de bord** : L'instruction ADD peut lever une exception de débordement (Overflow). Le code inclut donc un appel à `removeBOV(iterator)` pour supprimer l'instruction de branchement sur débordement (`BOV`) qui suit généralement l'addition, car un simple chargement ne peut pas provoquer d'overflow.

La gestion des instructions intermidiare se fait directeent à partir du programme principal de l'optimisation load (**loadOptimize**). **Code Java correspondant** :

```
private boolean loadZeroAddOptimize(ListIterator<AbstractLine> iterator, Line lineL, Line nextLi
    //... Veuillez consulter le projet pour le script complet et commenté ( on ne présente ici 
    Operand src = addIsnt.getOperand1();
    DVal srcD = (DVal) src;
    nextLineL.setInstruction(new LOAD(srcD, registerDest));
    changed = true;
    removeBOV(iterator); // supprimer le BOV inutile s'il y en a
    while(iterator.hasPrevious()){
        AbstractLine previousLine = iterator.previous();
        if (previousLine == lineL){
            iterator.remove();
            return true;
        }
    }

    return true;
}
```

Voici la description de cette optimisation pour votre document RMarkdown. Cette règle s'attaque à un motif très courant généré par les compilateurs non optimisés lors de la manipulation de variables globales ou locales.

Élimination de l'accès mémoire redondant

Il s'agit de la suppression d'une séquence d'instructions de type :

```
STORE Ri, addr
LOAD addr, Ri
```

Cette optimisation ne s'applique que lorsque le `LOAD` suit immédiatement le `STORE`. Cette contrainte de proximité directe est essentielle pour garantir la stabilité et la cohérence des données pour les raisons suivantes :

- **Absence d'effets de bord** : Si d'autres instructions s'intercalaient entre le stockage et le chargement, elles pourraient modifier soit le contenu du registre R_i, soit la valeur à l'adresse mémoire `addr`. En limitant l'optimisation à des instructions consécutives, on s'assure qu'aucune perturbation n'a pu modifier l'état du processeur entre les deux opérations.

- **Sûreté de l'optimisation** : Le code vérifie que le registre et l'adresse sont strictement identiques. Dans ce contexte précis, le `LOAD` ne fait que confirmer une valeur que le processeur

possède déjà dans ses registres. Sa suppression est donc sémantiquement neutre mais physiquement bénéfique.

L'impact sur les performances est double : on économise 4 cycles au total (2 pour le `STORE` et 2 pour le `LOAD`), tout en réduisant l'utilisation du bus mémoire.

Code Java correspondant :

```
private void storeLoadOptimize(ListIterator<AbstractLine> iterator, Line lineL, Instruction inst
    STORE storeInst = (STORE) instruction;
    // ... Veuillez consulter le projet pour le script complet et commenté
    if (iterator.hasNext()){
    ....
        DAddr addrStore = (DAddr) storeInst.getOperand2();
        if (!(nextInstruction instanceof LOAD)){
            iterator.previous();
            return;
        }
        LOAD loadInst = (LOAD) nextInstruction;
        .....
        DAddr addrLoad = (DAddr) loadInst.getOperand1();

        // Vérifier si il s'agit du meme registre: si c'est le cas ces 2 instructions sont inut
        if (registerLoad.getNumber() == registerStore.getNumber() && (addrLoad.equals(addrStore)
            iterator.remove();
            iterator.previous();
            iterator.remove();
            iterator.next();
            changed = true;
            return;
        }
        iterator.previous();
    }}
```

3.2.1.3 Optimisation des STORE Redondants

Il s'agit de la suppression des instruction de type:

```
STORE R1, X
STORE R2, X
```

qui peut être factorisée en :

```
STORE R2, X
```

En effet, la première instruction `STORE R1, X` est inutile, car la valeur qu'elle écrit en mémoire est immédiatement remplacée par la seconde écriture. La suppression du premier `STORE` ne modifie donc pas la sémantique du programme.

Contrairement à l'optimisation des `LOAD` redondants, cette optimisation a volontairement été limitée à des séquences strictement consécutives. En effet, généraliser cette transformation à des cas non adjacents impliquerait une analyse plus fine des accès mémoire intermédiaires (aliasing, lectures implicites,

effets de bord), ce qui augmenterait significativement la complexité de l'implémentation et le risque d'introduire des erreurs sémantiques.

Ce choix reflète une approche conservatrice : privilégier des optimisations simples, sûres et locales, tout en garantissant la correction du code généré.

Code Java correspondant :

```
private void storeRedundantOptimize(ListIterator<AbstractLine> iterator, Line lineL, Instructior
    // ... Veuillez consulter le projet pour le script complet et commenté
    ....
    STORE nextStore = (STORE) nextInstruction;
    DAddr addrNext = (DAddr) nextStore.getOperand2();
    if (!addrStore.equals(addrNext)) {
        iterator.previous();
        return ;
    }
    iterator.previous(); // revenir sur le second
    iterator.previous(); // revenir sur le premier
    iterator.remove();   // supprimer le premier
    iterator.next();     // revenir après le second
    changed = true;
}
```

3.2.1.4 Les Fonctions Utilitaires de Sécurité

L'optimiseurPeephole s'appuie sur trois fonctions clés pour garantir que chaque transformation est sémantiquement neutre (le code est plus rapide, mais le résultat reste inchangé).

1. Identification des éléments neutres Les méthodes `isImmediateZero` et `isImmediateOne` repèrent les constantes qui n'altèrent pas le résultat d'une opération (ex: x+0). Elles permettent de déclencher les simplifications arithmétiques les plus simples sans calcul supplémentaire.

2. Analyse d'interférence (`instructionTouchesRegister`) C'est le pilier de la sécurité. Cette fonction détermine si une instruction lit ou écrit dans un registre donné.

Elle vérifie les opérandes directs, les registres d'indexation (offset) et les lectures implicites (comme la convention IMA où WINT lit systématiquement R1).

Rôle : Elle interdit toute optimisation si une instruction intermédiaire a besoin de la valeur du registre ou menace de la modifier.

3. Détection des barrières de contrôle (`instructionIsControlOrBranche`) Cette méthode identifie les ruptures de flux : sauts (BRA, ...), appels (BSR), retours (RTS) ou arrêt (HALT).

Rôle : Elle définit une frontière infranchissable. Dès qu'une telle instruction est détectée, l'optimiseur arrête son exploration car le chemin d'exécution devient imprévisible.

3.2.2 Propagations AND Dead STORE's

Alors que la `Peephole Optimization` se concentre sur des fenêtres d'instructions locales, la `Constant Propagation` et la `Copy Propagation` visent à suivre le flux des données à travers les registres et la mémoire. Ces deux passes travaillent de concert pour éliminer les redondances introduites lors de la génération de code, notamment lors des accès répétés aux variables.

3.2.2.1 Constant Propagation

L'objectif est de remplacer les lectures en mémoire (`LOAD addr, Ri`) par des chargements immédiats (`LOAD #k, Ri`) lorsque la valeur à l'adresse cible est connue statiquement.
Exemple de cas d'optimisation valide
Code Deca original:

```
{
    int a = 3;
    int b = a + 1;
}
```

Code IMA non optimisé :

```
LOAD #3, R2
STORE R2, 1(GB)  ; a = 3
LOAD 1(GB), R2   ; charger a
ADD #1, R2       ; a + 1
STORE R2, 2(GB)  ; b = 4
```

Code IMA après Constant Propagation :

```
LOAD #3, R2
STORE R2, 1(GB)
LOAD #3, R2      ; Propagation : 1(GB) est connu comme valant 3
ADD #1, R2
STORE R2, 2(GB)
```

Et c'est ici où intervient le **DeadStoreElimination** pour nettoyer les stores inutiles qui résultent de cette propagation. et donne:

```
LOAD #3, R2
LOAD #3, R2
ADD #1, R2
STORE R2, 2(GB)
```

Fonctionnement : L'optimiseur maintient deux tables de hachage durant son parcours linéaire :

- `whoAmI` : Associe une adresse mémoire à une valeur constante.
- `whatRAmI` : Associe un registre à une valeur constante.

Dès qu'un `STORE #k, x` ou un `LOAD #k, Ri` est rencontré, l'optimiseur "apprend" la constante. Si un `LOAD x, Rj` survient plus tard, il est remplacé par `LOAD #k, Rj`, économisant ainsi un accès mémoire (gain de 2 cycles). Pour garantir la correction, toute instruction modifiant un registre ou une adresse (barrière de contrôle, étiquette, ou opération binaire) entraîne la suppression de l'entrée correspondante dans les tables.

Extraits du Code Java correspondant :
Rencontre d'un Label :

```
if (lineL.getLabel() != null) {
      whoAmI.clear();
      whatRAmI.clear();
      continue;
  }
```

Rencontre d'un Branchement où Barrière:

```
if (instructionIsControlOrBranche(instruction) || lineL.getLabel() != null) {
    whoAmI.clear();
    whatRAmI.clear();
    continue;
}
```

En raison de l'absence d'un graphe de flot de contrôle (CFG), notre analyse est strictement linéaire. Cette approche impose une règle de sécurité critique : l'invalidation systématique des données en cas de rupture de flux.

Pour garantir la sûreté de l'optimisation, l'optimiseur réinitialise ses tables de hachage (clear()) dans les deux cas suivants :

- **Rencontre d'un Label :** Un saut peut arriver de n'importe quel point du programme, rendant l'état des registres et de la mémoire imprévisible à cet endroit.
- **Branchements et Appels (BRA, BSR, RTS, etc.) :** Ces instructions rompent la continuité du parcours. Ne pouvant prédire l'état du processeur après ces sauts, l'optimiseur "laisse tomber" toutes les optimisations accumulées jusqu'à cet instant.

L'invalidation des registres à la rencontre des BinaryInstructions

```
if ((instruction instanceof BinaryInstructionDValToReg)) {

    BinaryInstructionDValToReg bin =
            (BinaryInstructionDValToReg) instruction;

    GPRegister register = (GPRegister) bin.getOperand2();
    whatRAmI.remove(register.getNumber());
    continue;
}
```

Entre les étapes de chargement (LOAD) et de stockage (STORE), l'optimiseur doit surveiller toute modification des registres pour éviter de propager des valeurs périmées.

En architecture IMA, un registre est "écrasé" dès qu'il est la destination d'une opération binaire (comme ADD, SUB, MUL, etc.). Le code implémente donc une règle de sécurité stricte :

- **Action :** Si une instruction de type BinaryInstructionDValToReg est rencontrée, le registre de destination est immédiatement retiré de la table whatRAmI.

- **Raison :** Une opération arithmétique change la valeur du registre. Si l'on ne supprimait pas cette entrée, l'optimiseur continuerait de propager l'ancienne constante, ce qui corromprait la sémantique du programme.

Rencontre d'un LOAD :

```
if (instruction instanceof LOAD) {
    \\ ... voir code pour la totalité du code avec les commentaire explicatifs.
    int registerNum = register.getNumber();
    if (valueOp instanceof ImmediateInteger) {
        whatRAmI.put(registerNum, (ImmediateInteger) valueOp);
        continue;
    }
    if (valueOp instanceof DAddr) { /* LOAD x, Ri */
        DAddr variable = (DAddr) valueOp;
        ImmediateInteger constant = whoAmI.get(variable);
        if (constant != null) {
            lineL.setInstruction(new LOAD(constant, register));
            whatRAmI.put(registerNum, constant);
        } else {whatRAmI.remove(registerNum);}
        continue; }
    whatRAmI.remove(registerNum);
    continue; }
```

La méthode traite les instructions LOAD pour apprendre ou propager des constantes :

- **Apprentissage (Immédiats)** : Si l'instruction est LOAD #k, Ri, l'optimiseur enregistre dans whatRAmI que le registre Ri contient désormais la constante k.
- **Propagation (Adresses) :** Si l'instruction charge une variable (LOAD x, Ri), le code vérifie si x possède une valeur connue dans whoAmI. Si c'est le cas, il réécrit l'instruction en chargement immédiat.
- **Sécurité et Adressage :** L'optimisation se limite aux adresses directes (DAddr). Les adressages indirects complexes (RegisterOffset) sont ignorés pour éviter les problèmes d'aliasing et les limitations liées aux instances d'objets en Java, garantissant ainsi la stabilité de l'analyse.
- **Invalidation :** Si la valeur chargée est inconnue, le registre est immédiatement marqué comme invalide pour éviter toute propagation erronée.

Rencontre d'un STORE :

```
if (instruction instanceof STORE) {
    // ... Veuillez consulter le projet pour le script complet et commenté
    ImmediateInteger constant = whatRAmI.get(registerNum);
    if (constant != null) { whoAmI.put(variable, constant);// La variable devient constante}
    else {whoAmI.remove(variable);// Valeur inconnue faut se débarasser d'elle }
    continue;}
```

L'instruction STORE permet de lier une valeur contenue dans un registre à une adresse mémoire. L'optimiseur utilise ce moment pour mettre à jour ses connaissances :

- **Mémorisation :** Si le registre source (Ri) est connu pour contenir une constante (via whatRAmI), cette constante est associée à l'adresse mémoire de destination dans whoAmI. Cela permet de propager la valeur vers de futurs LOAD depuis cette même adresse.
- **Invalidation par précaution :** Si le contenu du registre est inconnu, l'optimiseur supprime toute constante précédemment associée à cette adresse mémoire. Cette étape est cruciale pour éviter de propager une valeur périmée après que la variable a été modifiée par une donnée dynamique.

3.2.2.2 Copy Propagation

La Copy Propagation est conceptuellement identique à la Constant Propagation : elle vise à suivre le flux des données pour éliminer les redondances. La différence fondamentale réside dans l'élément propagé : au lieu de constantes, l'optimiseur suit les alias de variables (adresses mémoire) pour réduire les chaînes d'affectations.

Exemple de cas d'optimisation valide

Code Deca original :

```
{
    int a;
    int b;
    int c;

    a = 1;
    b = a;
    c = b;
    println(c);
}
```

Code IMA non optimisé :

```
LOAD #1, R2
STORE R2, 3(GB)
LOAD 3(GB), R2
STORE R2, 4(GB)
LOAD 4(GB), R2
STORE R2, 5(GB)
LOAD 5(GB), R1
WINT
```

Code IMA après Copy Propagation :

```
LOAD #1, R2
STORE R2, 3(GB)
LOAD #1, R2
STORE R2, 4(GB)
LOAD #1, R2
STORE R2, 5(GB)
LOAD #1, R1
WINT
```

Comme pour la propagation de constantes, le DeadStoreElimination intervient ensuite pour supprimer les `STORE` s'il devient inutile, aboutissant à :

```
LOAD #1, R2
LOAD #1, R1
WINT
```

Fonctionnement : L'optimiseur maintient deux tables de hachage traduisant les relations d'alias :

- `whoAmI` : Associe une adresse mémoire à une autre adresse source (`b` → `a`).
- `whatRAmI` : Associe un registre à une adresse source (`Ri` → `a`).

Extraits du Code Java correspondant :
Voir la grande similarité avec `constantPropagation` on ne cite ici que les extraits qui font la différence avec ce dernier.

Rencontre d'un LOAD :

```java
if(instruction instanceof LOAD){
    // .. voir le code dans le projet pour la totalité de cet extrait et les commentaires explic
    int registerNum = register.getNumber();
    if (!(valueOp instanceof DAddr)){
        whatRAmI.remove(registerNum);
        continue;
    }
    else{
        DAddr variableB = (DAddr) valueOp;
        DAddr variableA = whoAmI.get(variableB);
        if (variableA != null){
            lineL.setInstruction(new LOAD(variableA, register));
            whatRAmI.put(registerNum, variableA); // et on met le registre à jour
        }
        else{whatRAmI.put(registerNum, variableB); }
        continue;}}
```

Le but ici est d'identifier si une variable chargée est en réalité une copie d'une autre, afin de remonter directement à la source originale.

- **1.** Filtrage des opérandes L'optimiseur ne s'intéresse qu'aux chargements depuis la mémoire (DAddr). Si le LOAD concerne une valeur immédiate (ex: #10), l'association actuelle du registre est supprimée (whatRAmI.remove) car il ne contient plus l'image fidèle d'une variable.
- **2.** Substitution par la source (Le "Chain-Link") Si l'on charge une variable B, le code consulte la table whoAmI. Si B est connue comme une copie de A : L'instruction est réécrite dynamiquement pour charger A au lieu de B. Le registre R_i est alors marqué comme contenant A. Exemple : Si le code a fait b = a, alors LOAD b, R1 devient LOAD a, R_1.

- **3.** Apprentissage de l'état Si aucune relation de copie n'est connue pour la variable chargée, l'optimiseur enregistre simplement dans whatRAmI que le registre R_i contient désormais la valeur de cette variable. Cela servira de base pour de futures propagations si ce registre est plus tard stocké ailleurs (STORE).

Rencontre d'un STORE :

```
if (instruction instanceof STORE){
    // ... Veuillez consulter le programme dans le projet pour un script plus détaillé
    DAddr variableB = (DAddr) valueOp;
    DAddr variableA = whatRAmI.get(registerNum);
    if (variableA != null){
        // meme registre a été utilisé donc c'est propagé:
        whoAmI.put(variableB, variableA);
        // hors context : je suis en train d'entendre S.T.A.Y de Hans Zimmer si ça vous intéress
    }

    else{
        whoAmI.remove(variableB);
    }
    continue;
}
```

Ce bloc de code est responsable de l'enregistrement des relations d'équivalence entre deux variables en mémoire. C'est ici que l'optimiseur "apprend" qu'une variable est devenue la copie d'une autre.

- **1.** Identification du transfert Lors d'une instruction STORE Ri, b, l'optimiseur regarde quel est le contenu logique du registre Ri via la table whatRAmI. Si le registre Ri contient déjà la valeur d'une variable a (mémorisée lors d'un LOAD précédent), alors on en déduit l'affectation logique : b = a.
- **2.** Enregistrement de la copie Cas positif : Si une source variableA est trouvée, on crée un alias dans la table whoAmI (whoAmI.put(variableB, variableA)). Toute utilisation future de b pourra être remplacée par a. Cas négatif : Si le registre contient une valeur inconnue ou calculée, toute ancienne relation concernant b est supprimée (whoAmI.remove(variableB)). Cela garantit que si b est modifié, on ne propage plus son ancienne valeur.
- **3.** Sécurité de la référence L'affectation b = a casse les propagations précédentes de b. Si b était auparavant une copie d'une autre variable x, cette relation est écrasée par la nouvelle source a, assurant la cohérence du flux de données.

3.2.2.3 Dead STORE's Elimination (DSE)

Le DeadStoreOptimizer agit comme une phase de nettoyage final. Son rôle est de supprimer les instructions `STORE` dont la valeur écrite en mémoire n'est jamais relue (écritures "mortes"). Cette passe est l'alliée indispensable des **propagations précédentes (copy et constant Propagation)** : en remplaçant des accès mémoire par des immédiats ou des registres, la propagation "tue" la vivacité de certaines adresses, permettant à la DSE de les supprimer.

Algorithme par Blocs de Base

Pour minimiser le coût de compilation et garantir la cohérence, l'optimiseur procède en deux étapes :

1. **Segmentation en blocs :** Le programme est découpé en blocs linéaires. Toute étiquette (`Label`) ou instruction de contrôle (`BRA`, `BSR`, etc.) marque la fin d'un bloc.
2. **Analyse de vivacité rétrograde :** Chaque bloc est parcouru de la fin vers le début (`backward analysis`). On maintient un ensemble `live` des adresses mémoires lues par un `LOAD`.

Construction des Blocks (Extraits Java - Methode optimize de DeadStoreOptimize) :

```java
// ... Veuillez consulter le programme dans le projet pour un script plus détaillé
for (AbstractLine line : lines) {
    if (!line.isLine()) {
        idx++;
        continue;
    }
    Line lineL = (Line) line;
    Instruction instruction = lineL.getInstruction();
    if (lineL.getLabel() != null && idx != start) {
        start = idx;
    }
    if (instruction != null && instructionIsControlOrBranche(instruction)) {
        blocks.add(new int[]{start, idx});
        start = idx + 1;
    }
    idx++;
}
if (start < lines.size()) {
    blocks.add(new int[]{start, lines.size() - 1});
}
```

Logique de suppression (Extraits Java) :

```java
//... Veuillez consulter le programme dans le projet pour un script plus détaillé
if (instruction instanceof STORE) {
    STORE store = (STORE) instruction;
    Operand op2 = store.getOperand2();

    if (!(op2 instanceof DAddr)) {
        continue; // prudence
    }
    DAddr storeAddr = (DAddr) op2;
    if (!live.contains(storeAddr)) {
        iterator.remove();
    } else {
        live.remove(storeAddr);
    }
    continue;
}
```

Sécurité : L'optimisation ne s'applique qu'aux blocs terminaux (finissant par `HALT`, `RTS` ou `ERROR`). Sans cette précaution, on risquerait de supprimer une écriture nécessaire à un autre bloc invisible depuis l'analyse locale.

3.2.2.4 Limitations : L'impact de l'Orienté Objet

Si nos **Propagation** et notre **DSE** sont performants sur les types élémentaires (`int`, `float`,..), leur efficacité devient très restreinte dans un contexte Avec Objet pour plusieurs raisons :

- **Multiplicité des barrières :** L'utilisation massive de méthodes entraîne de nombreux `BSR` (appels), `RTS` (retours), et les variables Global qui deviennet dépendantes du comportement de nos classes. Ces instructions sont des "boîtes noires" qui forcent l'optimiseur à invalider toutes ses connaissances par sécurité.
- **Changement de référence vs Copie :**
  - Pour les types élémentaires, une affectation est une copie de valeur : le lien est direct et facile à suivre.
  - Pour les objets, l'affectation est un partage de référence (aliasing). Plusieurs variables peuvent pointer vers le même objet en tas. Modifier un champ via une référence invalide potentiellement toutes les autres, ce qui rend la propagation de constantes sur les attributs d'objets extrêmement complexe sans une analyse de pointeurs globale.
- **Densité des étiquettes :** La structure des méthodes et de l'héritage génère un grand nombre de labels, ce qui fragmente le code en blocs très courts, réduisant la fenêtre de tir de l'optimiseur linéaire.

Donc la structure complexe des programmes orientés objet rend les optimisations de propagation et de suppression de stores presque inopérantes. Dans un tel contexte, tenter d'optimiser le code IMA consommerait des ressources de calcul pour un gain de performance quasi nul.

Dans une démarche de conception logicielle responsable, nous avons décidé que le DecacCompiler ne lancerait ces passes que sur les programmes dits "sans objets" (prog.getIsOOP() == false). Ce choix repose sur deux piliers :

- **Efficacité Computationnelle :** Éviter d'exécuter des algorithmes de parcours de listes et des analyses de blocs sur des codes où les barrières de flux (RTS, BSR) sont trop denses pour permettre des réductions significatives.
- **Responsabilité Environnementale :** En désactivant des passes inutiles dans le cas OOP, nous réduisons l'empreinte carbone de la phase de compilation, évitant ainsi une consommation énergétique en vain pour un résultat final identique.

3.2.3 Optimisations sur l'Arbre de Syntaxe Abstraite (AST)

Avant la génération du code IMA, le système d'optimisation effectue des transformations de haut niveau directement sur l'Arbre de Syntaxe Abstraite (AST). Ces passes permettent de simplifier la logique du programme en évaluant tout ce qui peut l'être statiquement, réduisant ainsi la charge de travail des étapes suivantes.

3.2.3.1 Constant Folding (ConstantFolder) L'optimisation **Constant Folding** (ou « repliement de constantes ») a pour objectif de réduire les expressions constantes dès la compilation. En évaluant les expressions dépendant uniquement de littéraux, on diminue le nombre d'instructions générées et, par extension, le nombre de calculs à effectuer lors de l'exécution. Cette passe est orchestrée par la classe ConstantFolder qui réalise une traversée de l'AST.

Principe de fonctionnement Le mécanisme repose sur la méthode **foldConstants()**, dont l'implémentation est distribuée de manière récursive sur les différents nœuds de l'arbre (fichiers sources des classes héritant d' **AbstractExpr**).

- Traversée récursive : Chaque nœud de l'AST tente de s'auto-évaluer en fonction des valeurs de ses enfants. Par exemple, un nœud Plus appellera **foldConstants()** sur ses opérandes gauche et droit avant de tenter de produire un unique nœud IntLiteral ou FloatLiteral représentant leur somme.
- Itération multi-passes : La méthode **apply()** du compilateur appelle cette traversée comme c'est expliqué ci-dessous en **Fonctionnement**.

Fonctionnement :

- **Itération multi-passes :** La méthode `apply()` parcourt le programme jusqu'à 20 fois (`MAX_PASSES`). Cette approche itérative est nécessaire pour gérer les dépendances successives.

  Exemple : $(2 + 3) + 4 \rightarrow 5 + 4 \rightarrow 9$
- **Évaluation des littéraux** : Les nœuds `IntLiteral`, `FloatLiteral` et `BooleanLiteral` sont évalués pour réduire les opérations binaires ou unaires.

  Exemple : `true && false` $\rightarrow$ `false`
- **Conversion explicite (ConvFloat) :** Si une opération combine un entier et un flottant, la classe `ConvFloat` transforme le littéral entier en `FloatLiteral`.

  Exemple : `float f = 3` $\rightarrow$ `3.0f`

  Remarque : cette optimisation va impliquer que le noeud ConvFloat soit supprimé de l'AST, ce qui est cohérent avec sa nature et n'affecte pas le résultat final. `3.0f`
- **Sécurité :** Les cas critiques comme la division par zéro sont gérés via `canFoldInt()` et `canFoldFloat()`, empêchant le repliement de l'expression pour laisser l'erreur se produire (ou non) à l'exécution.

Exemple d'un cas d'optimisation:

```
{
    // Calcul du nombre de secondes dans 2 heures
    int s = 2 * 60 * 60;
}
```

Exemple du code IMA avant opitimisation:

```
; --- Initialisation de s ---
LOAD #2, R2        ; Charger 2
MUL #60, R2        ; R2 = 2 * 60 (120)
MUL #60, R2        ; R2 = 120 * 60 (7200)
STORE R2, 1(GB)    ; Stocker le résultat dans s
```

Après optimisation:

```
; --- Initialisation de s (Optimisée) ---
LOAD #7200, R2     ; La constante a été calculée à la compilation
STORE R2, 1(GB)    ; Stockage direct
```

Quelques Scripts Java pour la démonstration :
Folding :
Dans Divide.java

```java
@Override
protected boolean canFoldInt(int a, int b) { return b != 0; }
```

```java
@Override
protected boolean canFoldFloat(float a, float b) { return b != 0.0f; }
```

Sécurité :
Dans ListInst.java

```java
public boolean foldConstants(DecacCompiler compiler) {
    boolean changed = false;

    for (int i = 0; i < size(); i++) {
        AbstractInst inst = getModifiableList().get(i);  // getModifiableList().get(i);

        // Si l'instruction est en réalité une expression (ex: Assign; Plus; appel; etc.)
        if (inst instanceof AbstractExpr) {
            AbstractExpr oldE = (AbstractExpr) inst;
            AbstractExpr newE = oldE.foldExpr(compiler);

            if (newE != oldE) {
                set(i, newE);        //
                changed = true;
            }
        } else {
            //
            changed |= inst.foldConstants(compiler);
        }
    }
    return changed;
}
```

Dans ConvFloat.java:

```java
@Override
public AbstractExpr foldExpr(DecacCompiler compiler) {
    AbstractExpr inner = getOperand().foldExpr(compiler);
    if (inner != getOperand()) setOperand(inner);

    if (inner instanceof IntLiteral) {
        FloatLiteral res = new FloatLiteral((float) ((IntLiteral) inner).getValue());
        res.setType(getType());
        res.setLocation(getLocation());
        return res;
    }
    return this;
}
```

3.2.3.2 Dead Code Elimination (DCE)

La passe Dead Code Elimination (DCE) supprime les parties inaccessibles ou inutiles du code. Elle s'exécute après le Constant Folding, car les simplifications de constantes révèlent souvent du code mort (`ex: un if (false) résultant d'un calcul`).

Sous-optimisations coordonnées La classe DeadCodeEliminator applique trois transformations de front :

- **Élimination du flot de contrôle mort :** Suppression des instructions situées après un return ou dans des branches de conditionnels dont le résultat est connu (if(false), while(false)).
- **Élimination des affectations inutilisées (Dead Stores) :** Les assignations à des variables jamais lues sont supprimées.
- **Gestion des effets de bord :** Si l'expression est "impure" (`ex: int x = f(); où x est inutile`), la variable x est supprimée mais l'appel f() est conservé et réinséré pour garantir l'exécution de ses effets de bord.
- **Suppression des variables inutilisées :** Les déclarations de variables locales qui ne subissent aucune lecture sont retirées de l'AST.

Les nœuds de l'AST (Classes héritant d' AbstractInst)

Le travail de suppression réel est délégué aux nœuds qui possèdent des blocs d'instructions. Ces classes surchargent ou utilisent la logique d'élimination :

- **ListInst (ou Block) :** C'est la classe la plus critique. C'est elle qui parcourt sa propre liste d'instructions et retire physiquement les nœuds inutiles (comme après un return).
- **IfThenElse :** Utilise la logique pour supprimer une branche entière (le then ou le else) si la condition est devenue un BooleanLiteral constant.
- **While :** Utilise eliminateDeadCode pour supprimer la boucle complète si la condition est false.
- **AbstractMain et AbstractMethodBody :** Utilisent la méthode pour nettoyer les blocs principaux du programme et des méthodes.

Processus global et convergence
Le processus est itératif (limité à 30 passages) pour atteindre un point fixe :

- **1.** Collecte des variables lues/écrites.
- **2.** `eliminateDeadCode()` → `eliminateDeadStores()` → `eliminateUnusedVars()`.
- **3.** Répétition tant que l'AST subit des modifications.

Exemple d'un cas d'optimisation:

```
{
    int x = 10;
    if (1 > 2) {
        x = 5; // Code mort (1 n'est jamais > 2)
    }
    print(x);
}
```

Exemple du code IMA avant optimisation:

```
LOAD #10, R2     ; Initialisation de x
STORE R2, 1(GB)
LOAD #1, R2      ; Début du IF
CMP #2, R2
BLE end_if       ; Si 1 <= 2, on saute le bloc
LOAD #5, R2      ; Branche TRUE
STORE R2, 1(GB) ; Cette instruction est générée inutilement
end_if:
LOAD 1(GB), R1  ;Fin du IF
WINT
```

Après optimisation:

```
LOAD #10, R2      ; Initialisation de x
STORE R2, 1(GB)
; --- Le bloc IF a été totalement supprimé de l'AST ---
LOAD 1(GB), R1    ; Print de x
WINT
```

Quelques Scripts Java pour la démonstration :
Elimination :
Dans ListInst.java

```
protected boolean eliminateDeadCode(DecacCompiler compiler) {
    boolean changed = false;
    List<AbstractInst> insts = getModifiableList();
    for (int i = 0; i < insts.size(); i++) {
        AbstractInst inst = insts.get(i);
        // return => tout ce qui suit est mort
        if (inst instanceof Return) {
```

```
            int oldSize = insts.size();
            insts.subList(i + 1, oldSize).clear();
            if (insts.size() != oldSize) changed = true;
            break;}
        // while(false) => supprime la boucle sans analyser le body
        if (inst instanceof While) {
            While w = (While) inst;
            AbstractExpr cond = w.getCondition();
            if (cond instanceof BooleanLiteral && !((BooleanLiteral) cond).getValue()) {
                insts.remove(i);
                changed = true;
                i--;
                continue;}}
        // if(true/false) => remplace par then/else
        if (inst instanceof IfThenElse) {
            .....
                continue;}}
        changed |= inst.eliminateDeadCode(compiler);}
    return changed;}
```

Dans IfThenElse.java:

```
@Override
protected boolean eliminateDeadCode(DecacCompiler compiler) {
    boolean changed = false;
    changed |= thenBranch.eliminateDeadCode(compiler);
    if (elseBranch != null) { changed |= elseBranch.eliminateDeadCode(compiler);}
    return changed;}
```

4 Conception et Architecture :

4.1 Organisation logicielle et Localisation :

L'intégralité des modules d'optimisation est regroupée dans le package `fr.ensimag.deca.optim` (répertoire `src/main/java/`). Cette centralisation facilite la maintenance et l'activation modulaire des passes au sein du `DecacCompiler`.

L'architecture se décline en deux niveaux distincts :

1. **Optimisations de haut niveau (AST) :** Les classes `ConstantFolder` et `DeadCodeEliminator` opèrent directement sur l'Arbre de Syntaxe Abstraite. Comme détaillé précédemment, leur logique repose sur une **distribution récursive** : les méthodes de réduction (telles que `foldConstants()`) sont implémentées à travers les différents nœuds de l'AST, permettant à l'optimisation de se propager de manière ascendante (bottom-up) depuis les littéraux jusqu'aux structures de contrôle.
2. **Optimisations de bas niveau (Post-Génération) :** Le répertoire contient également les optimiseurs agissant sur le code intermédiaire IMA (`PeepHoleOptimizer`,

`ConstantPropagationOptimizer`, `CopyPropagationOptimizer`, et `DeadStoreOptimizer`.). Ces derniers interviennent après la génération de code pour effectuer un nettoyage final sur la liste séquentielle des instructions, et donc ils ne sont pas délocalisé dans le reste des fichiers comme les optimisation de haut niveau.

4.2 Intégration :

'intégration des optimiseurs respecte la structure modulaire du compilateur Deca :

- **Intégration dans l'étape A/B (Front-end) :** Les classes `ConstantFolder` et `DeadCodeEliminator` interviennent juste après l'analyse contextuelle. Elles modifient l'AST pour fournir un arbre "épuré" à la génération de code.
- **Intégration dans l'étape C (Back-end) :** Les optimiseurs IMA `Peephole`, `Copy Propagation`, `Constant Propagation` et `Dead Store Elimination` agissent comme un post-processeur. Ils reçoivent un objet IMAProgram et modifient sa liste interne de lignes d'instructions avant l'écriture finale dans le fichier **.ass**.
- **Choix techniques :** Utilisation intensive du design pattern Visitor (pour le ConstantFolder) et de tables de hachage (HashMap) pour le suivi des états des registres et de la mémoire.

4.3 Orchestration et Séquencement des Optimisations

L'efficacité d'un compilateur ne dépend pas seulement de la qualité de ses optimiseurs, mais aussi de l'ordre dans lequel ils sont appliqués. Notre pipeline d'optimisation est stratégiquement réparti autour des phases clés de la compilation : la vérification contextuelle et la génération de code.

Extrait de DecacCompiler.java :

```
if (compilerOptions.getOptimize()) {
    ConstantFolder constantfolder = new ConstantFolder();
    constantfolder.apply(this, prog);

    DeadCodeEliminator deadcodeeliminator =  new DeadCodeEliminator();
    deadcodeeliminator.apply(this,prog);
}

addComment("start main program");
prog.codeGenProgram(this);
addComment("end main program");


if (compilerOptions.getOptimize()) {
    if (prog.getIsOOP() == false){
        CopyPropagationOptimizer copyPropagationOptimizer = new CopyPropagationOptimizer();
        copyPropagationOptimizer.optimize(program);

        ConstantPropagationOptimizer constantPropagationOptimizer = new ConstantPropagationOptim
```

```
        constantPropagationOptimizer.optimize(program);}
    PeepHoleOptimizer peepholeOptimizer = new PeepHoleOptimizer(!compilerOptions.getNoCheck());
    peepholeOptimizer.optimize(program);

    if (prog.getIsOOP() == false){
        DeadStoreOptimizer deadStoreOptimizer = new DeadStoreOptimizer();
        deadStoreOptimizer.optimize(program);

        peepholeOptimizer.optimize(program);
    }
}
```

1. Phase Pré-Génération : Optimisations sur l'AST
Les premières optimisations interviennent immédiatement après la Vérification Contextuelle (`verifyProgram`). À ce stade, l'arbre est entièrement décoré (types vérifiés, symboles liés), garantissant que toute transformation est sémantiquement sûre.

- **Placement :** Entre `assert(prog.checkAllDecorations())` et `prog.codeGenProgram(this)`.
- **Séquence :** Le `Constant Folding` est lancé en premier pour simplifier les expressions calculables. Il est immédiatement suivi du `Dead Code Eliminator`. Ce chaînage est crucial car le repliement de constantes révèle souvent des branches de code inaccessibles (ex: `if (false)`) que le `DCE` peut alors supprimer avant même qu'elles ne soient traduites en assembleur.

2. Phase de Génération de Code
La méthode `prog.codeGenProgram(this)` traduit `l'AST simplifié` en instructions IMA. Les commentaires "start main program" et "end main program" délimitent cette phase de production brute.

3. Phase Post-Génération : Optimisations de Bas Niveau

Une fois le code IMA produit sous forme de liste d'instructions, nous appliquons les optimiseurs de bas niveau. Ces derniers travaillent sur la réalité physique du programme (registres et mémoire).

- **Filtrage OOP :** Comme discuté, les optimisations `Copy Propagation`, `Constant Propagation` et texttttDead Store sont réservées aux programmes sans objets (getIsOOP() == false). Cela évite des calculs inutiles face aux barrières de flux induites par les méthodes.
- **Le rôle pivot du Peephole :** Il est appelé une première fois pour nettoyer les redondances simples issues de la génération. Il est appelé une seconde fois après le `Dead Store Optimizer.` Ce double passage est nécessaire car la suppression d'un `STORE` peut laisser derrière elle un registre chargé inutilement (`LOAD`), que le `Peephole` pourra alors éliminer pour parfaire l'optimisation.

5 Méthode de Validation

Pour garantir que nos optimisations ne cassent pas la sémantique du langage Deca, nous avons mis en place le protocole suivant :

- **Tests de Non-Régression :** Utilisation de la suite de tests existante. Le résultat (output) d'un programme compilé avec -o doit être strictement identique à celui sans -o.
- **Tests Spécifiques d'Optimisation :** Afin d'évaluer finement le comportement de nos algorithmes, nous avons ajouté une batterie de tests dédiés dans le répertoire src/test/deca/codegen/optim/. Ces tests se divisent en deux catégories :
  - **Conditions favorables (Gains maximaux) :** Programmes conçus pour exposer des chaînes de calculs constants, des copies de variables redondantes et du code mort évident. L'objectif est de valider que les optimiseurs atteignent bien le "point fixe" et réduisent le code au minimum théorique.
  - **Conditions avec barrières (Sûreté) :** Programmes intégrant volontairement des étiquettes (labels), des branchements conditionnels et des appels de méthodes. Ces tests vérifient que les optimiseurs détectent correctement les ruptures de flux et interrompent la propagation (flush des tables) au bon moment pour ne pas corrompre le programme.
- **Critères d'acceptation :**
  - **Sûreté :** Aucune instruction nécessaire ne doit être supprimée .
  - **Efficacité :** Le nombre de lignes du fichier .ass doit diminuer sur les benchmarks arithmétiques.
  - **Stabilité :** Pas de boucles infinies lors des passes itératives (limites de 20 et 30 passes).

6 Résultats et Discussion

6.1 Synthèse des validations

L'évaluation de notre pipeline d'optimisation sur notre suite de tests montre que l'efficacité du compilateur est extrêmement dépendante de la structure du code source. Les observations suivantes ressortent de nos tests de validation :

- **Réduction importante dans les cas favorables :** Sur des programmes à forte densité arithmétique et logique avec une structure favorisant nos optimisations, l'optimisation peut contribuer à une réduction du nombre d'instructions IMA supérieure à 50 % (même beaucoup plus, mais que dans des cas très théoriques qui sont fait pour maximiser le gain en optimisation). Ce gain est le résultat d'un "effet domino" où le Constant Folding et le DCE simplifient les expressions, et au peephle d'effectuer les simplifications algébriques possibles à la sortie
- **Optimisation des flux mémoire :** Selon la complexité des blocs élémentaires, nous estimons que l'élimination des accès redondants (LOAD et STORE) oscille entre 10 % et 40 %. Ces variations s'expliquent par la fréquence des "barrières" rencontrées (labels, branchements) qui forcent l'optimiseur à être plus conservateur.
- **Nature des résultats :** Il est important de souligner que ces chiffres sont des estimations basées sur nos benchmarks internes. Dans un programme réel, le gain final dépendra de la proportion de types élémentaires par rapport aux objets et de la linéarité du flot de contrôle.

6.2 Discussion et Limites

- **L'arbitrage de l'Analyse Locale** L'absence d'analyse inter-procédurale constitue une limite assumée. Nos optimiseurs traitent chaque méthode comme une unité isolée. Les axes d'amélioration suivants représenteraient l'étape supérieure d'un compilateur :

  - **1. Le Graphe de Flot de Contrôle (CFG) :** Actuellement, nos optimiseurs travaillent sur des blocs élémentaires linéaires. Cela signifie que dès qu'un branchement ou un label apparaît, l'optimiseur "perd" le fil des données par sécurité.

    L'utilisation d'un CFG permettrait de modéliser le programme comme un graphe où chaque nœud est un bloc et chaque arête un saut possible.

    * **Avantage :** On pourrait propager des constantes même à travers des if/else en calculant l'intersection des états à la réunion des branches (nœuds de jonction).
    * **Limitation actuelle :** Sans CFG, notre analyse s'arrête à chaque étiquette, ce qui fragmente les opportunités d'optimisation dans les codes contenant beaucoup de structures de contrôle.
  - **2. La forme SSA (Static Single Assignment) :** La forme SSA est une représentation intermédiaire où chaque variable n'est assignée qu'une seule fois dans le code. Pour gérer les variables qui changent de valeur selon le chemin emprunté (ex: dans une boucle), la forme SSA utilise des fonctions spéciales appelées fonctions ϕ.

    * **supériorité de SSA :** Il simplifie radicalement la propagation de copies et l'élimination de code mort, car chaque utilisation d'une variable pointe vers sa définition unique. Et il permet de détecter très facilement si une variable est "morte" à l'échelle de toute la méthode, et non pas seulement à l'intérieur d'un bloc.
    * **Le choix du projet :** L'implémentation du SSA nécessite une infrastructure complexe (construction du dominateur d'arbre, insertion des fonctions ϕ, conversion finale vers registres). Face aux contraintes du projet et à la structure de l'architecture IMA, nous avons jugé son coût de développement disproportionné par rapport aux gains attendus pour des programmes Deca standards.

- **La barrière de l'Orienté Objet :** La gestion des objets impose un conservatisme strict dû à l'aliasing et à l'incertitude du tas (`heap`) après un appel de méthode. Pour garantir la sûreté sémantique, les propagations sont désactivées en mode OOP, limitant l'optimisation des structures complexes.

- **Éco-responsabilité et Point Fixe :** L'approche itérative à point fixe assure une convergence rapide bien avant les limites de sécurité. Ce mécanisme minimise l'empreinte énergétique en limitant le travail du compilateur aux seules simplifications effectives (20 à 30 passes au maximum).

6.3 Ressources

- Le livre Aho, AV, Lam, MS, Sethi, R, & Ullman, JD(2006). Compilers: Principles, Techniques, and Tools (The Dragon Book). (Notre référence principale)

- Le cours "Compiler Optimization" par `compilerai` sur YouTube.

- Le cours "Software Analysis & Optimization" par `Udacity - Georgia Tech` sur YouTube.