femto-st SCIENCES & TECHNOLOGIES

afaq ISO 9001 Qualité AFNOR CERTIFICATION

SERVICE COMMUN ELECTRONIQUE ET INSTRUMENTATION

Rencontre « Optimiser un programme LabVIEW »

Recueil de bonnes pratiques de programmation et compréhension du compilateur LabVIEW

RÉSEAU DES ÉLECTRONICIENS INSTRUMENTALISTES

RdEI - NATIONAL

cnrs dépasser les frontières

Villeneuve d'Ascq, le 28 septembre 2017

Patrick NECTOUX

Préambule

La première partie de cette présentation est à destination des programmeurs débutants avec pour objectif de pouvoir adopter les bonnes pratiques de programmation dès les premiers pas et liées à la maintenabilité de son code.

La seconde, plus à destination des programmeurs expérimentés, survole le fonctionnement du compilateur LabVIEW dans l'optique d'identifier l'origine des conseils d'optimisation de votre code, et qui vous seront fournis aussi par ailleurs lors de cette journée technique.

I. Petit recueil de bonnes pratiques

a) Rapidité de développement

- S'approprier les modèles de conception et les exemples de projet
- Utiliser les librairies existantes et leurs fonctions afin de ne pas réinventer la roue
- Utiliser les « Type Definition », notamment avec les énumérateurs
- Utiliser les raccourcis-clavier
- Utiliser le « Quick drop » pour le placement des fonctions sur le diagramme

b) Modularité du code

- Structurer son code en définissant les sous-VI judicieusement
- Organiser les variables dans des clusters

I. Petit recueil de bonnes pratiques

c) Performance du code

- Refermer les références
- Éviter les nœuds de propriété sur les indicateurs
- Placer le terminal des indicateurs à l'endroit où le rafraichissement doit avoir lieu dans le code
- Identifier la nécessité de définir un timeout sur les structures évènementielles

d) Lisibilité du code

- Organiser la lecture de gauche à droite
- Placer les contrôles à gauche et les indicateurs à droite dans les sous-VI
- Aligner les fonctions sur le fil d'erreur
- Retravailler l'icône des sous-VI
- Résoudre les points de coercition …

I. Petit recueil de bonnes pratiques

e) Documentation du code

- Intégrer un descriptif dans les propriétés des VIs
- Intégrer des commentaires de type texte
- Faire apparaitre le label des sous-VIs
- Renseigner et faire apparaitre le label des structures (While, …)
- Faire apparaitre le label des fonctions
- Renseigner et faire apparaitre un label sur les fils de liaison

f) Documentation de l'IHM (Face-avant)

- Nommer les contrôles et indicateurs de manière judicieuse
- Créer des info-bulles sur les objets pour l'opérateur

I. Petit recueil de bonnes pratiques

g) Flux de données et gestion d'erreur

- Le fil d'erreur définit l'ordre d'exécution et traverse bien tous les nœuds
- Utilisation des structures conditionnelles pour le code qui ne renvoie pas de cluster d'erreur
- Utilisation des séquences avec parcimonie
- Utilisation du cluster d'erreur en entrée et en sortie de sous-VI
- Le VI Main se termine par l'exécution du gestionnaire d'erreur simple

II. Compilateur de LabVIEW

a) LabVIEW, langage multi-paradigme

- Implémentation d'une grande variété de concepts : flux de données, programmation évènementielle, langage orienté objet …
- Multi-plateforme : Windows, Mac et Linux
- Multi-cible : PowerPC, Intel x86, Sparc …
- Temps réel et FPGA Xilinx

Le compilateur évolue depuis 1986 avec la version 1.0 de LabVIEW jusqu'à nos jours en intégrant au fil de l'eau de nouvelles fonctionnalités. Il transforme le langage G en un code compilé ou cross-compilé, compréhensible et exécutable par le système cible.

II. Compilateur de LabVIEW

b) Processus de compilation

- Exécution de l'algorithme de propagation de type
  - ✓ Résolution des types implicites
  - ✓ Détection des erreurs de syntaxe
- Conversion par l'éditeur de diagramme en DFIR
- Transformations du graphe DFIR (DataFlow Intermediate Representation)
  - ✓ Décomposition
  - ✓ Optimisations (Inplacer et clumber)
  - ✓ Préparation à la génération du code machine
- Traduction dans une représentation intermédiaire LLVM (Low-Level Virtual Machine)
- Obtention du code machine

II. Compilateur de LabVIEW

c) Propagation de type et son algorithme

Cette étape détermine la validité du code, et stoppe la compilation en cas d'erreur. Elle enchaine les tâches suivantes :

- Résoudre les types implicites

  L'exemple de la primitive ou du nœud additionner permet d'illustrer cette notion de propagation : le type ou la représentation du numérique de sortie est celui ou celle des entrées.

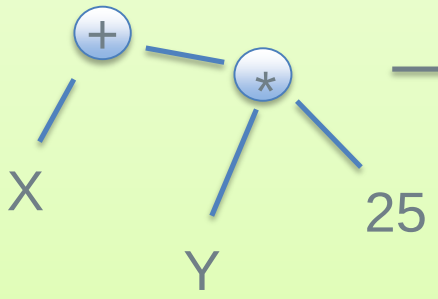
- Résoudre les appels de sous-VI (validité)
- Identifier le sens des fils de liaison
- Détecter et transférer les erreurs de syntaxe

II. Compilateur de LabVIEW

d) IR ou représentation intermédiaire du code

Une IR est une notion générique aux compilateurs modernes et qui constitue une représentation du programme. Elle est transformée lors de toutes les phases de la compilation.

Exemple d'IR :



Transformation d'un arbre de syntaxe abstraite en code séquentiel de type assembleur

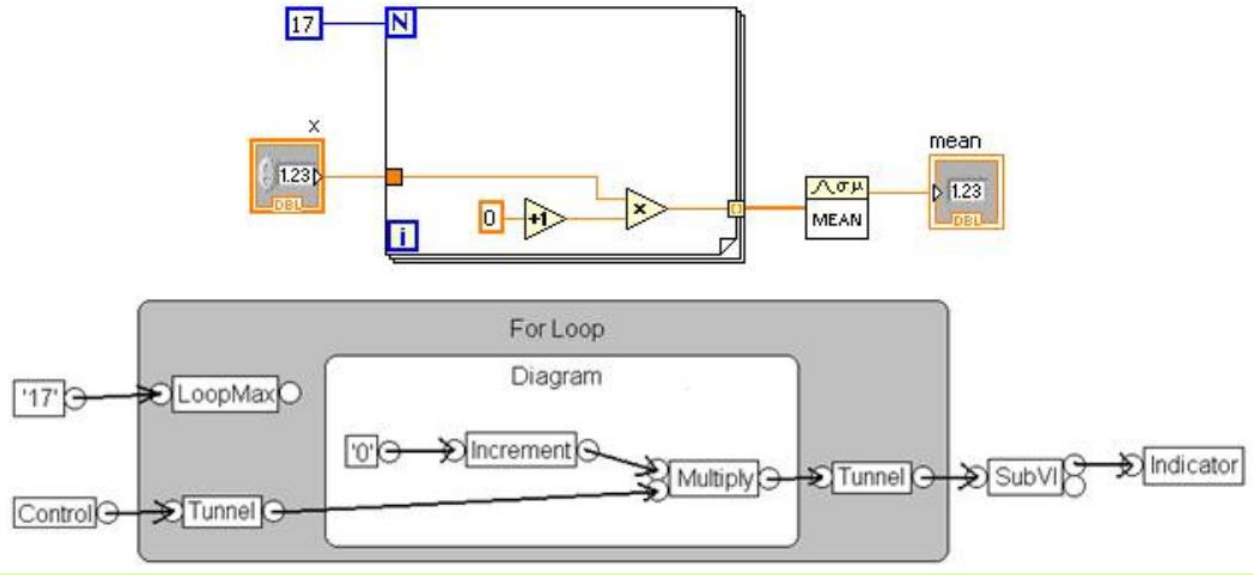
```
M0 <- Y
M1 <- 25
M2 <- M0 * M1
M0 <- M0 + X
```

Le compilateur de LabVIEW fait appel à deux IR, la DFIR et la LLVM.

II. Compilateur de LabVIEW

e) DFIR (DataFlow Intermediate Representation)

Elle est hiérarchique, graphique et ressemble au langage G. Elle sert également de point commun à de nombreuses transformations de haut et bas niveau.



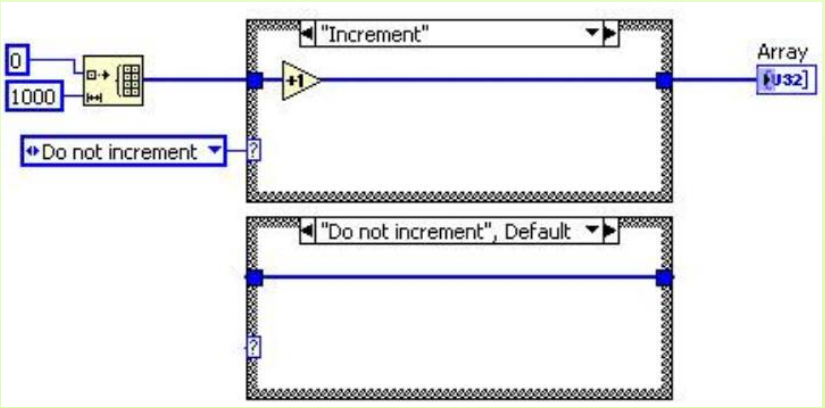
Exemple de code G LabVIEW et son graphe DFIR associé

II. Compilateur de LabVIEW

e) DFIR (suite)

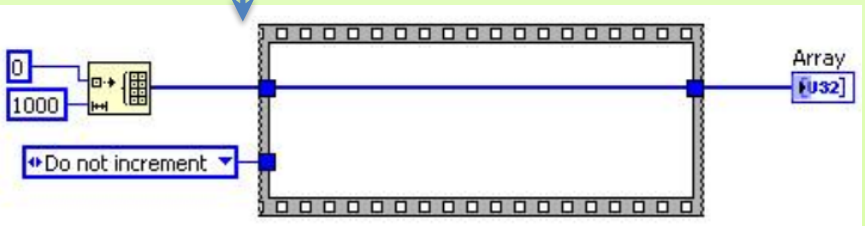
Les transformations suivantes ont pour objectif de réduire et de normaliser la représentation du code de haut niveau :

- ✓ Inlining des sous-Vis (inclure le code d'un sous-VI dans le VI appelant permet souvent de simplifier le code global)
- ✓ Élimination du code non accessible



Avant

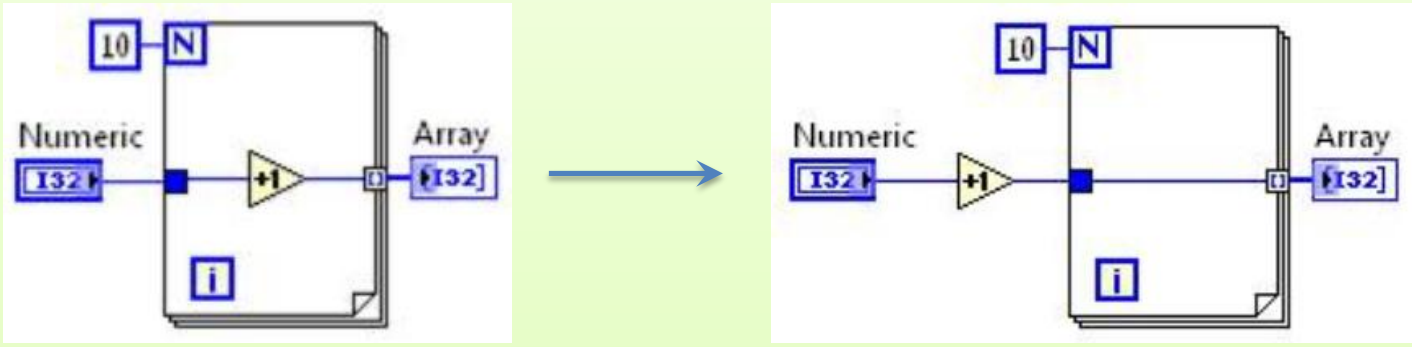
et après élimination du code non accessible



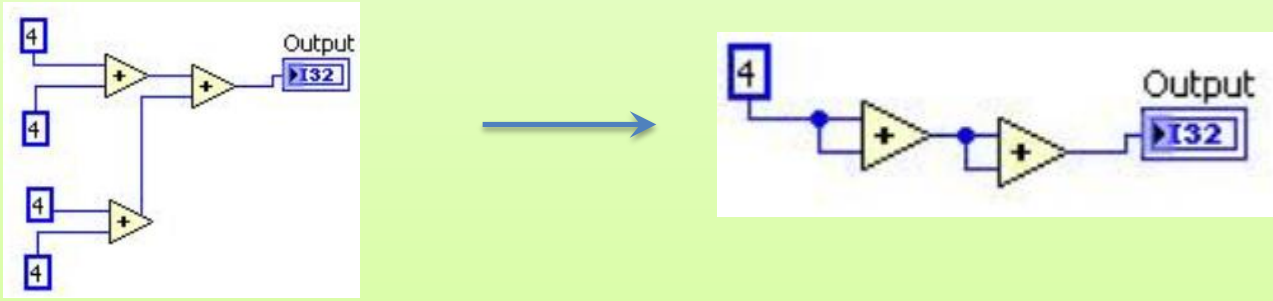
II. Compilateur de LabVIEW

e) DFIR (suite)

✓ Extraction du code invariant des boucles



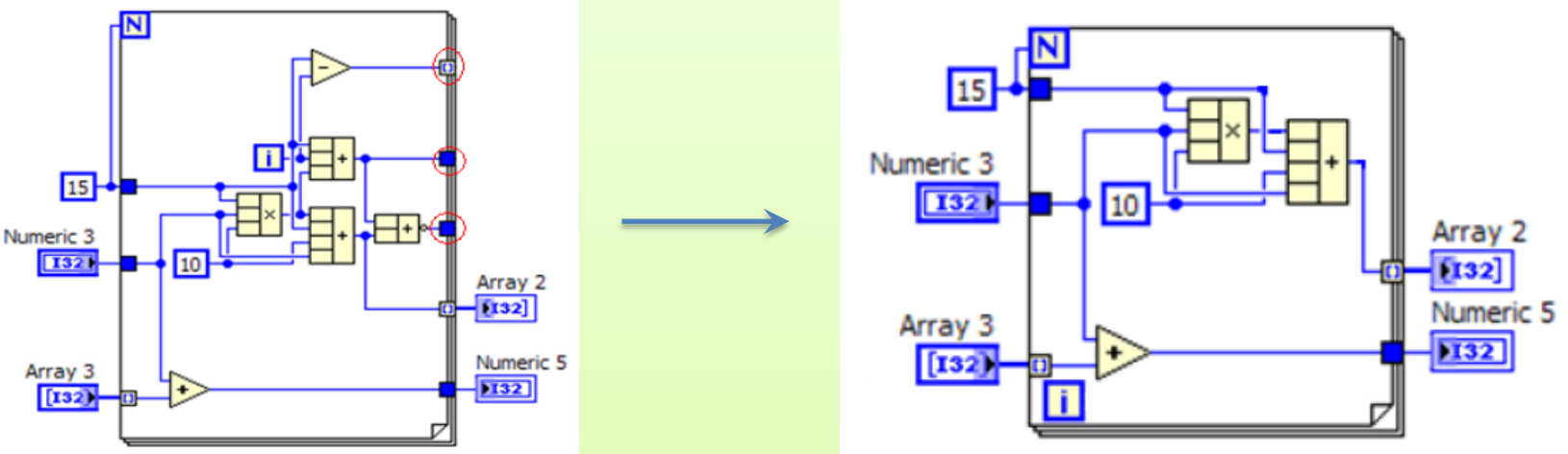
✓ Élimination des sous-expressions redondantes



II. Compilateur de LabVIEW

e) DFIR (suite)

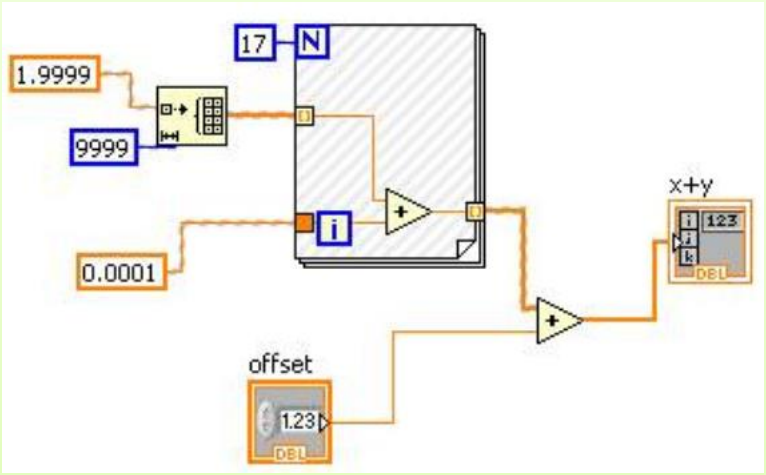
✓ Élimination du code mort (un tunnel non connecté en sortie de boucle)



II. Compilateur de LabVIEW

e) DFIR (suite)

- ✓ Réduction des constantes



- ✓ « Déroulage » de boucle a pour objectif la réduction du nombre d'itérations

II. Compilateur de LabVIEW

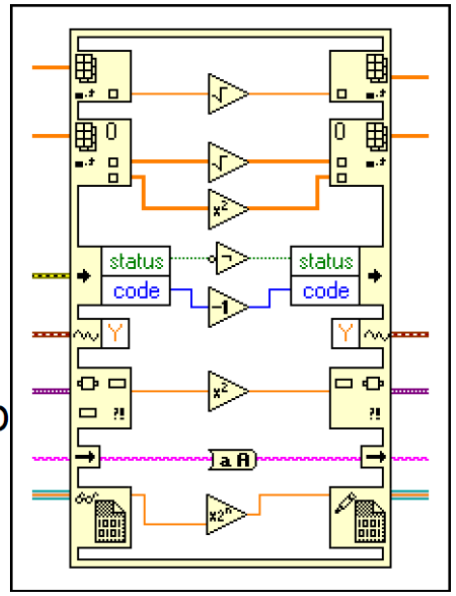
e) DFIR (suite)

Les transformations suivantes sont considérées come étant de bas niveau.

- ✓ Clumber, outil de création de blocs (notion de parallélisme)
- ✓ Inplacer, outil de réutilisation des buffers (éviter le copies de données)
- ✓ Allocateur de mémoire
- ✓ Générateur de code (processeur cible)

In place element structure

- Array index/replace element
- Array split/replace subarray
- Unbundle/bundle
- Waveform unbundle/bundle
- Variant to/from G data
- Simple “in place” relationship
- Data value reference read/write



II. Compilateur de LabVIEW

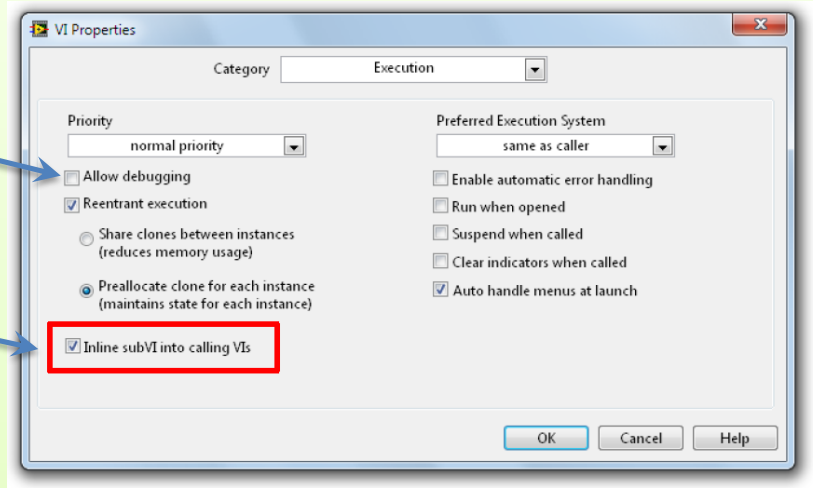
f) LLVM (Low-Level Virtual Machine)

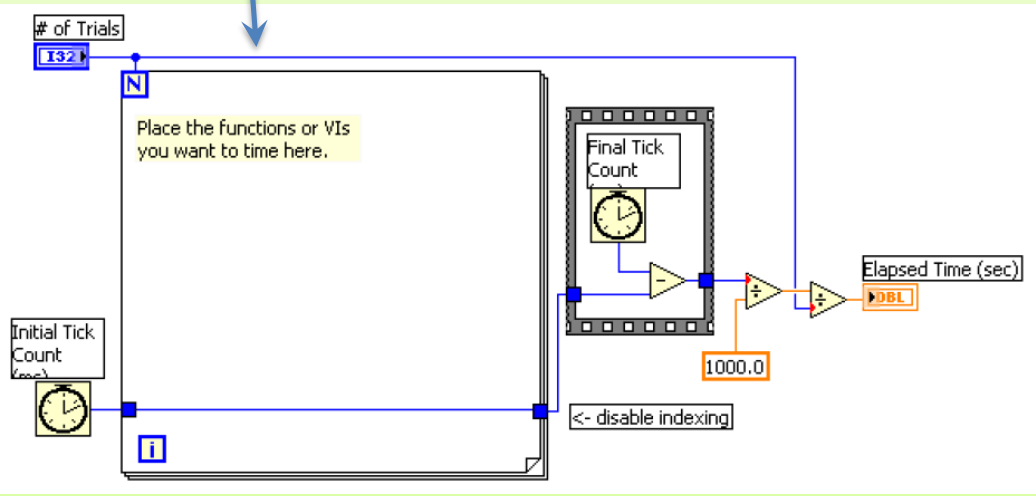
Cette infrastructure de compilateur libre et largement utilisée a pour objectif de réduire les temps d'exécution et utilise notamment les optimisations suivantes :

- ✓ Ordonnancement et combinaisons des instructions
- ✓ Propagation conditionnelle
- ✓ Loop unswitching
- ✓ Allocation des registres
- ✓ …

III. Que peut-on faire ?

- Inhiber le débogage
- Paralléliser les boucles
- Valider l'Inlining
- Mesurer les temps d'exécution et expérimenter





- Pipelining
- Allocation mémoire tableau

Et bien d'autres choses encore …

Questions diverses et commentaires

- C'est à vous

Merci de votre attention

Source : www.ni.com/tutorial/11472/en/