0.1 Interprocéduralité

L'interprocéduralité apporte de nombreux avantages dans le cadre de la génération de code pour systèmes embarqués. Outre l'argument le plus évident concernant la compilation de programmes complexes, cela permet de simplifier la génération de code en cachant des choses matérielles dans des procédures équivalentes, de la même manière que cela est fait avec des objets dans SystemC par exemple [**?**].

Il suffit de fournir des fonctions aux effets équivalents pour que le code généré puisse être correctement analysé par PIPS ou être exécutable en tant que simulation. C'est en traduisant ces appels de fonctions en primitives matérielles qu'on aura les exécutables pour le système cible. .

0.2 Langage de sortie

Un langage de sortie a été défini afin de s'interfacer avec les autres outils du projet. Afin d'éviter de faire de la génération de code de très bas niveau et parce que PIPS est à la base un compilateur source vers source, on va générer du source de bas niveau mais faisant appel à des pseudo fonctions qui seront exécutées par le simulateur ou interprétées par les autres outils.

0.3 Phases de compilation

La compilation à partir du source de haut niveau en divers programmes ciblant l'architecture de bas niveau est divisées en une succession de phases compilations. Ceci permet une hiérarchisation du problème simplifiant l'approche et permettant la réutilisation des morceaux de compilateur.

0.3.1 Analyse syntaxique

Cette phase primitive analyse le programme en entrée pour générer une représentation abstraite hiérarchique découpée en fonctions et procédures appelés ici *modules*.

Chaque module est composé de sa propre représentation hiérarchique autant que possible afin de garder un maximum de finesse aux phases d'analyses. Toutes les parties de code non structuré sont enfermées dans des objets *unstructured* que l'on peut voir comme des hyperblocs, eux-mêmes contenant du code plus ou moins structuré.

Les modules sont reliés entre eux par le graphe d'appels et le graphe des appelants pour permettre de gérer l'interprocéduralité.

- Analyse syntaxique ;
- phases d'optimisations classiques pour simplifier la suite, à savoir
  –

0.3.2 Analyse de régions de tableau

Il s'agit d'un cas particulier de graphe de dépendance ou on considère des morceaux de tableaux. Cela permet de manipuler plus globalement des applications qui ferait sinon exploser un graphe de dépendance plus classique.

Les régions vont permettre de savoir quels sont les éléments de données qu'une fonction a besoin pour faire son calcul et quels sont les éléments produits par cette fonction qui seront utilisés par la suite.

Ces informations seront utilisées dans la suite de la compilation pour générer par exemple les communications entre opérateurs ou processeurs ou encore pour faire de la parallélisation de haut niveau.

0.3.3 Optimisations de code classiques

Diverses optimisation sont utilisées pour simplifier au maximum le programme de départ et éviter de compiler en matériel des choses inutiles.

Les optimisations s'apportant mutuellement des simplifications il faut les appliquer dans un point fixe jusqu'à ce que le code ne soit plus optimisable avec ces méthodes.

Restructuration du graphe de contrôle

Évaluation partielle

Élimination de code mort

Préconditions
élimination Use-def

0.3.4 Parallélisation

Plusieurs techniques vont être employées ici : la parallélisation à gros grain et la parallélisation à grain fin.

Parallélisation à gros grain

Cette technique utilisant les régions de tableau va être utilisée pour paralléliser du code au niveau le plus haut telle que séquences ou des boucles à corps complexe pour rajouter la notion de parallélisme à gros grain tel que du parallélisme de processus légers (*threads*).

Parallélisation à grain fin

Cette technique de parallélisation plus classique est utile pour extraire du code du parallélisme de données massif sous forme de code vectoriel, de boucles parallèles style HPF ou d'instructions multimédia ou DSP.

0.3.5 Atomisation

Cette phase a pour effet de transformer du code complexe de type

```
A(3*I + 5) = F(J, B(24*C(J))*4
```

en code simple que l'on pourrait qualifier de type RISC tel que

```
TEMP_I_0 = 3*I + 5
TEMP_I_1 = C(J)
TEMP_I_2 = 24*TEMP_I_1
TEMP_I_3 = B(TEMP_I_2)
TEMP_I_4 = F(J,TEMP_I_3)
A(TEMP_I_0) = TEMP_I_4
```

Cela permet de simplifier les phases de compilation ou d'optimisation ultérieures telle que la suppression des sous-expressions communes.

En plus comme c'est du code très simple il est facile à traduire en RTL ou à remettre sous une forme plus compacte avec des techniques de propagation de copie et d'évaluation partielle pour augmenter le grain à nouveau pour une sortie SmallTalk de PHRASE.

0.3.6 Adaptation des opérateurs

Un programme de haut niveau peut manipuler des données de type incompatible avec les opérateurs matériels de la machine cible généralement plus limités pour des raisons de coût.

La traduction peut être faite au niveau des bibliothèques SmallTalk ou au sein du compilateur de plus haut niveau. L'intérêt de ce dernier cas est qu'on peut continuer d'optimiser le source du code ainsi transformé : pipeline, code en commun, etc.

Le code sur des données de 16 bits sera par exemple transformé de

```
A = B + C
```

en code sur 8 bits

```
! Calcule le poids faible :
TMP_ADD = PHRASE_ADD_CARRY(PHRASE_BIT_EXTRACT(B, 0, 7),
                           PHRASE_BIT_EXTRACT(C, 0, 7),
                           TMP_CARRY,
                           0)
! Calcule le poids fort :
A = PHRASE_BIT_MERGE(PHRASE_ADD_CARRY(PHRASE_BIT_EXTRACT(B, 15, 8),
                                      PHRASE_BIT_EXTRACT(C, 15, 8),
                                      TMP_USELESS,
                                      TMP_CARRY), 8, 15
                     TMP_ADD, 0, 7)
```

Cette phase prend un fichier de paramètres décrivant l'architecture cible.

Faudrait définir une liste d'opérateurs indispensables

0.3.7 Pipeline logiciel

0.3.8 Partitionnement

C'est la phase qui va découper le travail et les zones de stockage entre les différentes entités hétérogènes du système de manière automatique ou en suivant les directives de programmation fournies par l'utilisateur.

Afin de simplifier le problème, on va s'intéresser uniquement aux zones de stockage et on utilisera ensuite la règle des écritures locales pour partitionner les calculs, c'est à dire qu'on placera un calcul au même endroit que la variable devant servir au stockage du résultat.

Cette règle semble contraignante mais en fait elle permet de traiter tous les cas puisque si on veut disposer les calculs de manière particulière, il suffit de passer par un tableau intermédiaire disposé comme on l'entend. C'est identique à la réalisation de la primitive `ON` de HPF comme discuté dans [**?**].

Cette phase prend un fichier de paramètres décrivant l'architecture cible.

0.3.9 Explicitation des communications

Une fois le partitionnement effectué, toute utilisation d'une valeur stockée dans une zone et utilisée dans une autre sera encapsulée dans une fonction intrinsèque représentant une communication explicite [**?**] qui sera traduite lors de l'exécution par une communication concrète selon le protocole et matériel approprié : bus, fil d'attente, etc.

Cette phase permet de traduire le typage de localisation en communications simple entre les dispositifs hétérogènes.

Dans la suite les phases pourraient être externes à PIPS et gérées par des outils plus spécifiques

0.3.10 Génération d'automates

Si on décide d'exécuter du code quelconque sur un système reconfigurable qui n'est pas capable d'exécuter du flot de contrôle classique il faut traduire un code en un automate qui va s'exécuter matériellement.

Chaque branchement dans le graphe de contrôle (boucles ou tests) sera traduit par un changement d'état vers le nouvel état représentant la branche à prendre.

Cet automate pourra ensuite être synthétisé avec des outils architecturaux.

On peut en fait voir cette transformation comme le plaquage d'un graphe de contrôle quelconque sur une boucle infinie qui exécuterait séquentiellement chaque cas possible. C'est une transformation qui est utilisée dans les compilateurs VLIW ou SIMD [**?**] pour faire de la prédication de code ou de l'*if conversion* extrême.

0.3.11 Génération de code RTL

Afin de simplifier le chemin critique ou les opérateurs dans le système reconfigurable on peut découper dans cette phase les opérateurs en rajoutant des verrous comme barrières temporelles.

D'un point de vue logiciel, on peut voir ces rajouts comme un cas particulier des communications (§ 0.3.9) mais entre des opérateurs fonctionnels de la même entité.

Cette phase est aussi couplable à du pipeline logiciel.

Faudrait remonter cette phase + avant ?

0.3.12 Prettyprinters

Deux prettyprinters ont été développés pour le projet, un qui génère des fonctions intrinsèques exécutables dans un but de simulation ou qui sont remplacées par des instructions matérielles de PHRASE et un autre qui génère directement du SmallTalk pour le système MADEO.

Génération de pseudo fonctions intrinsèques

En fait, comme il s'agit de fonctions standard, il suffit d'utiliser le prettyprinter classique de PIPS et il n'y a rien d'autre à développer.

Génération de SmallTalk

En ce qui concerne la sortie SmallTalk il y a plus de choses à faire puisque il va falloir traduire le code contenant des fonctions intrinsèques en code MADEO.

Il faudra d'autre part sortir le code de contrôle qui sera synchronisé avec le code MADEO via encore l'usage de fonctions intrinsèques.