Construction de programmes pour systèmes embarqués

Compilation pour SOC

Paul Feautrier

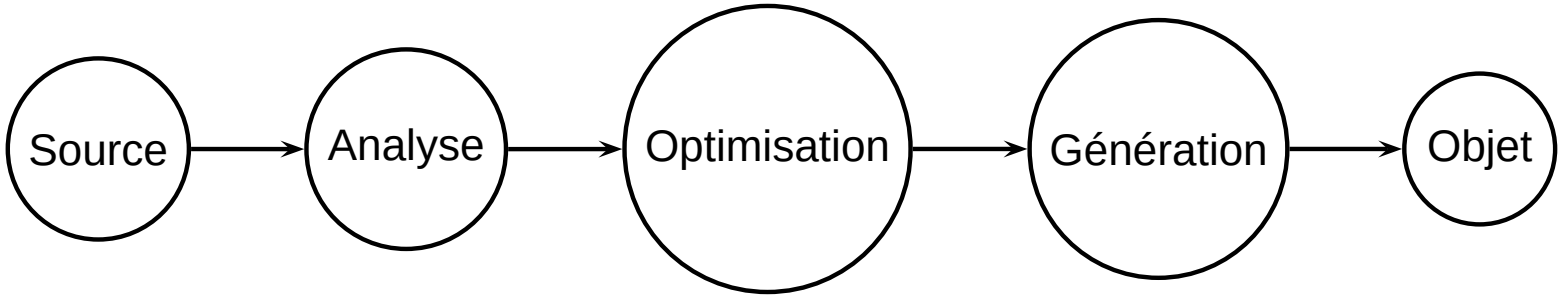
Paul.Feautrier@ens-lyon.fr.

ENS Lyon

Plan

- Tendances récentes en compilation
- Particularités de la compilation pour SOCs
- Quelques problèmes :
  - Le langage source
  - Contraintes temps réél et contraintes de ressources
  - Réduction de la taille de la mémoire
  - Modularité
  - Partitionnement
  - Compilateurs et systèmes d’exploitation
- Conclusions

Tendances récentes en compilation



- Affiner la phase d'analyse pour retrouver le « sens » du programme. Programme impératif $\rightarrow$ Programme en assignation unique $\rightarrow$ Recherche d' « idiomes » (réductions).
- Formaliser la partie « optimisation » comme un problème de *programmation mathématique*.
- Mais pour les systèmes embarqués et les SOCs, les fonctions objectifs sont plus difficiles à formaliser.

Particularités de la compilation pour SOCs

- En calcul à hautes performances, la configuration est donnée, et il s'agit d'obtenir les meilleures performances possibles pour un programme donné.
- Pour un système embarqué, il s'agit de trouver la plus « petite » configuration qui atteint les performances demandées (e.g. dessiner une image en $1/30$ème de seconde). Petite = surface de Si, consommation, coût, etc.
- Si le système embarqué est un SOC, le problème semble analogue; avec quelques différences :
  - Les paramètres ont des valeurs différentes : par exemple, bande passante de la mémoire plus élevée (?).
  - Le critère « surface de Si » semble plus important.

Le langage source

- Langage impératif (C, Matlab, System C)
  - Familier, simulable, spécification du parallélisme difficile.
- Langage en assignation unique (ou SARE)
  - Naturellement parallèle, intuitif pour les spécialistes de traitement du signal.
- Réseaux de processus de Kahn
  - Naturellement parallèle, intuitif pour les électroniciens.

On passe automatiquement d'un programme impératif à un SARE par *analyse du flot des données* et d'un réseau de processus de Kahn à un SARE par *comptage des messages*.

Granularité, paramétrage, reciblage

- Les méthodes de compilation modernes sont relativement indifférentes aux problèmes de granularité. Une opération élémentaire peut être une opération logique ou arithmétique, un appel de fonction, l'activation d'un IP, etc. Il suffit d'en connaître les caractéristiques d'entrée/sortie et les paramètres temporels. Il y a cependant des problèmes conceptuels à résoudre (concept de délai en temps multidimensionnel).
- Ces méthodes sont également facile à paramétrer. Par exemple, la quantité disponible d'une ressource intervient dans l'établissement des contraintes, mais la méthode de résolution n'en dépend pas.
- La réalisation d'un compilateur reciblable est un problème différent, qui peut être résolu par des techniques de réécriture.

Contraintes temps-réél, contraintes de ressource

Un ordonnancement est une fonction θ qui donne la date de lancement $\theta(u)$ de chaque opération u du programme.

- Contraintes de latence : $\forall u : 0 \leq \theta(u) \leq L$.
- Contraintes de dépendances : si u est en dépendance avec v $\theta(u) + \partial(u) \leq \theta(v)$, où $\partial(u)$ est le délai d'exécution de u.
- Contraintes de ressource : si u et v utilisent la même ressource, $\theta(u) \neq \theta(v)$.

Moyennant diverses hypothèses simplificatrices, on peut minimiser la latence à délais fixés, ou maximiser les délais à latence fixée, etc. La prise en compte des contraintes de ressources est encore imparfaite.

The Methodological Box

A	A/K	*terra incognita*				
ST	Farkas	(a)	(b)	(c)	(d)	
U	W/L, D/V	software pipelining				Touati
BB	sort	ILP, list algorithms				Touati
	NONE	**P**	**1**	**G**	**TAB**	**NL**

BB: basic blocs

ST: static control programs

U: perfect loop nests, uniform dependences

A: arbitrary programs

NONE: no resource constraints

1: a finite set of distinct resources

TAB: resources with reservation tables

P: a finite number of identical resources

G: a finite set of finite resource classes

NL: resources with non local usage

Gestion de la mémoire

- La mémoire est grande consommatrice de place et d'énergie.
- A ordonnancement donné :
  - Calculer la durée de vie de chaque *valeur*.
  - Affecter deux valeurs ayant des durées de vie disjointes à la même cellule de mémoire.
- A latence donnée :
  - Trouver l'ordonnancement utilisant le moins de mémoire.
  - Problème non encore résolu. D'autres contraintes doivent être introduites, sans quoi on risque d'obtenir un programme séquentiel sur un processeur ultra-rapide.
- Architecture de la hiérarchie de mémoire : cache ou *scratchpad*.
- Optimisation fine des accès à la mémoire (multiport).

Partitionnement

- Découpage du programme en phases (séquentielles, parallèles, en pipeline, etc.)
- Affectation des phases à divers types d'opérateurs (ASIC, DSP, IP, processeurs généralistes, etc).
- Prise en compte des temps de communication.
- Par chance, les mêmes techniques de programmation mathématique peuvent être utilisées pour concevoir un ASIC ou pour programmer un VLIW, etc.

Il est difficile de formaliser le problème du découpage comme un problème de programmation mathématique. On est conduit à utiliser des méthodes exploratoires associées à des idées venues de *l'optimisation multicritères*.

Modularité

- La compilation modulaire est indispensable pour traiter des exemples de taille réaliste.
- Il faut que le compilateur construise un *résumé* de chaque module pour permettre la compilation des autres modules.
- Que faut-il stocker pour poursuivre la parallélisation?
  - Les ensembles lus et modifiés [Triolet, Irigoin]? Cela suppose une exécution atomique du module et demande une estimation de sa durée d'exécution.
  - Les contraintes d'ordonnancement visibles de l'extérieur? Permet une parallélisation simultanée de l'intérieur et de l'extérieur du module. Est-ce possible dans le cadre du modèle polyédrique?

Compilation et systèmes d'exploitation

- A la conception d'une application non temps-réel et non parallèle, on néglige les interférences du système d'exploitation.
- Plus le matériel est complexe (parallélisme, hiérarchie de mémoire) et plus ces interférences sont graves. Le coût d'une commutation de contexte devient prohibitif s'il faut sauver une mémoire scratchpad.
- Il vaut mieux procéder par partage des processeurs que par partage de temps. Par exemple, traiter des interruptions sur un vidéoprocesseur produit de la neige sur l'écran. Il vaut mieux prévoir un processeur spécialisé.

Conclusions

Il y a de nombreux problèmes ouverts et intéressants en compilation pour SOCs et systèmes embarqués.

- Diverses optimisations, parfois antagonistes (ressources, mémoire, communication, consommation, etc.)
- Une méthode d'optimisation *globale* est elle concevable? Elle nécessiterait la mise en œuvre de techniques de plus en plus lourdes : PLNE, méthode des points-test, optimisation non linéaire, CSP.
- Beaucoup de codes embarqués sortent des limites du modèle polyédrique. Il faut développer des méthodes d'approximation, *worst case*, etc.
- Explorer les relations avec les domaines connexes : vérification, cadençabilité par exemple.