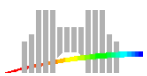
Compilation et Systèmes Embarqués

Paul Feautrier

ENS de Lyon
Paul.Feautrier@ens-lyon.fr

17 octobre 2007



Evolution de la technologie

- Augmentation au ralenti de la densité (doublement tous les trois ans)
- Stabilisation de la fréquence d'horloge (problèmes de dissipation)
- Saturation des astuces architecturales (pipeline, prédiction, hiérarchie mémoire)

La technologie pousse vers les systèmes parallèles homogènes (*multicore*) ou hétérogènes (*System on Chip SoC*).

Evolution des systèmes embarqués, I

Augmentation de la puissance de traitement

- La bande passante humaine n'est pas encore saturée (video, jeux)
- Les systèmes embarqués vont de plus en plus "parler entre eux"
- Un saut qualitatif : passer des application de *synthèse* aux applications de *reconnaissance*

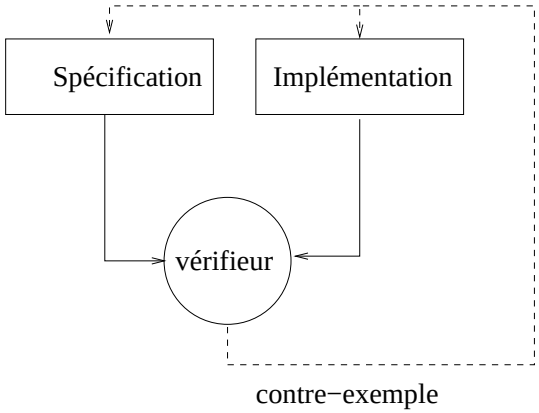
Evolution des systèmes embarqués, II

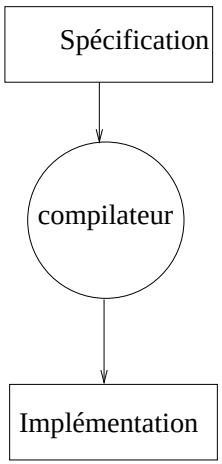
Contraintes non fonctionnelles

- Maîtriser la consommation électrique
- Maîtriser les coûts de fabrication (surface du chip)
- Augmenter la productivité des concepteurs (abstraction, réutilisation)
- Réduire la probabilité et la gravité des erreurs de conception

Deux approches : la compilation et la vérification.

Vérification ou compilation ?





- Les systèmes de preuve exploitent la redondance entre spécification et implémentation.
- Les spécifications vérifiables sont plus abstraites que les spécifications compilables.
- Les systèmes de preuve ne peuvent (actuellement) traiter que des petits exemples.
- Les compilateurs peuvent traiter de gros programmes, surtout s'ils possèdent un mécanisme de compilation séparée.

EVOLUTION DES COMPILATEURS

Que se passe-t-il en compilation ?

- Analyse approximative ou Interprétation Abstraite (Cousot)
- A la recherche de la régularité
  - Générateurs d'analyseurs syntaxique (pour mémoire)
  - Systèmes de réécriture
  - Programmation par contrainte

Interprétation abstraite

- Pour éviter les “mot à mot”, un compilateur doit “comprendre” le programme qu’il traduit
- Mais c’est impossible, sauf pour des langages de programmation triviaux (automates finis).
- On se contente d’une connaissance approchée, suffisante pour permettre une traduction de bonne qualité
- La méthode de l’interprétation abstraite est une technique systématique pour fabriquer des cadres d’approximation : on simule le programme source en simplifiant les ensembles de valeurs
- Exemple : on ne conserve que le signe des variables arithmétiques.

A la recherche de la régularité

- Les compilateurs anciens sont écrits sans modèle sous-jacent (*random code*)
- Exception : l'analyse syntaxique (vers 1970)
  - Modèle : les langages hors contexte
  - Outils : Lex et Yacc
- Objectif : trouver une approche analogue pour :
  - la génération du code objet
  - l'optimisation
  - la synthèse

Techniques de réécriture

- Trouver une représentation adaptée (très souvent, un arbre)
- Préparer une table de règles de réécriture

$$\textit{redex} \longrightarrow \textit{réduite}$$

  Le redex et sa réduite doivent être équivalents en un certain sens et peuvent contenir des *méta-variables*
- Ecrire un interpréteurs :
  - rechercher une instance d'un redex
  - la remplacer par l'instance correspondante de la réduite
  - poursuivre jusqu'à ce qu'aucune modification ne soit possible
- Exemples :
  - Générateur de code objet (Graham-Glanville)
  - Outil Stratego de Lex Agustejn

Programmation par contraintes, I

- Idée générale :
  - Rassembler toutes les contraintes du problème
  - Les soumettre à un solveur (programmation linéaire, optimisation combinatoire, branch-and-bound, ...)
  - Exploiter la solution
- Application
  - Contrainte I : le système objet doit implémenter la spécification source
  - Contrainte II : les performances (débit ou latence) doivent être supérieures à un objectif donné
  - Contrainte III : le système ne doit utiliser que les ressources matérielles disponibles
  - Contrainte IV : la consommation électrique du système doit être inférieure à une valeur donnée
  - etc ...

Programmation par contraintes, II

- Calcul à hautes performances :
  - Optimiser II (*performances*) sous les contraintes I (*correction*) et III (*ressources*).
- Systèmes embarqués :
  - Optimiser III sous les contraintes I et II
  - ou bien Optimiser IV (*énergie*) sous les contraintes I et II

Un peu de technique, I

L'équivalence de deux programmes est indécidable

- On se contente d'une condition *suffisante* d'équivalence.
- L'instruction R dépend de W si elle lit un résultat de W et si elle est exécutée après W.
- Pour que deux programmes ayant les mêmes instructions soient équivalents, il suffit que l'ordre des instructions en dépendance soit conservé
- Si $\theta(W), \theta(R)$ sont les dates d'exécution de W, R, il suffit donc que :

$$\theta(W) < \theta(R).$$

Un peu de technique, II

- La contraite de latence est très simple à écrire. Si L est la performance exigée :

$$\theta(S) < L$$

  pour toute instruction S.
- On obtient un système d'inéquations linéaires qui se résoud par des méthodes classiques.
- Si le programme contient des boucles, on ne peut plus énumérer les instructions, mais on s'en tire quand même en recherchant des formules explicites pour les dates d'exécution.

Parallélisme = plusieurs instructions qui s'exécutent à la même date

Un peu de technique, III

Le traitement de la containte de ressource est beaucoup plus difficile.

- Version simplifiée :

$$\forall t : \mathrm{Card}\ \{S \mid \theta(S) = t\} \leq P$$

où P est le nombre de ressources disponibles.

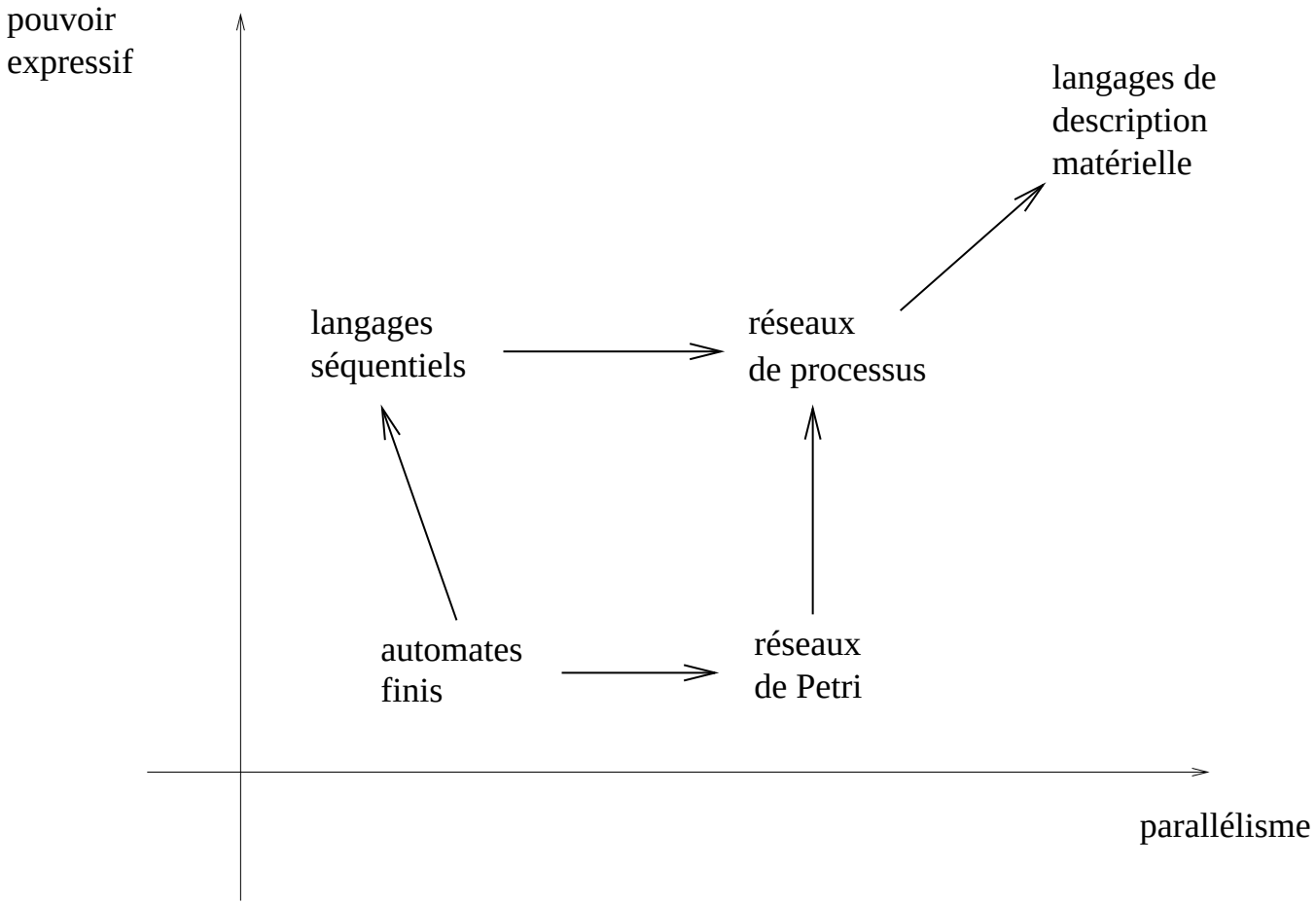
- On ne connait que des heuristiques :
  - tuilage
  - pipeline logiciel pour une boucle unique
  - introduction de dépendances virtuelles

REVUE DE FORMALISME

Formalismes de spécification des systèmes embarqués

- Descriptions incomplètes
  - Exemple : la logique temporelle
  - Utilisable seulement pour la vérification
- Description complète
  - Peut être implémentée
  - Paramètres de classification : pouvoir descriptif et représentation du parallélisme

Une classification



Automates finis

- Nombreuses variantes : RTL, diagrammes d'activité (UML), KISS
- Les langages synchrones se compilent directement en automates finis
- Plus :
  - Compilation très facile
  - Nombreux algorithmes efficaces (détermination, minimisation, etc)
  - Représentation limitée du parallélisme
- Moins :
  - Pouvoir expressif faible : un automate fini "ne sait pas compter"
- Extensions (automates à compteurs, etc.)

Langages séquentiels

- C, Matlab, Fortran XX
- Usage presque obligatoire pour la mise au point préliminaire
- Détection automatique du parallélisme possible à condition de contraindre le langage
- Nombreuses tentatives de synthèse directe à partir de C (SPARK, UGH, Catapult C)

Réseaux de processus

- Le concept de réseaux de processus communicants a son origine dans les premiers systèmes d'exploitation.
- Une première formalisation est due à Tony Hoare (CSP puis Occam). Un réseau à la Occam n'a pas nécessairement un comportement déterministe.
- C'est Gilles Kahn qui le premier a donné une condition suffisante de déterminisme (*Kahn Process Networks* ou KPN).
- Le modèle de Kahn a suscité de multiples réalisations (YAPI de Philips, les diagrammes d'interaction d'UML), des versions simplifiées (SDF, automates finis communicants, SDL).

Réseaux de processus : concepts

Un graphe dont les sommets sont des processus et les arcs des canaux.

- Processus : un programme séquentiel n'ayant accés qu'à ses variables locales et aux canaux auquels il est connecté.
- Canal : un espace mémoire partagé soumis à des disciplines d'accés :
  - lectures et écritures atomiques
  - lecture bloquante, écriture parfois bloquante
  - gestion en FIFO

Le problème du déterminisme

- Il est difficile de raisonner sur un système non déterministe
- Un système parallèle n'est pas en général déterministe
  - Variations incontrolées des vitesses d'exécution
  - Exception : les systèmes totalement synchrones
- Recherche de conditions *suffisantes* de déterminisme
  - Respecter les dépendances
  - Calcul instantané (cas des automates finis et des langages synchrones)
  - Conditions de Kahn : Canal = FIFO infini à un seul lecteur et un seul écrivain

Langages de description du matériel

VHDL, Verilog

- Langages complets, permettant la description du matériel *et* du logiciel
  - algorithmique (comportemental)
  - structurel
  - parallélisme
- Mais :
  - Seul un sous ensemble (mal défini) est synthétisable
  - Le problème du passage du comportemental au structurel est ouvert
- Nombreux travaux sur la sémantique de VHDL, plutôt orientés vers la vérification

COMPILATEURS ET SYSTEMES EMBARQUES

Quelques applications aux systèmes embarqués

- Ordonnancement symbolique
- Conception conjointe
- Synthèse de haut niveau
- Gestion de l'énergie

Parallélisme

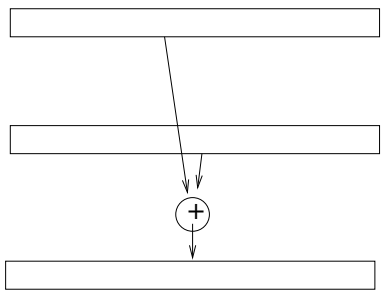
Les systèmes embarqués ont besoin de parallélisme :

- Performances
- Réactivité
- Economie d'énergie

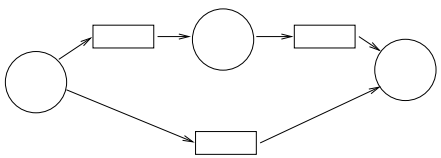
Mais on dit que la programmation parallèle est difficile et peu naturelle ! ! !

Programmation parallèle explicite

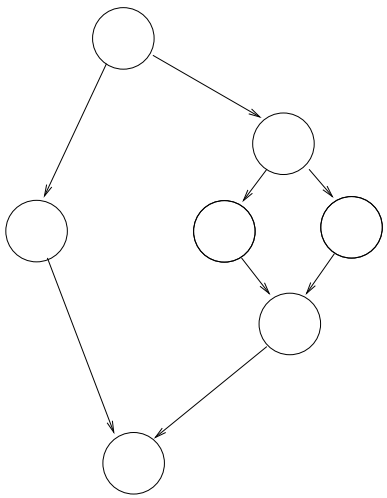
Il existe en fait des modèles de programmation parallèle naturels et faciles à utiliser.



Le parallélisme de données (algèbre linéaire)



Le parallélisme de flot (pipeline)



Le parallélisme "diviser pour régner"

Assistance du compilateur

- Vérifier les règles de programmation
  - Indépendance des tâches paralléles
  - Non recouvrement des tableaux
- ou bien, vérifier ou ajuster le degré de parallélisme (ce n'est pas parce qu'il y a plusieurs processus qu'il y a parallélisme)

Ordonnancement symbolique, I

- En général, une instruction est exécutée un grand nombre de fois (boucles, récursion)
- Un ordonnancement est alors une fonction qui donne l'heure d'exécution de chaque instance en fonction de son numéro
- La détermination des ordonnancement se ramène à un programme linéaire à condition que le programme respecte certaines conditions (modèle polyédrique) :
  - boucles à bornes affines,
  - indices affines

Ordonnancement symbolique, II

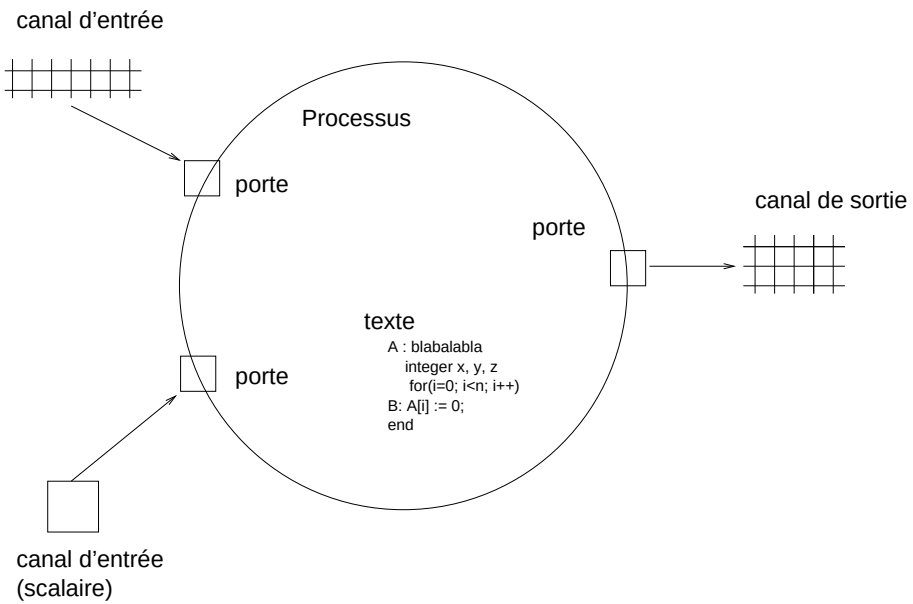
L'analyse des fonctions d'ordonnancement fournit une multitude de renseignements :

- Existence $\Rightarrow$ absence d'interblocage
- Inversion $\Rightarrow$ degré de parallélisme
- Bornitude des canaux
- Taille des tableaux (Darte et. al.)

Ordonnancement Symbolique et KPN

Les KPN ne sont pas obligatoirement dans le modèle polyédrique :

- Pour établir la correspondance entre émission et réception, il faut pouvoir compter les messages
- et il faut que les fonctions de comptage soient affines.



D'où l'introduction d'un modèle voisin, CRP (*Communicating Regular Processes*), où les canaux sont des tableaux adressables.

Conception conjointe, qu'est-ce ?

Codesign

- Partager la réalisation d'un système embarqué entre matériel et logiciel
- Développer simultanément un processeur, son compilateur et son logiciel

Pourquoi incorporer du logiciel dans un système embarqué ?

- Evolution très rapide des applications (marketing, mais aussi progrès technologique)
- Investissement initial moindre
  - "(presque) tout le monde sait programmer ..."
- Mise au point plus facile (prototypes, simulation)
- Une réalisation "tout logiciel" est facile ; les problèmes apparaissent quand on veut mélanger (performance, consommation)

Coopération matériel / logiciel

- Le point de départ idéal : une structure en processus
  - Si le programme de départ est monolithique, il faut la créer plus ou moins automatiquement (travaux d'Alex Turjan, Delft)
- Les communications entre logiciel et matériel sont tout simplement les *entrées / sorties*

Répartition matériel / logiciel

- Sauf architecture très particulière (processeur embarqué dans un FPGA) les entrées / sorties sont lentes
- Donc, réaliser en matériel les processus ayant un grand rapport calcul / données
- Plus un traitement est irrégulier, plus l'accélérateur qui le réalise ressemble à un processeur généraliste
- Donc, implémenter en logiciel les traitements irréguliers
- Prendre en compte également les performances demandées et l'énergie maximum disponible

Synthèse de haut niveau, qu'est-ce ?

Réaliser un circuit digital à partir d'une description en langage algorithmique.

- Le langage doit en général être contraint (e.g. pas de pointeurs) voire étendu (entiers de longueur non standard, calcul en virgule fixe, processus et canaux).
- Nombreux travaux :
  - UGH (Donnet, Augé, Pétrot, Greiner)
  - SPARK (Gupta, Nicolau)
  - Catapult C

Synthèse de haut niveau, les problèmes

- Transformer une spécification comportementale (séquentielle ou presque) en une description structurelle (tout fonctionne en même temps)
  - réinventer le compteur ordinal à chaque fois,
  - ou bien s'inspirer de la structure d'un processeur classique
- traitement des boucles :
  - exécution séquentielle, recherche du parallélisme dans le corps
  - déroulage partiel ou total
- traitement des test : contrôle ou spéculation

Le problème de la bibliothèque

A quel niveau décrire le résultat de la synthèse :

- A très bas niveau : on ne profite pas d'implémentations optimisées à la main
- A haut niveau : on peut utiliser des blocs optimisés (mémoire, opérateurs arithmétiques, etc)
- Le problème de la sélection des blocs est du même genre en plus difficile que la sélection d'instruction en compilation ordinaire.
- La deuxième approche correspond à une tendance actuelle des FPGA

Gestion de l'énergie, pourquoi ?

- Des applications de plus en plus complexes, alors que l'énergie consommée par opération élémentaire ne baisse plus
- Les objets portables sont alimentées par batterie, et les batteries ne suivent pas la loi de Moore
- Les prochaines technologies (45 nm) auront des courants de fuite élevés, et donc consommeront même si l'on ne s'en sert pas
- Même les objets non portables vont avoir des problèmes de consommation

Gestion de l'énergie, comment ?

- Utiliser des technologies à basse consommation (ce n'est pas de la compilation)
- Diviser l'objet en blocs fonctionnels aussi autonomes que possible (réseau sur chip, si possible asynchrone)
- Réduire au minimum la consommation des blocs non utilisés (coupure d'horloge, coupure de l'alimentation). C'est le compilateur qui sait !

Gestion de l'énergie, suite

- A traitement constant, réduire la fréquence ne réduit pas la consommation,
- ... mais réduire la fréquence permet de réduire la tension d'alimentation à peu prés dans les mêmes proportions
- Le compilateur connaît (en principe) les contraintes temporelles de l'application et peut ajuster (avec l'aide du système d'exploitation et du matériel), V et f à leurs valeurs minimales pour chaque bloc (problème d'optimisation faiblement non linéaire)
- Ajustage statique ou dynamique

Gestion de l'énergie, suite

- Toute optimisation qui réduit le nombre de cycles réduit la consommation. Toutes les optimisations classiques sont bénéfiques
- L'effet est amplifié si on remplace une opération forte consommatrice par une opération moins gourmande (exemple : un accès à la mémoire par un accès au cache).
- Dans les systèmes temps réels, on peut exploiter les erreurs de prédiction des durées d'exécution (en général surestimées) pour ralentir le processeur (optimisation dynamique par le système d'exploitation)
- Un problème ouvert : le fait de pouvoir ajuster la fréquence d'horloge des processeurs simplifie-t-il le problème de l'ordonnancement ? Relation avec les tâches malléables ?

CONCLUSION

Conclusion

Que doit savoir faire un compilateur pour systèmes embarqués ?

- Gérer le parallélisme
- Travailler à matériel variable
  - Exemple : l'allocation de registre, problème difficile pour un compilateur normal
  - En matériel, on peut toujours inférer un registre de plus
- Accepter des fonctions objectif exotiques, voire faire de l'optimisation multicritère
- Permettre et faciliter l'exploration d'architecture
- S'interfacer harmonieusement avec les outils de vérification

QUESTIONS ?