Construction de programmes pour systèmes embarqués

Compilation pour SOC

Paul Feautrier

Paul.Feautrier@inria.fr.

INRIA

Motivation

- Les chips deviennent des systèmes complets aussi complexes qu'un superordinateur d'il y a 5 ans;
- Il devient de plus en plus difficile de programmer un système embarqué "à la main".
  - problème d'optimisation combinatoire multicritères.
  - difficulté de la programmation "en assembleur" et inadéquation des langages de haut niveau.
- Question: les SOC posent-ils des problèmes de compilation spécifiques?

Revue d’architecture, I

Processeurs:

- Processeurs généralistes plus ou moins simplifiés.
- DSP.
- Circuits spécialisés reconfigurables (FPGA) ou non (ASIC).
  - Question : du point de vue du compilateur, y a-t-il une différence?

Revue d'architecture, II

Mémoire :

- Mémoire globale externe : temps d'accès élevé, forte consommation.
- Mémoire globale interne : temps d'accès non négligeable, dû au passage par le réseau, consommation plus faible.
- Mémoire locale, parfois organisée en cache. Le fonctionnement du cache est difficilement prédictible. La gestion d'une mémoire locale, *scratchpad* doit être assurée par le programmeur ou le compilateur.

Gestion (matérielle/logicielle) d'une hiérarchie de mémoire à multiples niveaux.

Communications, I

- Toutes ces unités doivent être interconnectées entre elles et avec l'extérieur du chip.
- Echange d'information à travers une mémoire (simplicité, rapidité) mais peu extensible.
- Chemins de données spécialisés.
- Communication par réseau.
  - Latence plus élevée, nécessitant le regroupement (ou vectorisation) des messages.
  - Problème du placement des données et des calculs,
  - Garantir un temps de réponse devient difficile.

Communications, II

- Question : La progammation d'un réseau interne est-elle différente de celle d'un réseau externe (Myrinet ou HSL)?
  - Primitives de communication?
  - Méthodes de synchronisation?
  - Grain = nombre d'instructions exécutées dans le temps de transmission d'un message. De l'ordre de plusieurs milliers pour un réseau standard (HSL), probablement nettement plus faible (10 à 100) pour un réseau interne, donc nettement plus favorable.

Problèmes des applications embarquées

- Parallélisme : à la fois pour obtenir les performances requises et pour réduire la consommation d'énergie.
- Conception hiérarchiques : parce que nombres d'algorithmes existent "tout faits" sous forme d'IP ou de procédures, et aussi parce que le temps de compilation risque d'être trop élevé.
- Contraintes temps réél : plus ou moins dures (video, son, etc.).
- Optimisation multicritère : performance, coût, consommation, *time to market*.

Classement des applications embarquées

- Systèmes réactifs. Asynchrones, apériodiques, en général contraintes temps rééel dures.
- Calcul “au fil de l’eau” (*streaming*). Traitement du signal, audio, video, cryptographie. Périodique ou quasi-périodique, peut être synchronisé. Contraintes temps rééel, haut débit d’information.
- Calcul Haute Performance, souvent comme composant quasi-autonome d’un système d’un autre type : analyse ou synthèse d’image, simulation.
- Concept de régularité : les opérations exécutées et les données utilisées peuvent être prévues à la compilation.
- Beaucoup de systèmes sont mixtes. Example : un lecteur de CD-Audio a une composante “au fil de l’eau” (le traitement du signal audio) et une composante réactive (l’interface avec l’utilisateur).

Spécification des applications embarquées

- Un compilateur doit évidemment partir d'une spécification de l'application pour générer le programme objet! Cette spécification peut être de plus ou moins haut niveau.
- Le compilateur doit également disposer d'une description de l'architecture cible (nombre et type des processeurs, latences, description des mémoires, moyens de communication, etc).
- Il est commode que le langage de spécification puisse exprimer le parallélisme, mais il n'est pas obligatoire que le parallélisme de la spécification soit le même que celui de l'implémentation.
- Le programmeur doit pouvoir suppléer une phase de compilation défaillante ou inexistante.
- Candidats : langages à assignation unique (Alpha) limités aux application régulières, ou langages à base de réseaux de processus de Kahn, plus généralistes.

Compilation : vue cavalière

- Gestion du parallélisme.
  - Partitionnement, calcul des dépendances, ordonnancement, placement.
- Gestion de la mémoire.
  - Réduction de la mémoire, spécialement importante dans le cas des programmes en assignation unique.
  - Optimisation de la localité, pour caches et mémoires locales.
- Gestion des communications et/ou des synchronisations.
- Génération de code.
  - Langage de bas niveau (C) + primitives.
  - Langage de simulation (SystemC).
  - Langage binaire.

Partitionnement

- Découper l’application en tâches et confier chacune des tâches à un élément du système.
- Choisir entre des processeurs généralistes et un ASIC.
- Il n’existe pas de méthode automatique de partitionnement. On fait de l’exploration d’architecture.
- Des automatisations partielles sont sans doute possible.

Ordonnancement, I

- Principe: on attribue à chaque opération une date de lancement. Toutes les opérations qui ont la même date de lancement sont exécutées en parallèle.
  Contrainte de causalité:

$$u \ \delta \ v \Rightarrow \theta(u) + \partial(u) \leq \theta(v).$$

  Contrainte de ressources : le nombre d'opérations exécutées en parallèle est limité par le nombre de ressources disponibles.
- $\theta(u)$ fonction affine des compte-tours des boucles englobant u.
- C'est ici que l'on peut introduire les contraintes temps-réél.

Ordonnancement, II

Pas de contraintes de ressources.

- La fonction affine $\theta(v) - \theta(u) - \partial(u)$ doit être non-négative dans le polyèdre $u \ \delta \ v$.
- On peut se ramener à un problème fini par le lemme de Farkas ou la méthode des sommets.
- Tous les algorithmes de parallélisation connus (y compris Kennedy et Allen) sont des variantes de l'algorithme d'ordonnancement obtenues en « approximant » la relation de dépendance [Darte et Vivien 1999]

Ordonnancement, III

Avec contraintes de ressources.

- C’est ici que peuvent s’introduire les processeurs hétérogènes.
- Problème beaucoup plus difficile.
- Heuristique quand le degré de parallélisme est plus grand que le nombre de ressources : on ordonnance sans contraintes de ressources, puis on tuile.
- Solution exacte (ou à dégradation limitée) dans le cas à une dimension (une seule boucle) : pipeline logiciel.
- Pas de méthode connue pour le pipeline logiciel à plusieurs dimensions.

Placement, I

Sans contrainte de ressources.

- Principe: on attribue à chaque opération un « processeur virtuel ». Toutes les opérations assignées au même processeur sont exécutées en séquence.
- De même, chaque donnée est attribuée à (la mémoire d') un processeur. Pour éviter des communications, chaque calcul doit être effectué par le processeur qui en détient les données.

$$\pi(u) = \Pi(f(u))$$

- Système d'équations linéaires et homogènes à résoudre par une méthode de Gauss incrémentale.
- Certaines contraintes ne peuvent pas être satisfaites et engendrent des communications (synchronisations) résiduelles.

Placement, II

Avec contraintes de ressources. Qu'est-ce?

- Heuristique : on affecte plusieurs processeurs rééls à un seul processeur physique. Cette distribution peut être faite soit par le compilateur soit par le système d'exploitation (cas de la Connection Machine).
- Est-il possible de prendre en compte les contraintes de ressource dès la détermination des fonctions de placement?
- Comment construire le programme parallèle à partir d'un placement? Le problème est résolu pour le cas d'un seul type de ressource, mais pas pour plusieurs types.

Gestion de la mémoire

Trois ordres de problèmes :

- Cohérence de la hiérarchie de mémoire.
- Economiser la mémoire, spécialement quand la spécification est équationnelle et quand il y a des signaux infinis.
- Gestion de la localité.

Cohérence

- Les protocoles de cohérence classiques ont pour objectif essentiel la transparence.
- “Quand un processeur lit une cellule de mémoire, il doit obtenir la valeur la plus récemment écrite, que la cellule soit cachée ou non”.
- Quand le logiciel est conscient du parallélisme et de la présence d’une hiérarchie de mémoire, la question ne se pose plus (comparer à la gestion des registres).
- Problème marginal : accès à des fragments de la ligne de cache ou de mémoire locale.

Economiser la mémoire

- Effet important sur le coût et la consommation.
- A partir d'un ordonnancement, il est possible de déterminer la durée de vie de chaque valeur.
- On peut alors récupérer les cellules de mémoire qui ne sont plus utiles et y mettre d'autres valeurs.
- Solution par réindexation (Chamski, Lefebvre), par projection (Rajopadhye, Quilleré), couplage avec l'ordonnancement (Vivien et. al.).

Caches et mémoires locales

Deux mécanismes génants:

- associativité limitée d'où des interférences.
- le mécanisme de remplacement (FIFO, LRU, RANDOM, etc.).rend la prédiction des performances difficiles.

Mémoire locale.

- Bloc de mémoire ayant la taille et les performances d'un cache, mais sans système de gestion transparente.
- Question : comment la mémoire locale est elle vue depuis le logiciel?
- Choisir le contenu de la mémoire locale.
- Organiser les données de la mémoire locale.
- Ecrire les codes de remplissage et de vidage.
- Modifier le code de calcul.

Génération de code

- Les algorithmes d'ordonnancement et de placement ne donnent pas directement le programme parallèle.
- Exemple : programme ordonnancé:

  for $t = 0, L$
  forall $\{u|\theta(u) = t\}$
  do u

- Il faut parcourir dans un ordre quelconque le polyèdre $\{u|\theta(u) = t\}$ (le front à la date t).
- On détermine les bornes inférieures et supérieures de chaque boucle par des méthodes de programmation linéaire [Ancourt et Irigoin, Darte, Collard et Risset, Pugh et. al., Lam et. al., Lengauer et. al., Xue, Quilleré].

Génération des communications

- Une fois un placement connu, on peut caractériser les données qui doivent être échangées entre unités fonctionnelles. Elles forment des polyèdres ou des images de polyèdres.
- On peut parcourir le polyèdre des données à émettre. A chaque point du parcours, on copie une donnée dans un buffer d'émission.
- De même, on parcours le polyèdre des données reçues et on copie une valeur du buffer de réception vers l'espace des données.
- Noter que ce code n'est pas zéro-copie. Il ne peut y avoir zéro-copie que si les données lues et écrites sont "isomorphes".

Génération des *netlists*

Problème beaucoup plus complexe.

- Il faut faire des hypothèses sur la nature de l'architecture :
  - Multiprocesseur,
  - Pipeline
  - Réseau systolique (MMAlpha)
  - Ensemble d'automates.
- Cette phase ne peut pas être entièrement automatisée à l'heure actuelle.

Où sont les problèmes?

- Beaucoup de problèmes ont été résolus dans d'autres contextes : les solutions ne demandent qu'à être adaptées: placement, ordonnancement sans contraintes de ressources, génération de code, économie de la mémoire, partiellement la localité, etc?
- Prendre en compte les problèmes de ressources.
- Beaucoup de problèmes ont été résolus isolément. Or en général les diverses optimisations interagissent. Il faut unifier les contextes et si possible combiner les optimisations.
- Prise en compte des applications irrégulières.

Intégration

- Nombre de problèmes ont déja été résolus, et il existe même des logiciels.
- Mais ces logiciels sont autonomes, font des hypothèses variables sur le programme source, ne contiennent pas de phase d'analyse ni de phase de génération de code (explication : la durée des thèses françaises).
- Le problème est d'interconnecter ces logiciels et de les intégrer dans une plateforme d'analyse / optimisation / parallélisation / génération pour système enfouis.
- Travail énorme, mais pas irréalisable vu l'abondance d'outils (logiciels libres) et de compilateurs Open Source (gcc, etc.).
- Nécessité de langages communs pour la représentation intermédiaire des programmes et des rachitectures.
- Proposition : langage(s) ad hoc au-dessus de XML.

Parallélisation Modulaire

- La parallélisation modulaire n'est pas la même chose que la parallélisation interprocédurale.
  - En compilation séparée, le problème est de savoir ce qu'il faut stocker « à coté » d'un module pour pouvoir compiler le reste du programme.
  - En général, on note, pour chaque fonction, son type et le type de ses arguments. C'est insuffisant pour paralléliser.
- 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?

Applications irrégulières

- Applications dont il est impossible de prédire le comportement à la compilation :
  - Le comportement est trop complexe pour nos moyens d'investigation actuels.
  - Le comportement dépend des données.
- Ce type d'application devient de plus en plus fréquent dans le monde des systèmes embarqués :
  - Compression/décompression de données. La quantité de données à traiter / transmettre ne peut être prédite.
  - Génération d'image par triangles / textures. On ne sait pas a priori combien il a de triangles par image.

Méthodes “worst case”

- On programme pour le cas le plus difficile.
  - Le taux de compression est maximum (ou minimum).
  - Les deux branches d’un test sont éxécutés.
  - Que faire pour une boucle `while`?
- La méthode aboutit en général à un énorme sur-dimensionnement.

Méthodes Probabilistes

- On remplace les bornes par des valeurs moyennes.
- On emploie des méthodes de spéculation. On fait comme si l'on était dans le cas favorable, quitte à corriger si l'on s'est trompé.
- Il n'est plus possible de donner des garanties "absolues".
- Mais, une garantie "absolue", est-ce que cela existe?
- La réponse dépend en partie du contexte. Générer une mauvaise image tous les 10^6 frames est acceptable. Crasher un avion tous les 10^6 atterrissages ne l'est pas.

Est-il possible de rechercher les régularités cachées?

Plan de Travail

Il existe sans doute des problèmes de compilation spécifiques aux systèmes embarqués. Je ne suis pas convaincu qu'il en existe de spécifiques aux SOC.

- Court terme : résoudre quelques problèmes urgents : contraintes de ressources, gestion des mémoire locales.
- Moyen terme : Construction d'une plate forme, intégration des optimisations.
- Long terme : Traitement des codes irréguliers. Que font les experts?