Optimisation de codes embarqués bas niveau

Survol général

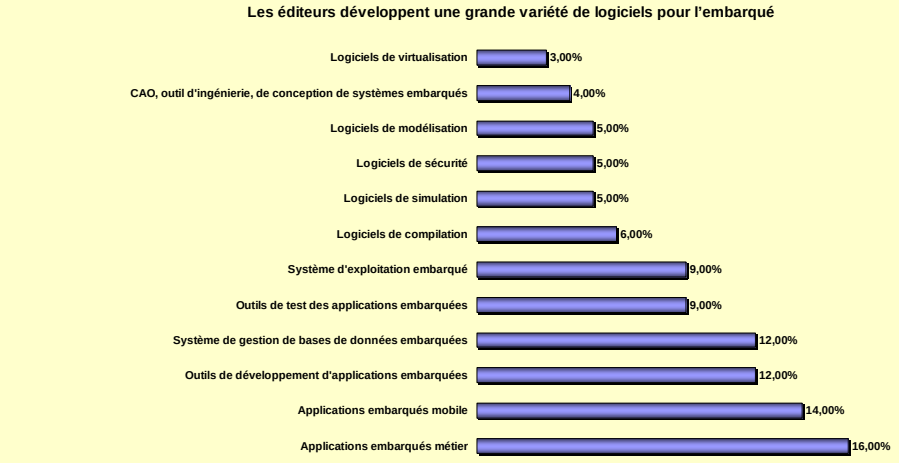
Sid TOUATI

Ecole thématique ARCHI07, mars 2007

Plan

- Enjeux de l'optimisation de codes embarqués
- Optimisations pour la vitesse
- Optimisation pour la mémoire
- Optimisations pour l'énergie et la puissance

Les logiciels embarqués



Sources (mars 2007) : IDC Consultants

Ancienne thématique, nouveaux enjeux !

- Les systèmes embarqués à l'origine : *control-driven*
  - Temps réel dur, langages spécifiques, marché restreint
- De nos jours : *data-driven*
  - Temps réel mou, langages impératifs, marché grand public

Diversité des systèmes embarqués

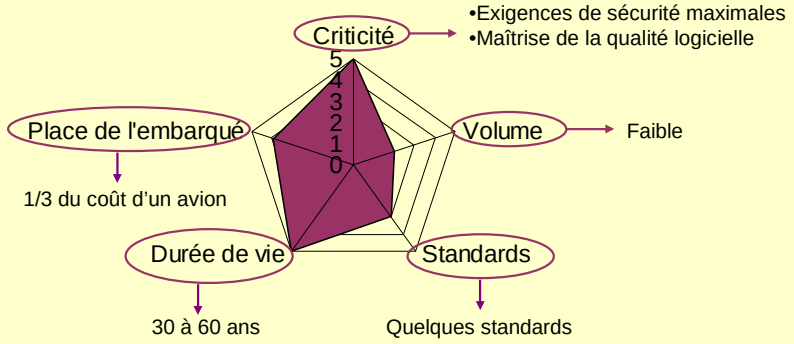
- Aéronautique
- Automobile
- Transport ferroviaire
- Construction électrique
- Télécoms
- Cartes à puce

La diversité s'exprime par plusieurs facteurs :

1. Criticité : performances, fiabilité, autonomie. Attention, un frein ABS qui ne fonctionne pas c'est plus grave qu'un lecteur vidéo qui ne décode plus le son
2. Volume, nombre d'unités à déployer
3. Standards : langages, systèmes, méthodologie de conceptions, protocoles, etc.
4. Durée de vie du système embarqué
5. Place de l'embarqué dans le produit

Les systèmes embarqués dans l'aéronautique

Aéronautique



Sources (mars 2007) : Pierre Audoin Consultants

Des enjeux liés à la qualité du logiciel - mais aussi des enjeux économiques

Croissance de la **complexité** et **réduction de coûts**

Ouverture des systèmes : maintenir les niveaux de sécurité

Développer des architectures qui permettent la maintenance et la mise à jour des systèmes sur le long terme

Maîtriser la complexité des systèmes

Vers une utilisation de la génération automatique de code qualifiée et de la modélisation

"Artisanat de luxe" versus mode industriel

Futur: la différentiation passera en partie par les **services apportés sur base logicielle.**

Une industrie du logiciel embarqué en évolution

Équipementiers peu nombreux

Stratégie avionneurs : garder des développements critiques en interne (15/20%) et sous-traiter le reste :

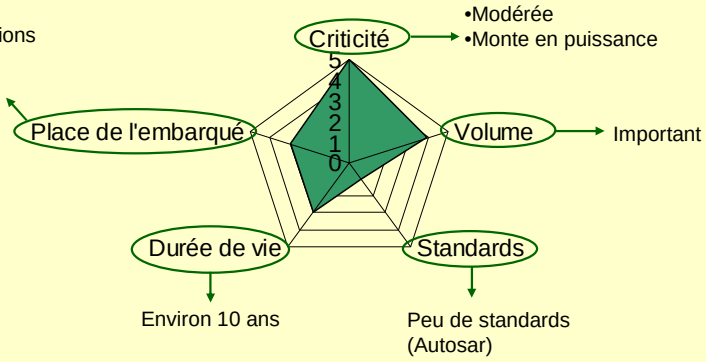
équipementiers, SSII, (éditeurs)

Évolution des rôles SSII/éditeurs vers plus de responsabilités -> Nouvelle génération d'équipementiers "pure players" ?

Les systèmes embarqués dans l'automobile

Automobile

•Part des systèmes embarqués dans le coût : environ 15%
• 80% des innovations
• 10% de la RD



Sources (mars 2007) : Pierre Audoin Consultants

Des enjeux économiques mais aussi marketing

Coût : le nombre de calculateurs (boîtiers) par voiture plafonne, mais le nombre de fonctions croît de façon exponentielle.

Remplacer une fonction mécanique par une fonction électrique/électronique

Quelle valeur à la fonction?

Viabilité en termes de coûts, encombrement, sûreté de fonctionnement…?

L'innovation est un argument de vente dans l'automobile

Une industrie du logiciel embarqué peu mature

Le constructeur est initiateur/leader dans la conception et le transfert de technologies L'organisation de la

sous-traitance est très structurée

Équipementiers de rang 1 - Marché très concentré, très forte concurrence (diminution

de coûts, innovation) - peu de culture logicielle

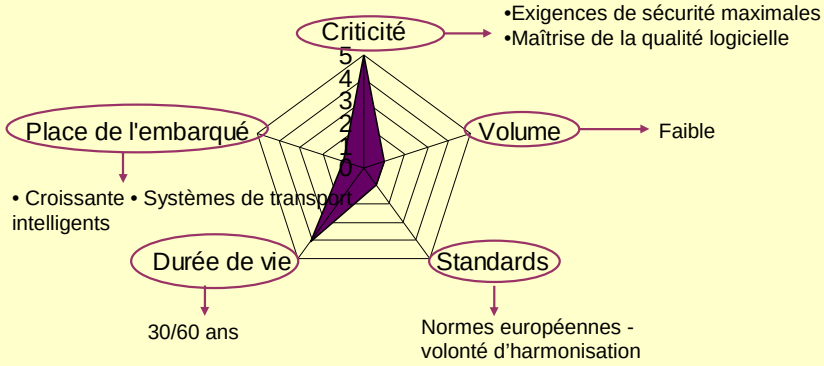
Équipementiers de rang 2 - développement/production de sous-ensembles

Le logiciel entre dans le cœur de métier des équipementiers

Modes de sous-traitance : SSII en régie, peu d'achats aux éditeurs

Les systèmes embarqués dans le ferroviaire

Ferroviaire



Sources (mars 2007) : Pierre Audoin Consultants

Des enjeux essentiellement économiques

Besoins d'interfaçage, **interopérabilité** entre la partie roulante et la partie fixe. Outils de développement

différents

EN 50128 recommande: « Réutiliser autant que possible les logiciels déjà vérifiés/validés. »

Les projets sont « uniques »=> systèmes différents (et propriétaires) => peu de réutilisation.

Les grands industriels sont dominants : pas de standards dans le logiciel. Pas d'interchangeabilité entre

les constructeurs. Mais évolution en cours (Modurban). Besoin fort de normalisation des couches logicielles.

Transferts de technologies difficiles entre l'industriel et l'opérateur.

Une industrie du logiciel embarqué en construction

Les opérateurs de transport expriment les besoins (performances, fonctionnalités…) et travaillent avec les

industriels pour valider les technologies.

Les outils de développement/méthodologies dépendent de l'industriel.

Peu de sociétés tierces s'engagent sur des niveaux de sécurité élevé => maintenance réalisée en interne ou

par l'industriel.

Évolution du ferroviaire européen vers une ingénierie des composants: les composants sont associés à des

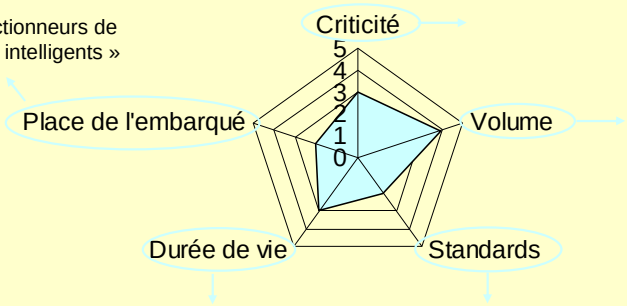
certificats définissant un contexte d'utilisation.

• **Face à face industriels/opérateurs**

Les systèmes embarqués dans la construction électrique

Construction électrique

•Beaucoup de fonctions électroniques
•Capteurs - actionneurs de plus en plus « intelligents »



Volumes importants (grand public)

15-20 ans pour des postes/automates

Sources (mars 2007) : Pierre Audoin Consultants

Des enjeux liés à la compétitivité des produits

Automates, basse tension: « Le logiciel permet de lutter contre la banalisation du produit »

Durée de vie : garantir au minimum l'iso-fonctionnalité et l'inter-fonctionnement avec les autres systèmes sur la durée

Augmenter le niveau de service offert par les fonctions électroniques - Améliorer leurs performances

L'introduction du logiciel ne doit pas se faire au détriment de la **qualité** (sécurité, fiabilité, disponibilité).

Volume de développement logiciel en forte croissance pour des budgets de R&D constants - gérer la

complexité

Augmenter la productivité : augmenter la réutilisation, développer l'utilisation d'outils indépendants de la

plateforme

Une industrie du logiciel embarqué peu mature

L'essentiel des développements est réalisé en interne

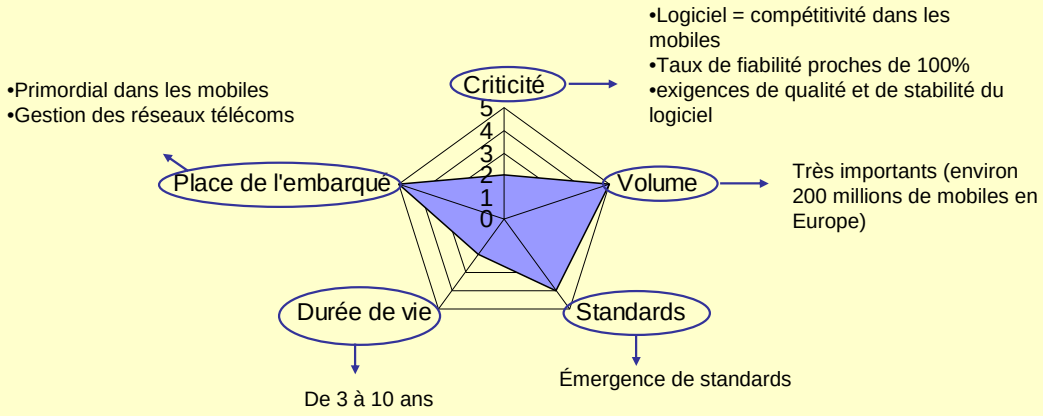
Ateliers logiciels: des produits « maison » souvent liés au matériel

Mode d'utilisation de la sous-traitance logiciel : en général peu

mature, mais il existe de véritables partenariats

Les systèmes embarqués dans les télécoms

Télécoms



Sources (mars 2007) : Pierre Audoin Consultants

•**Des enjeux essentiellement économiques et marketing**

• **Pression sur les prix très forte** : solutions de développement à bas coût (offshore).

• **Convergence** : architectures (IMS) qui garantissent une interopérabilité des réseaux fixes/mobiles,

•voix/données.

• **Intelligence des réseaux** : doivent porter la convergence ET réduire les coûts opérationnels des

•opérateurs ce qui augmente la complexité des solutions déployées

• **Une industrie du logiciel embarqué relativement mature**

• Forte concentration sur le secteur

• Les ressources externes sont en priorité dédiées au développements de nouveaux produits, ou pour

•les activités d'intégration et de test notamment dans le cadre de certification européennes

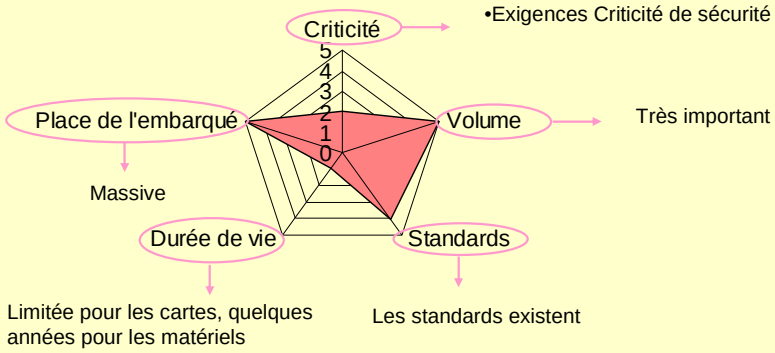
• Recours à la sous-traitance fréquent mais souhait de maîtrise du savoir faire.

• De nombreux éditeurs présents sur la mobilité et des niches de compétences applicatives

Forfait, offshore, édition de fonctions innovantes : un secteur très dynamique

Les systèmes embarqués dans la carte à puce

Cartes à puces



Sources (mars 2007) : Pierre Audoin Consultants

Des enjeux technologiques et économiques

Technologies de micro noyaux. Les contraintes ne permettent pas d'utiliser les outils tels que embedded Linux

Complexité croissante des produits : de plus en plus complexe de faire du test avec des

méthodes classiques. L'automatisation et la « sécurisation » des test est un enjeu majeur

Nécessité d'améliorer les **capacités de portabilité** pour passer d'une plate-forme hardware à une

autre

En outillant le développement, il est possible d'améliorer le **« time to market »**

Une industrie du logiciel relativement mature mais propriétaire

La sous-traitance est utilisée principalement sur des phases de customisation. La sous traitance

n'est pas utilisée pour les parties technologiques centrales

Les équipementiers sont peu nombreux et les développements restent souvent propriétaires

Forte croissance des besoins : identification, personnalisation,…

Un écosystème logiciel déjà substantiel

Caractéristiques des systèmes embarqués « grand public »

- De modestes capacités de stockage et un encombrement réduit (miniaturisation);
- Un nombre très élevé d'unités à déployer;
- Une forte dépendance vis-à-vis des aspects de consommation d'énergie et de puissance électrique;
- Des applications exigeant des performances garanties ou temps réel « mou ».

•tout gain, qui serait négligeable dans le cas du calcul à haute performance, entraînerait une grande économie d'échelle dans le domaine de l'embarqué ;

•toute erreur de programmation (éventuellement issue du compilateur) a un coût (financier, commercial) important ;

• une très forte dépendance vis-à-vis des aspects de consommation d'énergie et de puissance électrique;

• des applications exigeant des performances garanties ou temps réels, le but n'étant pas de générer des tâches rapides, mais des tâches qui arrivent à l'heure. Ceci est un aspect important : dans un système embarqué, il est souvent inutile de minimiser le temps d'exécution au plus bas niveau possible. Il suffit d'obtenir un temps d'exécution ne dépassant pas une certaine borne (contrainte temps réel).

La notion de « temps réel » doit être maniée avec précaution. En effet, il faut toujours préciser l'échel de temps. Car du temps réel avec des nano ou mili-secondes n'a rien à voir avec du temps réel à l'échelle des minutes ou des heures.

Facteurs limitatifs pour l'utilisateur grand public

- Coût :
  - Plusieurs appareils électroménagers coûtent moins cher que le processeur pentium. Pour un produit à ≈200 €, le budget dédié au calcul est à peu près de 50 €.
- Taille :
  - plusieurs produits embarqués sont plus petits que les processeurs généraux (ex: des téléphones portables sont plus petits que le Xeon)
- Autonomie de la batterie

Synthèse générale sur les systèmes embarqués (tout domaine confondu)

Explosion des besoins : doublement du nombre d'objets intelligents et de la complexité embarquée tous les 2/3 ans

Chaque industrie est confrontée à des **contraintes spécifiques**

Mais les **enjeux sont similaires**

Maîtrise des coûts

Maîtrise de la complexité

Pérennité des solutions développées

Utilisation des processeurs embarqués

- Electroménager: audio, vidéo, caméras and divertissements domestiques.
- Périphériques d'autres ordinateurs généraux: contrôleurs de disques, modems, cartes vidéos.
- Communication: téléphonie and réseaux de données.
- Automobile: sécurité, contrôle de moteur, navigation (GPS), freinage ABS, etc.

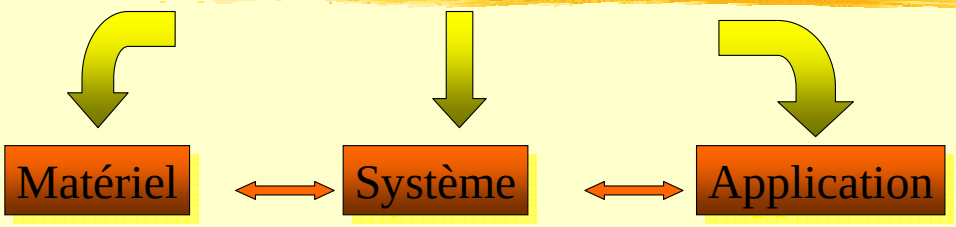
Types des processeurs embarqués

- Processeurs généraux embarqués
  - ARM, MIPS, 68K, x86, ST200….
  - Bus de données à 32 bits.
- DSP
  - TI, Motorola, Agere/Lucent, Analog Devices, ST.
  - Opérations arithmétiques très efficaces dans les noyaux de codes orientés « boucles de calculs ».
- Microcontrôleurs
  - Industrie électronique – souvent conçus pour une application unique.
  - 8 bit (70%), 16 bit, 32 bit.

Critères d'optimisation

- Vitesse de traitement
  - Temps réel mou
- Energie
  - Consommation
  - Puissance, dissipation de chaleur
  - Efficacité énergétique
  - Autonomie de batterie – énergie consommée pour effectuer une tâche en un temps fixé.
- Mémoire
  - Coût limité, taille de code et de données bornée.

Les leviers d'optimisation d'un système embarqué



- Electronique
- Architecture
- Circuits reconfigurables

- JIT
- Gestion de processus
- Interactions

- Compilation
- Expert-Programmeur

Au niveau applicatif, nous avons principalement deux leviers:

•Optimisation automatique de code : compilation

•Optimisation manuelle ou semi-automatique de code : une armée d'experts (en Sibérie)

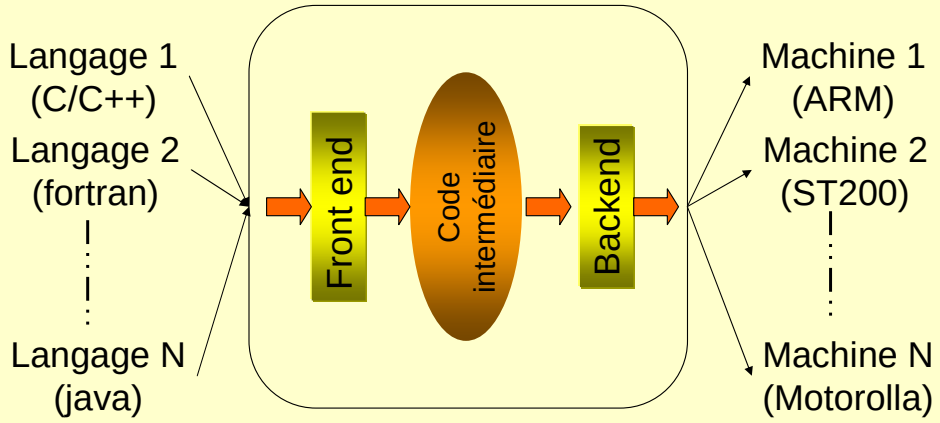
Charte pour l'optimisation automatique de code

- Toute technique d'optimisation de code ne doit pas modifier sa sémantique, *i.e.*, le résultat d'exécution
- Pas de tolérance aux bugs !
  - utiliser les méthodes formelles ?
  - En réalité, y a des bugs dans les compilateurs!!
- Vœux pieux : l'optimisation de code doit être rapide
  - la complexité de l'algorithme utilisée doit être polynomiale en fonction de la taille du programme compilé
  - Ce n'est pas toujours le cas !

Quelle partie de code optimiser ?

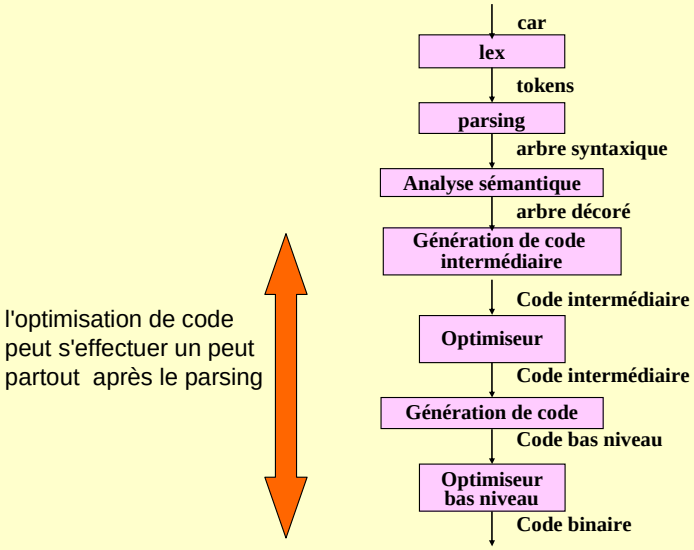
- Les parties les plus souvent exécutées : boucles, fonctions appelées plusieurs fois, etc.
- Comment détecter les parties « chaudes »?
  - Profilage
  - Dépendances vis à vis des données en entrée
  - Souvent, tout optimiser automatiquement sans discernement !
  - Parfois, les parties « chaudes » du programme peuvent être traitées au cas pas cas par un expert

Structure « scolaire » d'un compilateur



A des fins d'économie, on essaye tant bien que mal de porter un compilateur existant vers différents langage et architectures .

A quelle étape de compilation



De nos jours, un compilateur optimisant contient des dizaines de modules d'optimisations. Dans le monde de la recherche académique et d'ingénierie, des dizaines de techniques sont publiés tous les ans, mais ne sont pas toutes portées à l'industrie.

Les techniques d'optimisations de codes

- Il y en a des milliers publiées, des centaines implémentées, et la recherche continue de progresser…
- Inter-influence : elles se marchent sur le pied
- Pour chaque technique, il y a des tas d'algorithmes proposés, avec divers efficacités difficilement mesurables.
- Dans ce cours, je vous explique ce que font quelques techniques usuelles, pourquoi et dans quel cas elles sont utilisées, leurs effets attendus, mais je ne vous explique pas comment elles procèdent!
- Je ne vous donnerai pas non plus les évaluations « fantaisistes » de leurs impacts précis!

Que dit la science sur les limites mathématiques ?

- Compilation statique : la notion d'un code « optimal », pour un jeu de données quelconque en entrée, n'existe pas
- Compilation itérative : il est impossible de trouver automatiquement la meilleure combinaison des techniques d'optimisations de codes
- Equivalence entre programmes : il est impossible de détecter automatiquement l'équivalence sémantique entre deux algorithmes

Corolaire : le recours à des experts humains est inévitable.

Il y a des centaines de techniques d'optimisations de codes, dont les effets sont interdépendants.

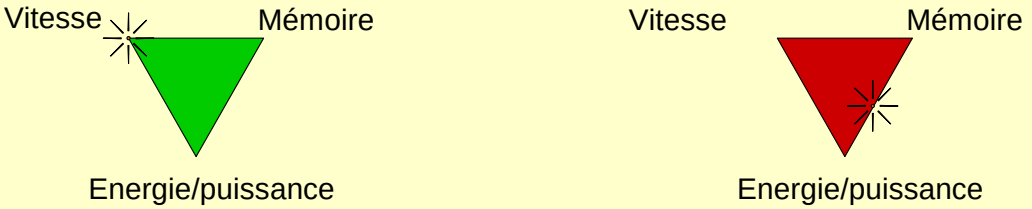
Mystères scientifiques

- Relation (même qualitative) taille de code et vitesse d'exécution
  - Actuellement, il n'y a que des réponses *ad-hoc,* au cas par cas !
- Représentation minimale d'un programme en mémoire
  - Complexité de Kolmogorov

En pratique

- Les compilateurs optimisant proposent un grand nombre d'options laissées à l'utilisateur
- Certaines options sont regroupées dans -O2, -O3, etc.
- Aucune garantie d'efficacité, même si c'est démontré sur des programmes benchmarks.

Convention visuel dans ce cours



Tendances d'impacts positifs

Tendances d'impacts négatifs

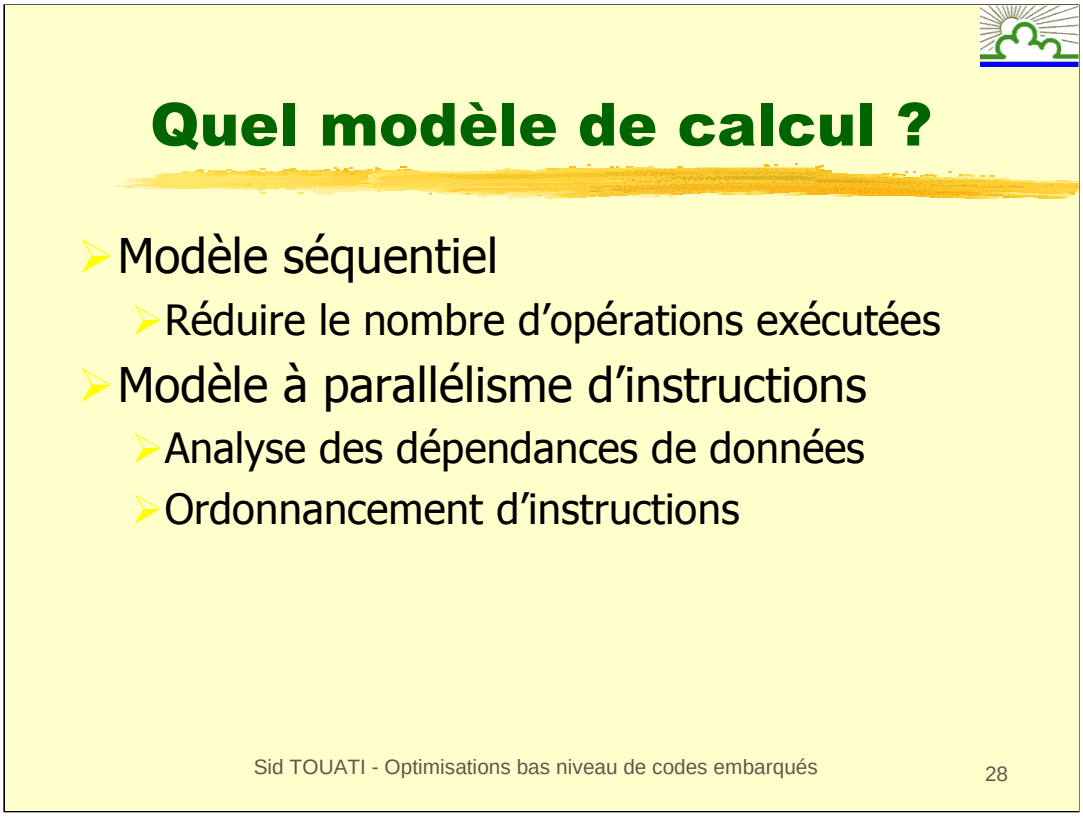
L'étoile sur le triangle vert donne la tendance positive d'une technique d'optimisation de code par rapport aux trois critères de performances dans les logiciels embarqués (vitesse, taille mémoire, énergie)

L'étoile sur le triangle rouge donne la tendance négative d'une technique d'optimisation de code par rapport aux trois critères de performances dans les logiciels embarqués (vitesse, taille mémoire, énergie)

L'absence d'étoile ou de triangle veut dire pas de tendance claire (dans un sens ou dans un autre).

Plan

- Enjeux de l'optimisation de codes embarqués
- Optimisations pour la vitesse
  - Modèle d'exécution séquentiel
  - Modèle d'exécution ILP
- Optimisation pour la mémoire
- Optimisations pour l'énergie et la puissance



Réduire le nombre d'opérations exécutées : un domaine de recherche mature.

Plan

- Enjeux de l'optimisation de codes embarqués
- Optimisations pour la vitesse
  - Modèle d'exécution séquentiel
  - Modèle d'exécution ILP
- Optimisation pour la mémoire
- Optimisations pour l'énergie et la puissance

Pour une exécution séquentielle

- Toutes les optimisations scalaires traditionnelles :
  - élimination de sous expressions communes, élimination de code mort, réduction en force, propagation de constante, déplacement d'invariants de boucles, etc.

Initialement, ces techniques ont été conçues pour l'accélération des calculs. Par effet de bord, elles peuvent être aussi bénéfiques pour l'énergie et la taille mémoire.

Exemples simples : optimisations scalaires

```
int foo() {
 a = x+y-z;
 b = x+y-z;
 c = (x+y-z)<<2;
 return c;
 b = 24;
}
```

```
int foo() {
 a = x+y-z;
 b = a ;
 c = a << 2;
 return c;
 b = 24;
}
```

```
int foo() {
 a = x+y-z;
 b = a;
 c = a << 2;
 return c;
 b = 24;
}
```

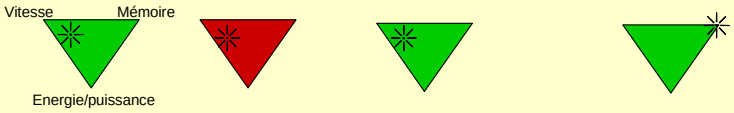
```
int foo() {
 a = x+y-z;
 b = a;
 c = a << 2;
 return c;
 b = 24;
}
```

code initial

Sous-expression commune

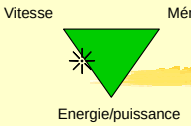
code mort

code inaccessible



Le code mort est tout code exécuté mais qui ne contribue pas au résultat du programme. Une analyse de dépendances de données permet de détecter les instructions du programmes qui ne contribuent pas au résultat final (d'une fonction par ex).

Déplacement des invariants de boucle



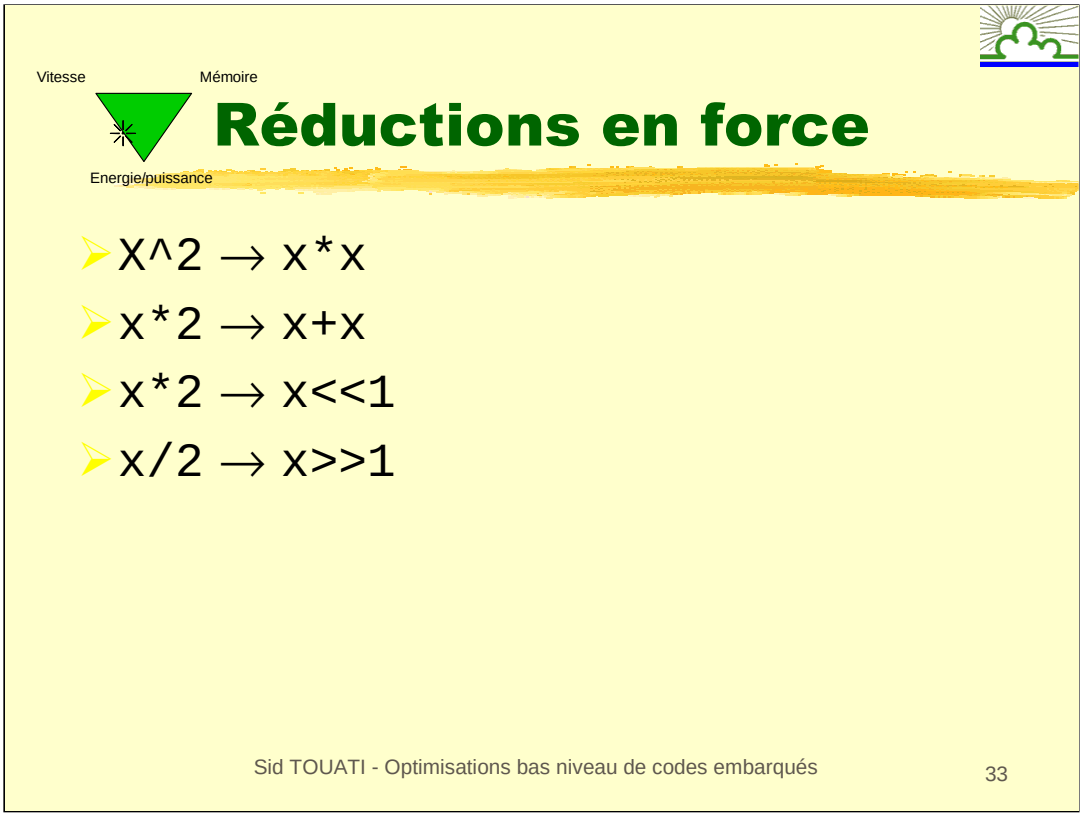
- Si le résultat d'une expression n'est pas modifié à l'intérieur d'une boucle, on peut déplacer l'expression à l'extérieur de cette boucle

```
for (i=0; i<n; i++){
 a[i]=x+y+z;
}
```

```
t=x+y+z
for (i=1; i<n; i++)
 a[i]=t
```

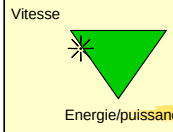
```
for (i=0; i<n; i++){
  if (x<3) a[i]=1;
}
```

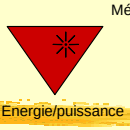
```
if (x<3){
  for (i=1; i<n; i++)
  a[i]=1;
}
```



Si nous étions dans le cas d'un modèle d'exécution ILP, il y a un risque d'impact négatif sur la vitesse de traitement. En effet, la réduction en force augmente la pression sur l'unité de calcul entier…

Spécialisation : déroulage de boucle





- L'*overhead* de boucle se réduit, taille de code explose.

```
n=5
…
for (i=0; i<n; i++){
 a[i]=1;
}
```

```
a[0]=1;
a[1]=1;
a[2]=1;
a[3]=1;
a[4]=1;
```


La spécialisation et la spéculation sont des techniques d'optimisations globales à manier avec grande prudence à cause de l'expansion de taille de code.

Spécialisation : clonage de fonctions

- L'appel à une fonction est remplacé par le code de la fonction.

```
int main (){
  …
  y=foo(5);
  …
}
```

```
int foo (int y){
  x=y+3;
  if (x-5= 0) …
  return x;
}
```

```
int main (){
  …
  x= 5+3;
  if(x-5=0) …
  y = x;
  …
}
```

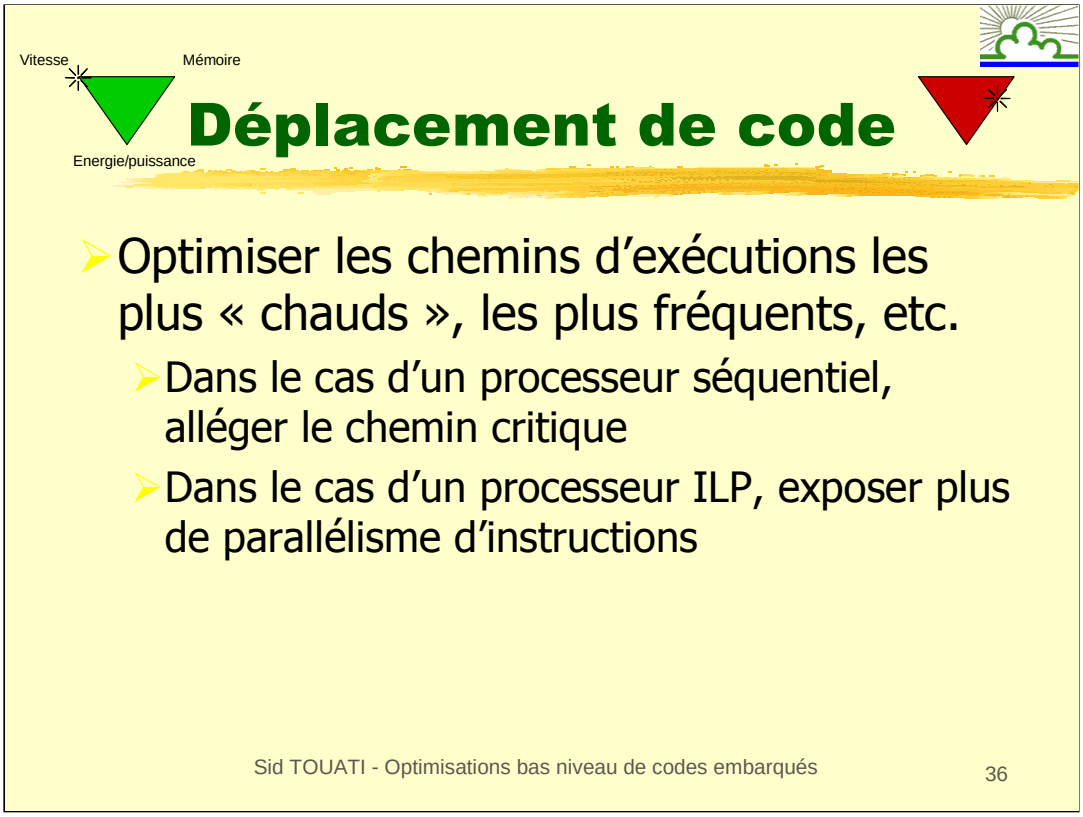
clonage

```
int main (){
  …
  x= 8;
  if(8 = 0) …
  y = 8;
  …
}
```

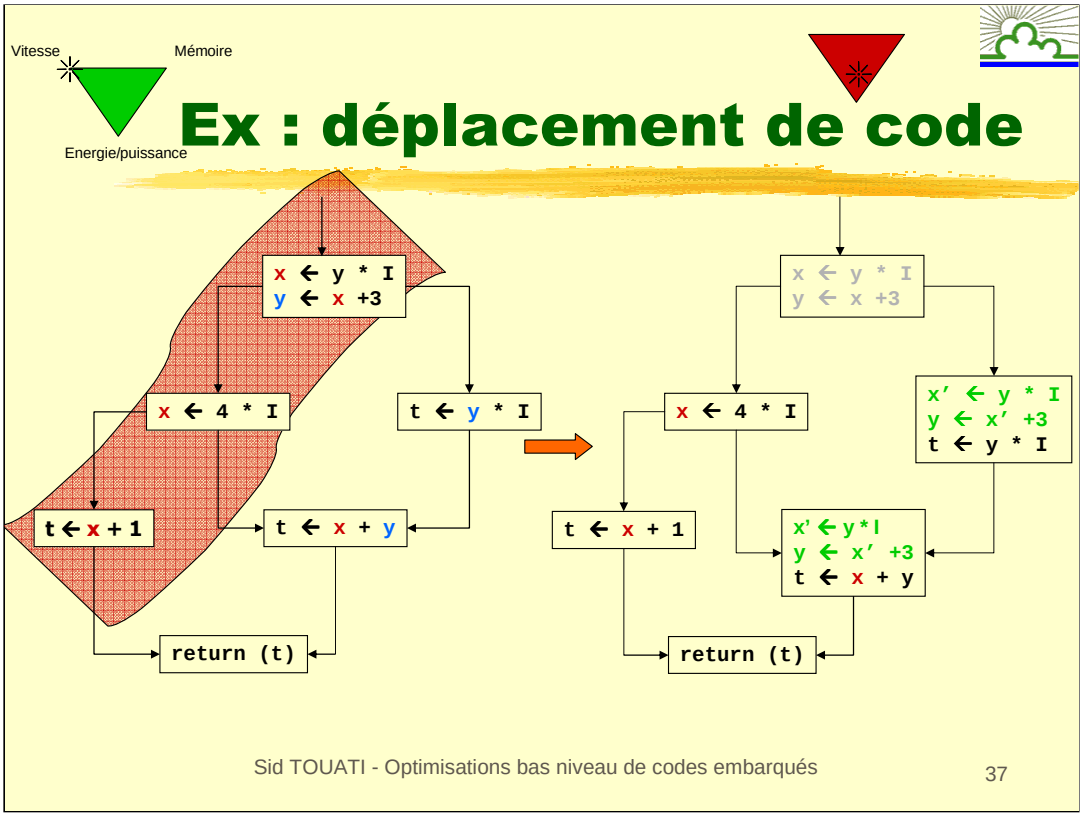

Il faut décider d'un budget pour la taille de code si on veut utiliser la spécialisation de procédures.

Le clonage est aussi appelé versionning.

L'*overhead* des appels de fonctions se réduit, au prix de taille de code.



Engendre du code de compensation, et une possibilité de perte d'énergie pour cause d'exécution de code en plus.



En vert, c'est le code de compensation qui permet de garder un résultat valide si le flot d'exécution n'a pas suivi le chemin spéculé.

Plan

- Enjeux de l'optimisation de codes embarqués
- Optimisations pour la vitesse
  - Modèle d'exécution séquentiel
  - Modèle d'exécution à parallélisme d'instructions (ILP)
- Optimisation pour la mémoire
- Optimisations pour l'énergie et la puissance

Quelle métrique pour la vitesse d'exécution ?

- Depuis l'introduction de l'ILP dans les années 80, il n'existe plus de corrélation entre vitesse d'exécution et nombre d'opérations exécutées.
- Beaucoup d'informaticiens s'accrochent toujours à cette idée fausse!
- Il n'y a pas de métrique/modèle simple évident pour la vitesse d'exécution d'un programme à parallélisme d'instructions.

Contraintes sur l'ILP

- Contraintes de ressources
  - Tables de réservation, pipeline
- Contraintes de registres
- Contraintes de codage d'instructions VLIW
- Autre contraintes:
  - Branchements, Caches, Queues load/store, etc.

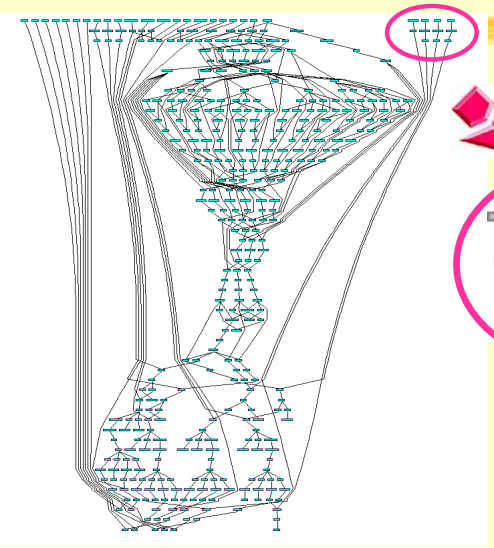
Exploitation du parallélisme d'instructions

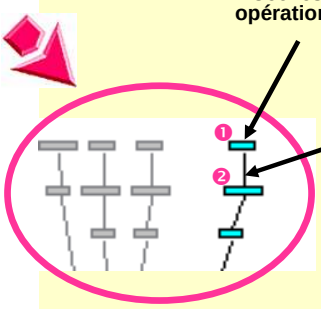
- Ordonnancement d'instructions
  - Science : théorie de l'ordonnancement, complexité, RO, etc.
  - Engineering : techniques *ad-hoc*
- Deux grandes familles d'ordonnancements:
  - Ordonnancements apériodiques, acycliques : blocs de base, CFG sans boucle, etc.
  - Ordonnancements périodiques, cycliques : boucles internes, nids de boucles

Plan

- Enjeux de l'optimisation de codes embarqués
- Optimisations pour la vitesse
  - Modèle d'exécution séquentiel
  - Modèle d'exécution ILP : cas acyclique (apériodique)
- Optimisation pour la mémoire
- Optimisations pour l'énergie et la puissance

Ex: calcul sur un seul pixel d'image



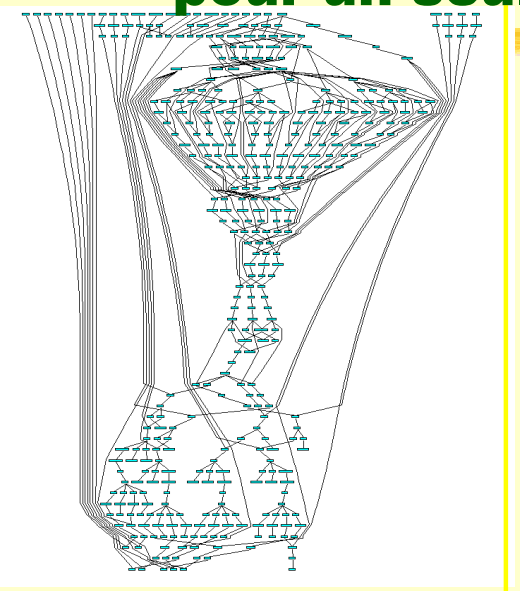


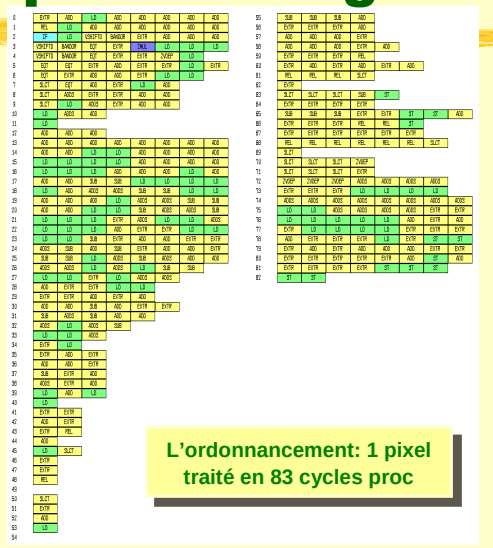
Algorithme: programme typique d'une imprimante couleur HP (zoom, conversion de couleurs, etc.)

Source: Josh Fisher

Ce slide est emprunté de Joseph Fisher de HP-Labs

Ex: ordonnancement d'instructions pour un seul pixel d'image



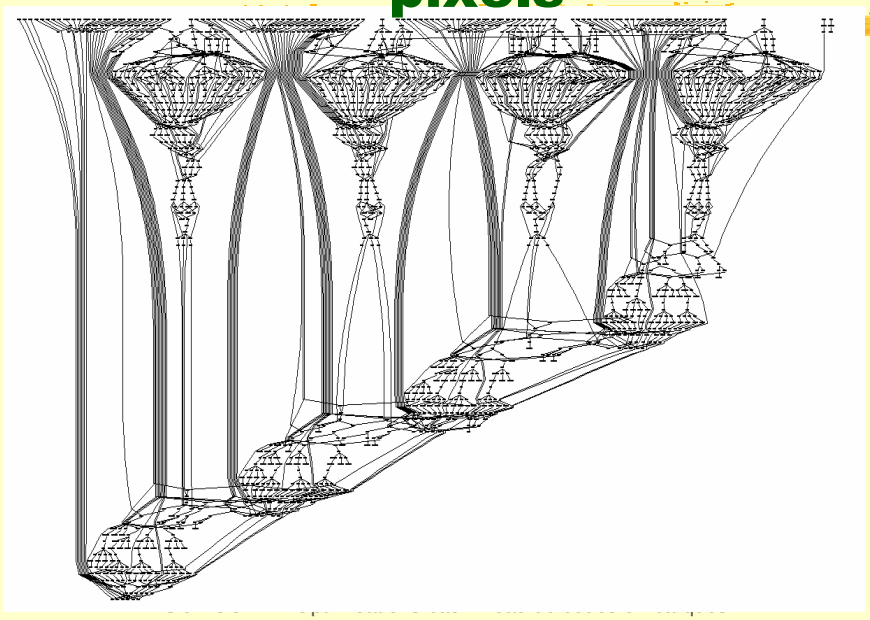


L'ordonnancement: 1 pixel traité en 83 cycles proc

Source: Josh Fisher

Ce slide est emprunté de Joseph Fisher de HP-Labs

Ex: Dérouler le traitement sur 4 pixels

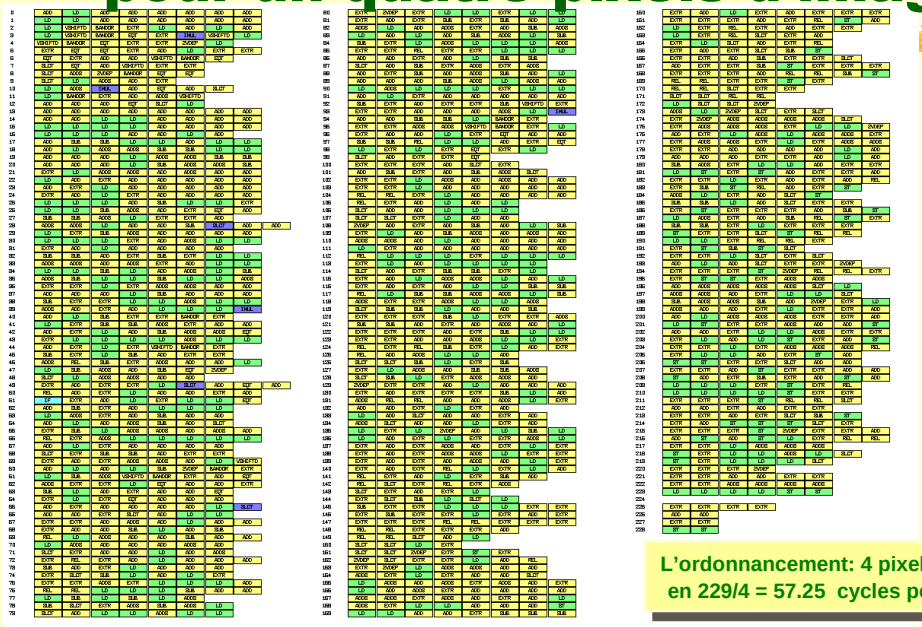


45

Source: Josh Fisher

Ce slide est emprunté de Joseph Fisher de HP-Labs

Ex: ordonnancement d'instructions pour un quatre pixels d'image



L'ordonnancement: 4 pixels traités en 229/4 = 57.25 cycles per pixel.

46

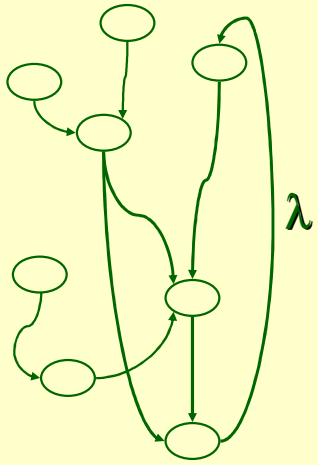
Source: Josh Fisher

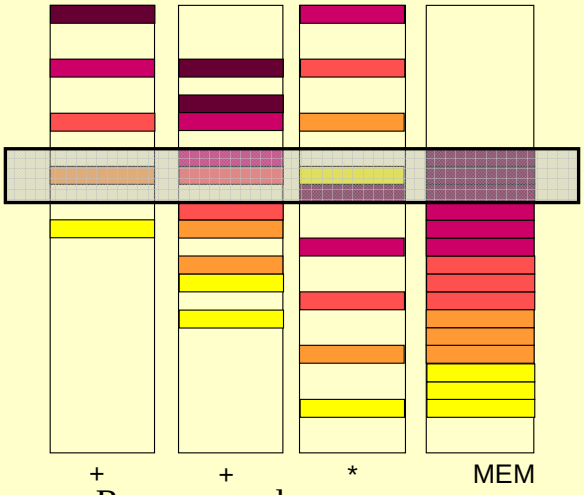
Ce slide est emprunté de Joseph Fisher de HP-Labs

Plan

- Enjeux de l'optimisation de codes embarqués
- Optimisations pour la vitesse
  - Modèle d'exécution séquentiel
  - Modèle d'exécution ILP : cas cyclique (périodique)
- Optimisation pour la mémoire
- Optimisations pour l'énergie et la puissance

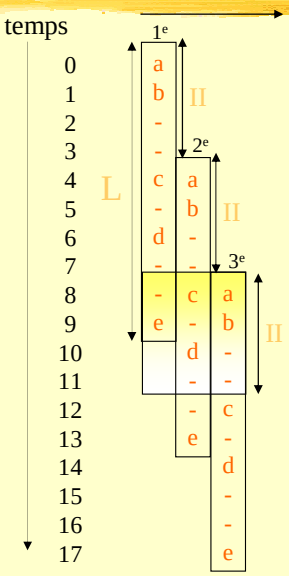
Pipeline Logiciel





Source: Christine Eisenbeis

Pipeline Logiciel



Déroulage de boucle vs. Pipeline logiciel

- Soit *n* le nombre d'itérations à exécuter
- Pipeline logiciel
  - Soit *L* la durée d'exécution d'une itération d'origine, et *II* la durée d'exécution de la nouvelle boucle
  - $T_{pipeline} = L+(n-1)\times II$
- Déroulage de boucle
  - Soit *L'* la durée d'exécution moyenne d'une itération d'origine
  - $T_{deroul}=L' \times n$

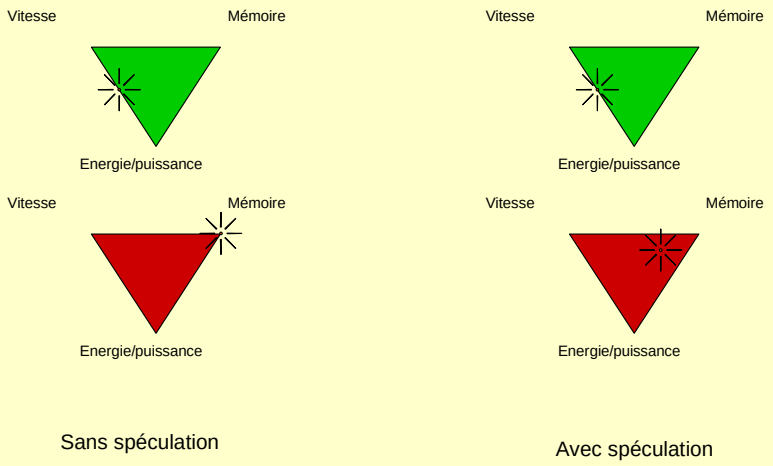
$T_{pipeline}$ = Temps d'exécution de la boucle pipelinée

T_{deroul} = Temps d'exécution de la boucle déroulée

Déroulage de boucle vs. Pipeline logiciel

- Donc :
  - *Si n ≥ (II-L)/(II-L')* alors le pipeline logiciel est plus rentable en terme de temps d'exécution
  - En général, pour les grandes valeurs de *n,* il vaut mieux pipeliner la boucle
- Pour la taille de code, cela reste une question ouverte !

Effet de l'extraction de l'ILP



Sans spéculation

Avec spéculation

Très efficace pour l'accélération de traitement

Néfaste en général sur la taille de code

Si pas de spéculation, extraire et exploiter l'ILP est bénéfique pour la consommation d'énergie

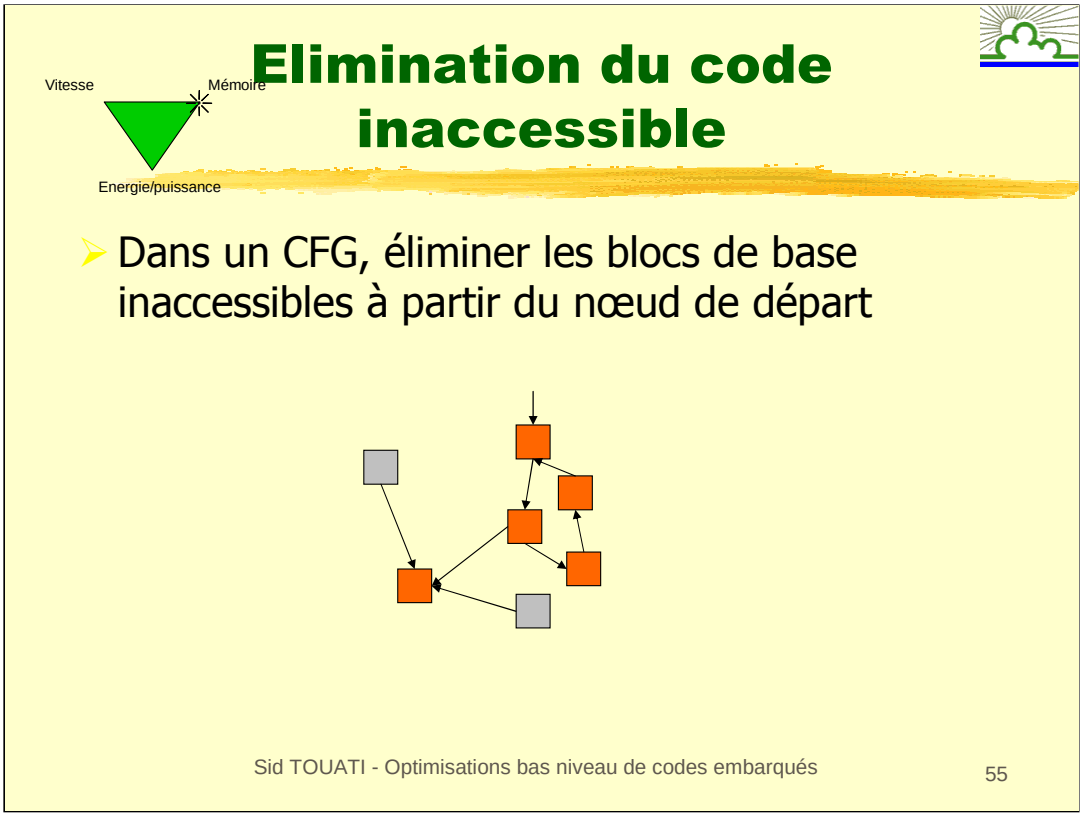
Si la spéculation est utilisé, la consommation d'énergie peut augmenter pour cause d'exécution de code inutile

Plan

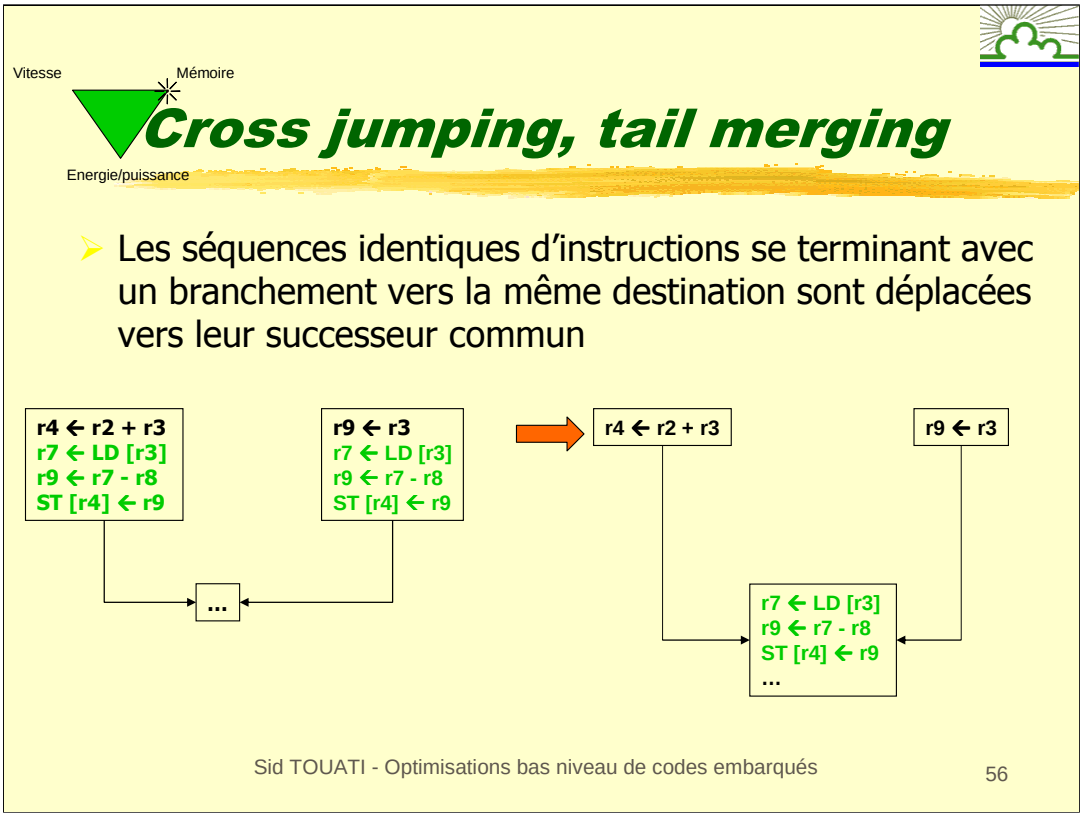
- Enjeux de l'optimisation de codes embarqués
- Optimisations pour la vitesse
- Optimisation pour la mémoire
  - Optimisation de taille de code via compilation
  - Optimisation de taille de code via ISA
  - Optimisation du placement mémoire
- Optimisations pour l'énergie et la puissance

Techniques de compilation pour la taille de code

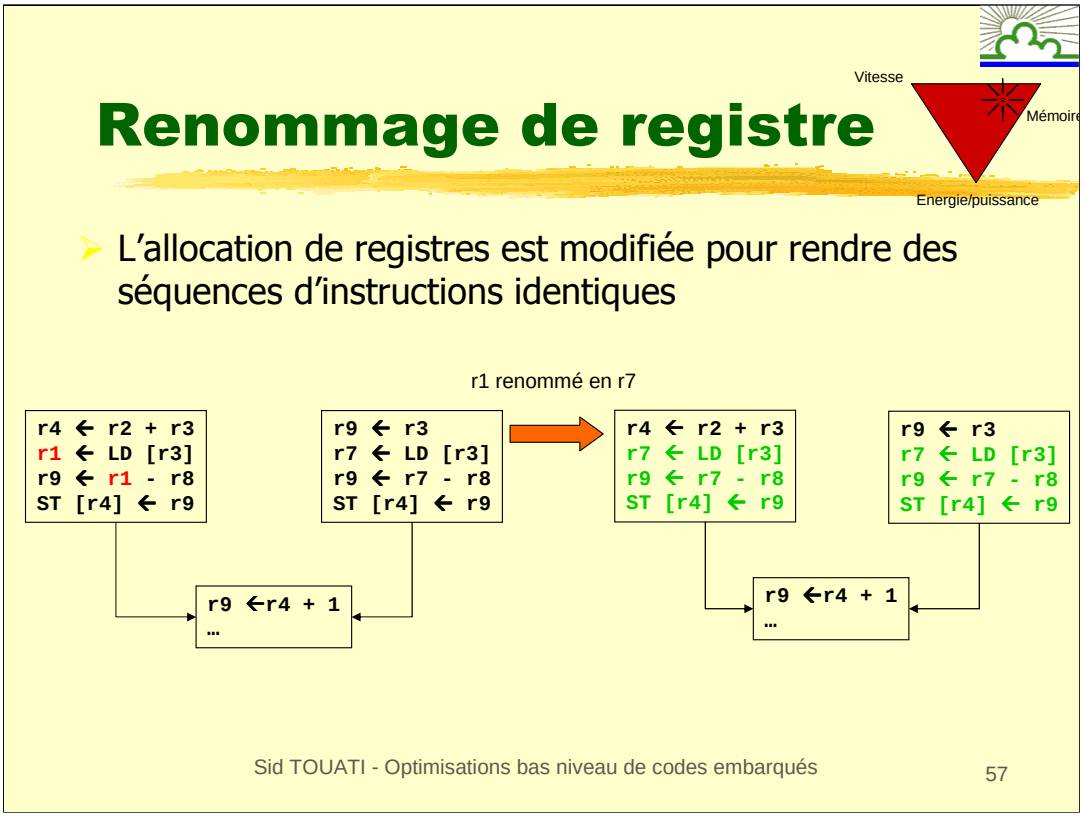
- Autoriser ou pas des optimisations standard
  - Autoriser: élimination de code mort, de sous expressions communes. Load-store redondants, etc.
  - Interdire: Déroulage de boucle, spécialisation de fonctions.
- Utilisation des branchements
  - Fusion de queue (*tail merging*), *cross-jumping*
  - Abstraction procédurale
- Compaction du code exécutable
  - Techniques de dictionnaires : code interprété
  - ISA à bi-largeur (*dual-width ISA*) : Thumb/ARM, MIPS 16/32
  - Compacter des fragments « froids » de code
  - Compromis entre gain d'espace and temps d'exécution



Une propagation de constante peut révéler du code mort.



La technique du *cross_jumping* n'a aucun coût en terme de temps d'exécution. Par contre, elle est peu efficace, dans le sens où les séquences identiques lexicalement sont rares dans les vrais codes : par ex, il suffit que deux registres soient différents dans deux séquences pour qu'elles ne soient pas identiques.

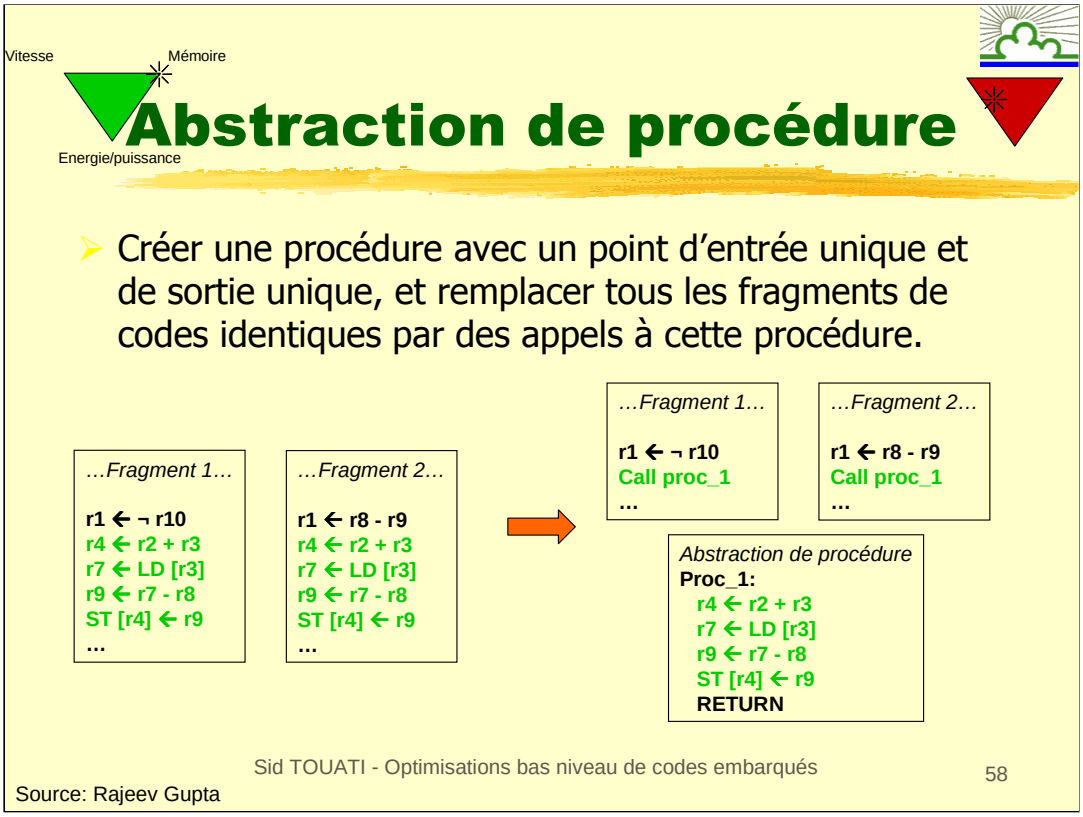


Le renommage de registres en général sert indirectement à plusieurs objectifs:

-Extraction de l'ILP : renommage statique par le compilateur, renommage dynamique par le processeur

-Aide au cross jumoing, à la compression de code, etc.

Il peut y avoir un coût faible en terme de taille de code, causé par l'insertion d'opérations de transferts (move) pour corriger le code.



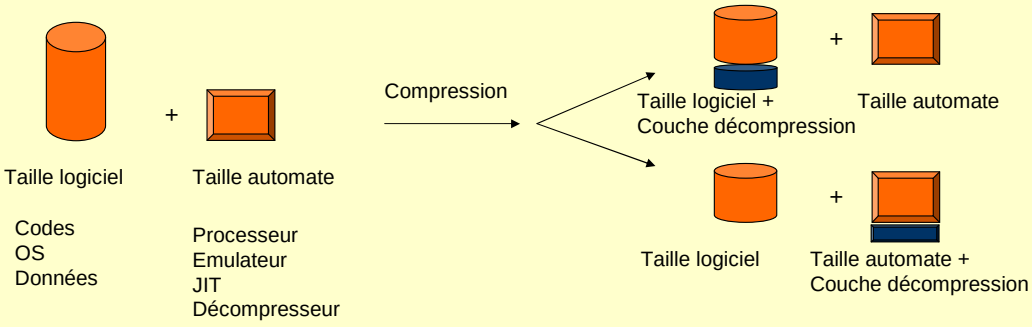
Coût

Taille de code : les séquences d'instructions remplacées par des instructions d'appels, et les procédures abstraites doivent avoir une instruction de retour.

Temps d'exécution: appels de fonctions (avec éventuellement des passages de paramètres, sauvegarde de contexte, etc.)

Compression : que dit la science ?

- La théorie la plus générale derrière la compression/décompression est basée sur les travaux de Kolmogorov



Les travaux de Kolmogorov n'apportent pas de réponses mathématiques complètes à ce concept de « complexité de kolmogorov ». Cela reste ouvert.

Relation taille programme et performance

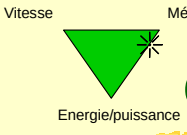
- Aucune réponse générale claire, question ouverte scientifiquement
  - Un petit programme n'est pas nécessairement plus lent
  - Ni plus rapide par ailleurs …
- Techniques et réponses au cas par cas uniquement

Compression de code

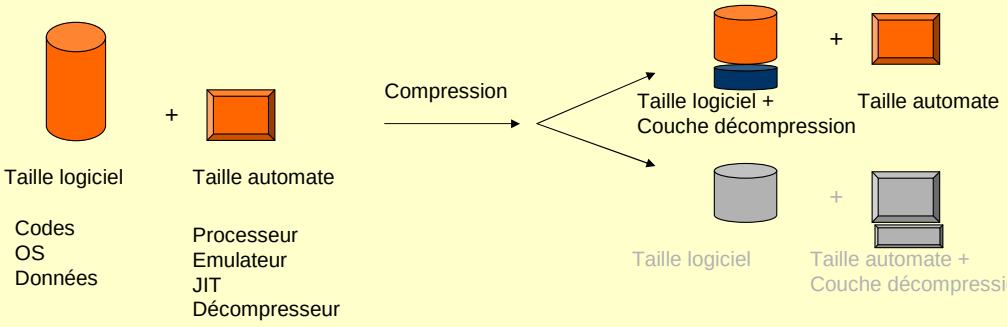
- Les techniques de compression de données ne sont guère efficaces pour la compression de code (entropie faible).
- La compression de code doit répondre à plusieurs impératifs
  - Branchements à des codes non encore décompressés
  - Efficacité de la compression et décompression
  - Taille de l'espace nécessaire à la décompression

•Cout en terme de vitesse de traitement : temps de décompression et/ou émulation. Mais il y a des effets de bord : en effet, la taille réduite du code améliore parfois les effets de cache, ce qui permet parfois des accélérations d'exécution. Mais en général, compresser du code réduit la vitesse de traitement.

•Les algorithmes de compression de données travaillent généralement à une granularité fixe : typiquement des octets, bien qu'il y ait des variantes. Or, les instructions sont constitués de plusieurs champs à granularité non fixe (octets, bits, etc.). Ainsi, le taux de compression d'un algorithme de compression de données appliqué sur du code n'est pas très satisfaisant. De plus, les techniques de compression de données décompressent le code entièrement, en un seul coup. L'espace nécessaire à la décompression est donc maximal.



Compression logicielle

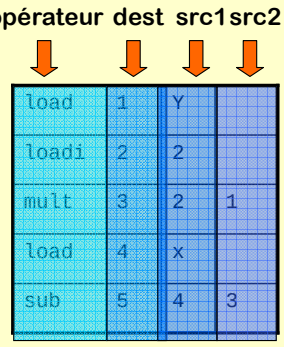


Compression de code et décomposition en flux

- Les instructions sont composées de plusieurs champs : opcode, numéros de registres, données immédiates, etc.
- Considérer chacun des champs comme un flux séparé, et compresser chaque flux indépendamment des autres avec une technique de compression de données.

```
r1 ← load y
r2 ← 2
r3 ← r2 × r1
r4 ←  load x
r5 ← r4 - r3
```

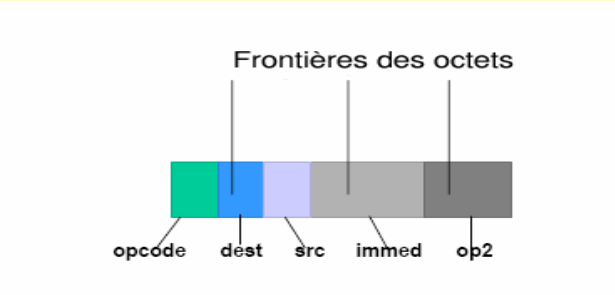
assembleur RISC



Instructions et flux à compresser

Problème de compression de flux de codes

- Les champs des instructions ne s'arrêtent pas forcément aux frontières des octets
  - Les techniques de compression par flux n'exploitent pas pleinement les similarités entre les champs d'instructions.
  - Une autre concept : compression par dictionnaire



Compression de code avec dictionnaires

- Mettre les séquences d'instructions les plus fréquentes dans un dictionnaire
- Le code est une séquence de références aux éléments du dictionnaires
- Une variante : compresser le dictionnaire également !

(100)	load	y	
(101)	loadI	2	
(102)	mult	(100)	(101)
(103)	load	x	
(104)	sub	(103)	(102)

Dictionnaire des instructions

100	14	166	79	100	45	101	345

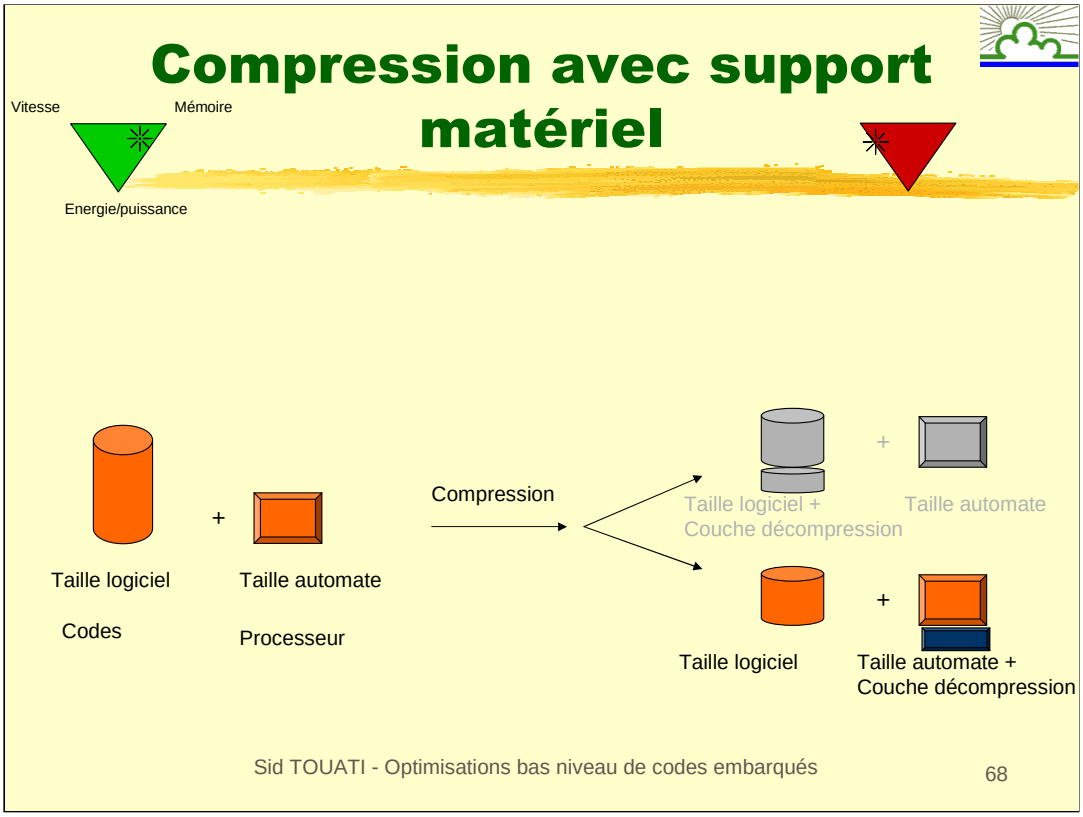
Code : numéros des instructions

Compression de code avec profilage d'application

- Afin d'accélérer l'exécution d'un programme compressé, ne compresser que les fragments de codes « froids »
  - Faire un profilage
- Utiliser un *buffer* pour la décompression
  - Garder une fonction à la fois.
  - Si plusieurs fonctions, techniques de remplacements : FIFO, permanent + FIFO, etc.

Plan

- Enjeux de l'optimisation de codes embarqués
- Optimisations pour la vitesse
- Optimisation pour la mémoire
  - Optimisation de taille de code via compilation
  - Optimisation de taille de code via ISA
  - Optimisation du placement mémoire
- Optimisations pour l'énergie et la puissance



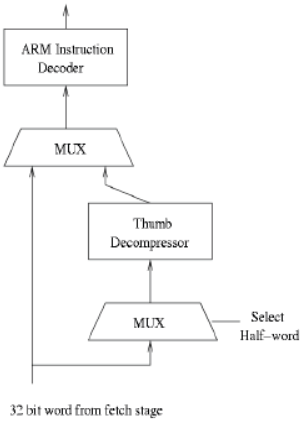
Le coût de décompression matériel en terme de vitesse d'exécution est rattrapée par les gains effectués sur les effets de cache et la bande passante mémoire

Exemple d'ARM

- Processeur à modèle d'exécution séquentielle
- Architecture RISC
- Deux ISA: ARM 32 vs. Thumb16 bits
  - Accès à 16 vs. 8 registres.
  - Code à trois adresses vs. Code à deux adresses.
  - Exécution prédicatée en mode ARM.
  - Moins d'instructions en mode Thumb
  - Etc.
- Le mode Thumb d'ARM améliore la densité du code:
  - Taille de code réduite en largeur
  - Effets de I-Caches améliorés
  - Amélioration de l'utilisation de la bande passante mémoire (bus)

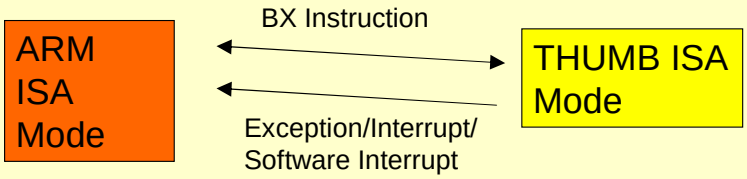
Exécution de code ARM/Thumb

- Le cœur du processeur exécute des instructions ARM
- Le décompresseur matériel convertit les instructions Thumb vers leur équivalent en instructions ARM
- La décompression matérielle s'effectue à l'étage du décodage du pipeline



Source: Rajeev Gupta

Switch entre les deux modes ARM/THUMB



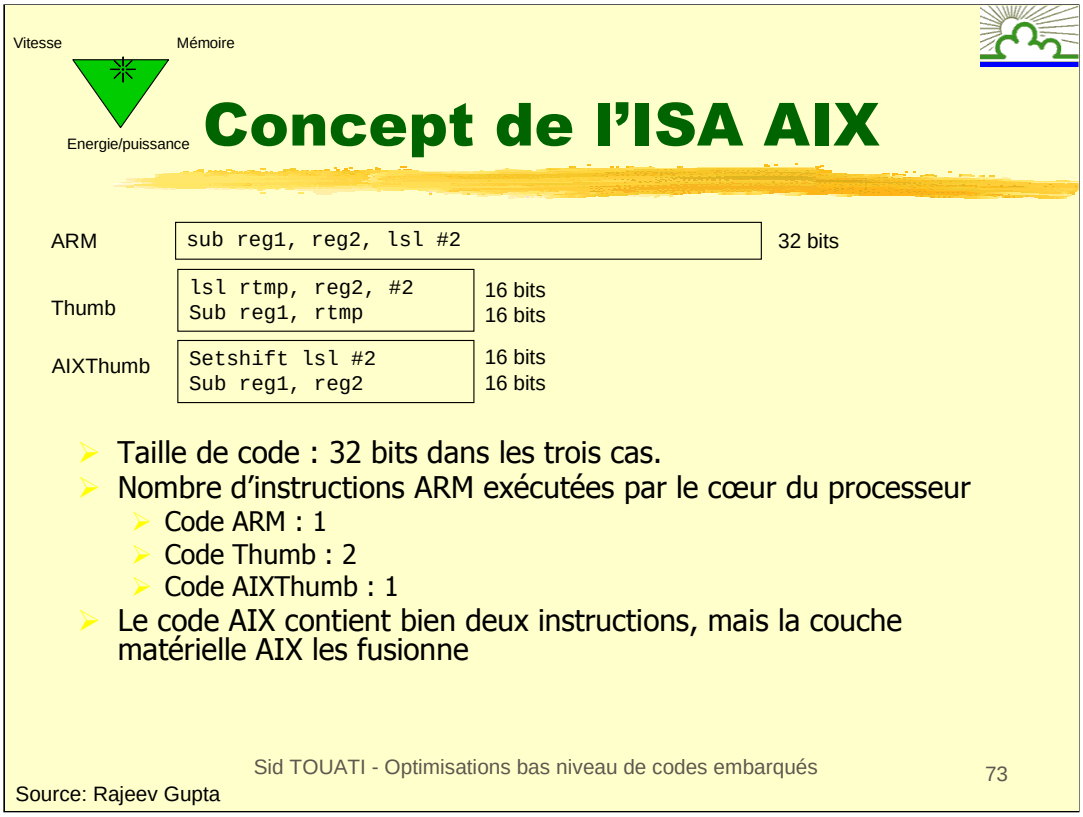
- Quand faut il opérer un changement ARM/Thumb ?
  - Pour les fonctions « froides »
  - Si budget de taille de code est serré
  - Plein de techniques et heuristiques…

Il y a plusieurs méthodes pour passer du mode ARM à un mode thumb et vice versa. La méthode la plus usuelle est d'utiliser l'instruction de branchement spécial BX (Branch and eXcange).

Etant donné que les instructions ARM et Thumb sont alignés sur des frontières de 16 ou 32 bits, le bit 0 du PC n'est jamais utilisé par les instructions de branchements standards. Ainsi, l'instruction BX se branchera à l'adresse spécifiée dans le registre argument *Rm* en passant en mode Thumb si le bit Rm[0] est égal à 1, en mode ARM sinon.

Réduire l'écart entre ARM et Thumb

- Observations
  - L'ISA Thumb n'a pas toutes les possibilités de l'ISA ARM standard
  - De petites séquences d'instructions ARM deviennent des séquences plus longues si elles sont traduites en instructions Thumb.
- Solution *ad-hoc*:
  - Continuer à nourrir le processeur avec des instructions 16 bits
  - Ajouter une couche matérielle qui fait des optimisations à lucarne (*peephole optimizer*)
    - Convertit de façon optimisée des séquences Thumb en séquences ARM équivalentes dans le processeur.
    - Ajouter de nouvelles instructions à l'ISA de thumb : *Augmenting Instruction set eXtensions (AIX)*
  - Les paires d'instructions AIX et Thumb sont fusionnées par le processeur pour former des instructions ARM.



Avec AIXthumb, la taille de code est généralement entre la taille de code ARM et la taille de code Thumb. La vitesse de traitement aussi.

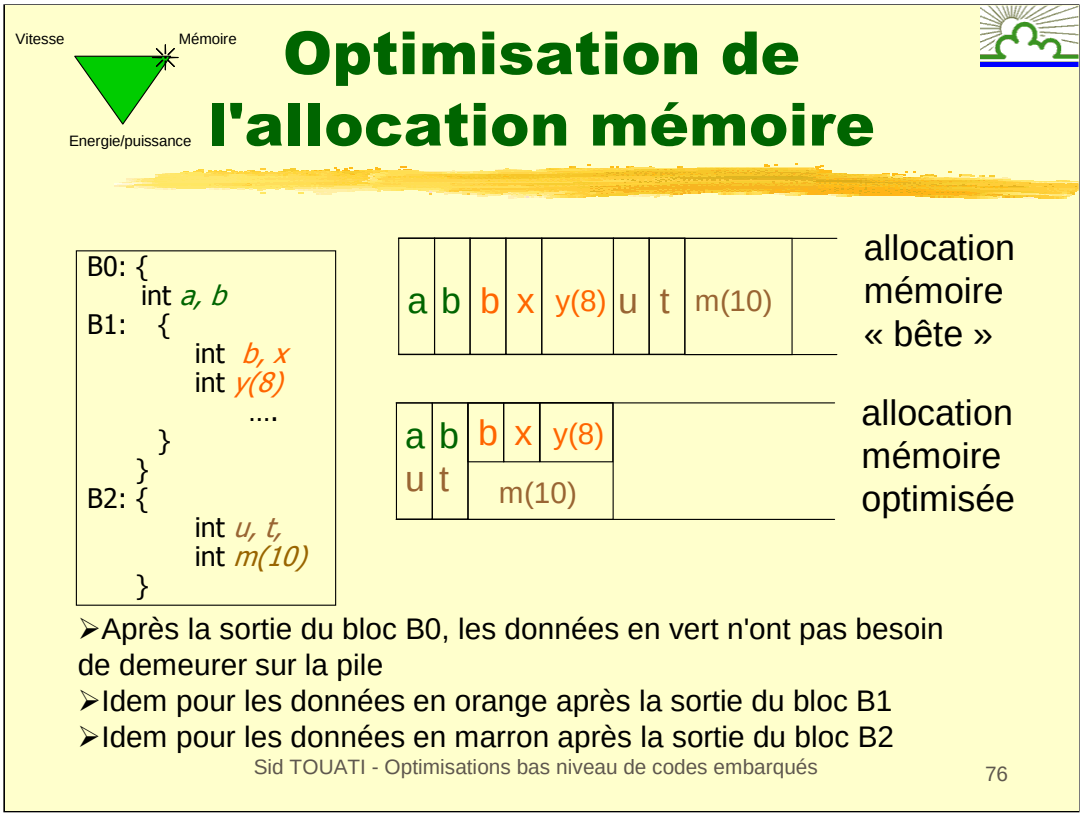
En général, l'insertion d'instruction AIXthumb arrive en post-compilation : par ex, en optimisant le code objet, qui est un code binaire.

Plan

- Enjeux de l'optimisation de codes embarqués
- Optimisations pour la vitesse
- Optimisation pour la mémoire
  - Optimisation de taille de code via compilation
  - Optimisation de taille de code via ISA
  - Optimisation du placement mémoire
    - Exemple d'optimisation de l'espace de données
    - Exemple de technique placement de données sur DSP
    - Exemple de technique de placement de code
- Optimisations pour l'énergie et la puissance

Optimisation de l'allocation mémoire

- Certaines variables ou données locales ne sont jamais en vie simultanément : elles peuvent partager le même espace.
- C'est le cas quand elles sont déclarés dans deux blocs non imbriqués.
- Le compilateur peut donc leur allouer la même zone mémoire.



L'optimisation de l'allocation mémoire pour les tableaux multidimensionnels est une tache plus ardue. Bien qu'elle engendre un gain certain en espace de donnée, elle peut avoir un impact négatif sur les temps d'exécution à cause des calculs d'adresse compliqués (modulo) supplémentaires.

Plan

- Enjeux de l'optimisation de codes embarqués
- Optimisations pour la vitesse
- Optimisation pour la mémoire
  - Optimisation de taille de code via compilation
  - Optimisation de taille de code via ISA
  - Optimisation du placement mémoire
    - Exemple d'optimisation de l'espace de données (scratch-pad)
    - Exemple de technique placement de données sur DSP
    - Exemple de technique de placement de code
- Optimisations pour l'énergie et la puissance

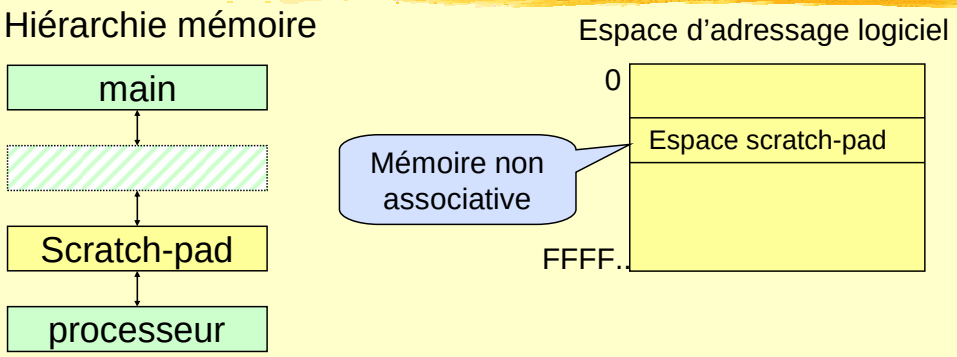
Caches et systèmes embarqués

- La « prévisibilité » du temps d'exécution d'un programme embarqué est une caractéristique importante
- L'utilisation des caches avec des politiques de remplacement à effets difficilement prévisibles
- En présence de cache, une analyse WCET devient très complexe

Dans le cas où l'on souhaite avoir des garanties temps réel en présence de caches, les analyses WCET sont souvent très pessimistes, entraînant des sous-exploitations des caches.

D'autres arguments sont contre l'utilisation des caches classiques dans les systèmes embarqués : consommation d'énergie (SRAM) et la taille du sillicium du cache.

Mémoire scratch-pad : caches gérés par logiciel :



Source: Rajeev Gupta

Mémoire scratch-pad

- C'est une mémoire flash à faible consommation d'énergie, à taille physique plus compacte
- C'est un cache géré par logiciel : le compilateur place des données dans une plage d'adressage spécifique
  - Latence d'accès connu par le compilateur : WCET devient plus précis, les techniques d'extraction d'ILP deviennent plus fiables, etc.

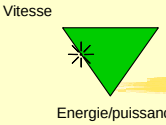
Défis à la compilation

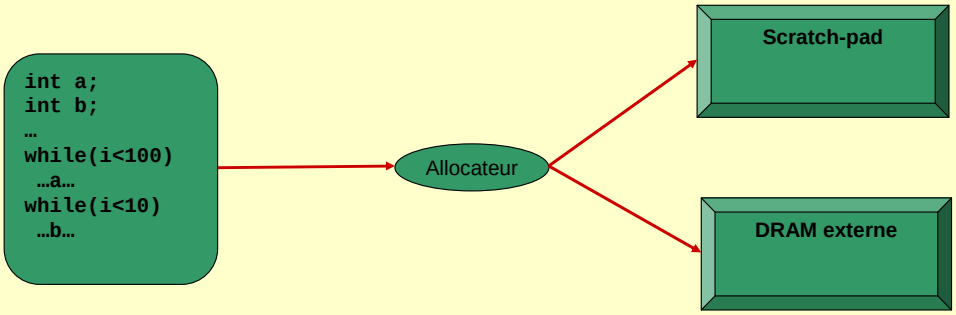
- Comment ferait le compilateur pour placer les données des programmes dans la mémoire scratch-pad?
- But : Placer les données les plus « fréquemment » utilisées dans la mémoire scratch-pad
  - Réduire le temps d'exécution
  - Réduire la consommation d'énergie

Stratégies pour Scratch-pad

- Allocation statique
  - Le placement de données est fixée à la compilation et ne change pas durant l'exécution du programme.
  - Données statiques globales, données sur piles
- Allocation dynamique
  - Le compilateur insère du code pour déplacer les données à l'exécution du scratch-pad à la mémoire, et vice-versa.
  - Données statiques globales, données sur piles, données sur le tas

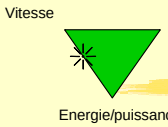
Scratch-pad: allocation statique

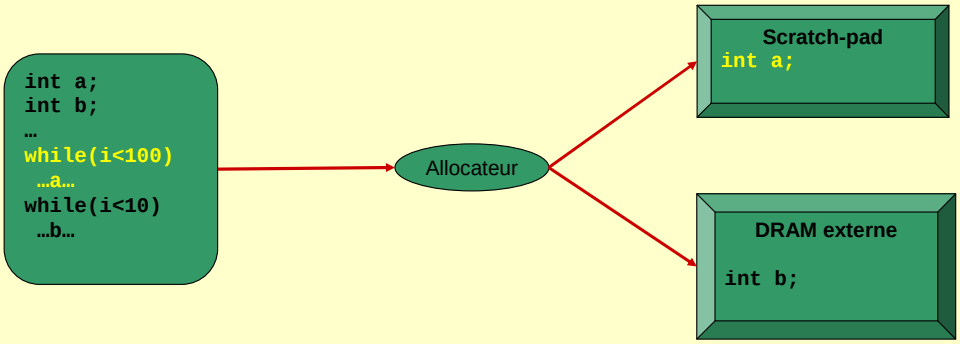




Source: Rajeev Gupta

Scratch-pad: allocation statique





Source: Rajeev Gupta

Cette technique d'allocation statique n'est probablement pas optimale, mais apporte des gains positifs en terme de vitesse de traitement et de consommation d'énergie.

A priori, je ne vois pas d'impact négatif, si ce n'est le simple fait de ne pas être une méthode de placement « optimale ».

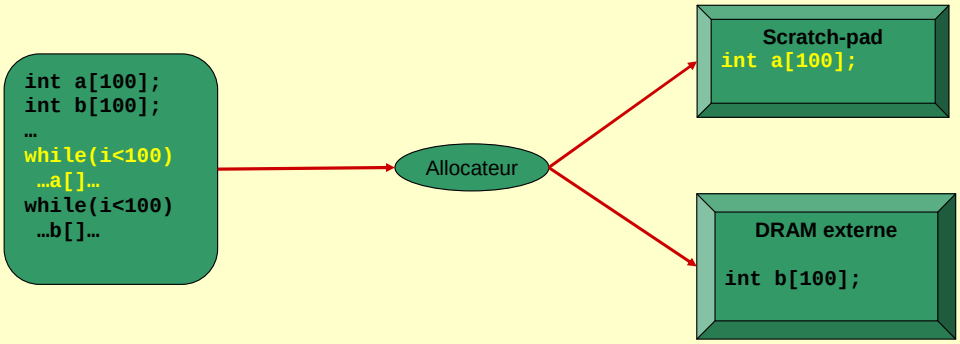
•Formalisation par PLNE possible

•Estimation de la fréquence d'accès via profilage

•Maximiser le gain obtenu en allouant des données sur scratch-pad

•Contrainte : la taille globale des données à allouer est bornée par la taille du scratch-pad

Scratch-pad: problème de l'allocation statique

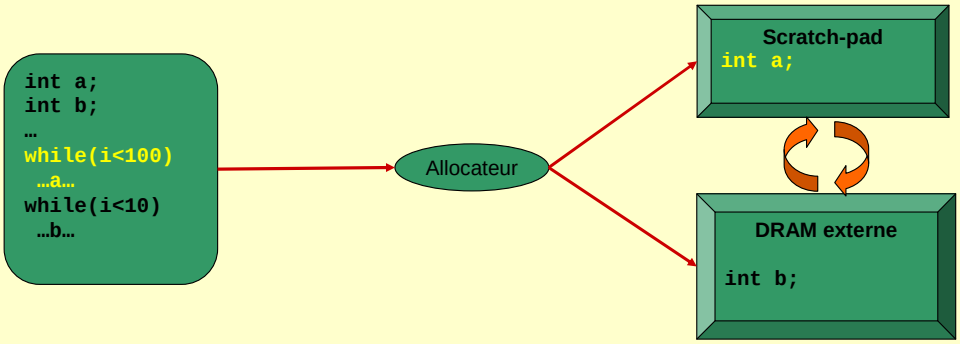


Est-il possible de faire mieux avec une allocation dynamique ?

Source: Rajeev Gupta

Le fait que l'allocation statique reste inchangée durant l'exécution, des opportunités d'optimisation mémoire restent non exploitées.

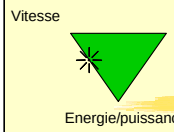
Scratch-pad: allocation dynamique

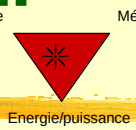


Source: Rajeev Gupta

L'allocation dynamique fait en sorte que des données puisse transiter de la DRAM vers la scratch-pad (et vice-versa) durant l'exécution du programme.

Scratch-pad: allocation dynamique






```
int a[100];
int b[100];
…
// a est dans la scratch-pad
while(i<100)
  …a…
// copier a vers la DRAM
// copier b vers la scratch-pad
while(i<100)
  …b…
…
```

Coût de transfert

Le compilateur décide de ce comportement dynamique statiquement en insérant du code supplémentaire

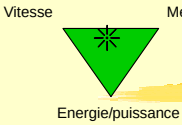
Source: Rajeev Gupta

Cette technique d'allocation dynamique s'appuie sur une estimation du gain et des coût de transfert. Si l'estimation n'est pas précise (car liée au profilage), il peut y avoir des pertes de temps d'exécution et d'énergie, et un peu d'augmentation de taille de code (opérations de transferts).

Plan

- Enjeux de l'optimisation de codes embarqués
- Optimisations pour la vitesse
- Optimisation pour la mémoire
  - Optimisation de taille de code via compilation
  - Optimisation de taille de code via ISA
  - Optimisation du placement mémoire
    - Exemple d'optimisation de l'espace de données
    - Exemple de technique placement de données sur DSP
    - Exemple de technique de placement de code
- Optimisations pour l'énergie et la puissance

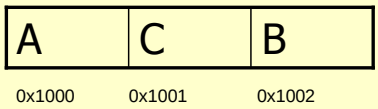
Placement mémoire de variables sur DSP



- Les processeurs DSP ont des modes d'adressage spécialement conçus pour le traitement des flux de données
- Mécanisme d'accès mémoire avec auto-incrément/décrément du registre d'adressage
  - X ← LD [AR++] : coût nul d'auto-incrémentation du registre adresse

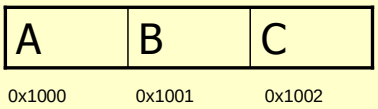
Exemple

Faire ‘A’ + ‘B’, résultat dans un accumulateur




```
$AR0  = &A
$ACC  = *$AR0
$AR0  = $AR0 + 2
$ACC += *$AR0
```

Calcul explicit d’adresse




```
$AR0  = &A
$ACC  = *$AR0++
$ACC += *$AR0
```

Auto-Incrément

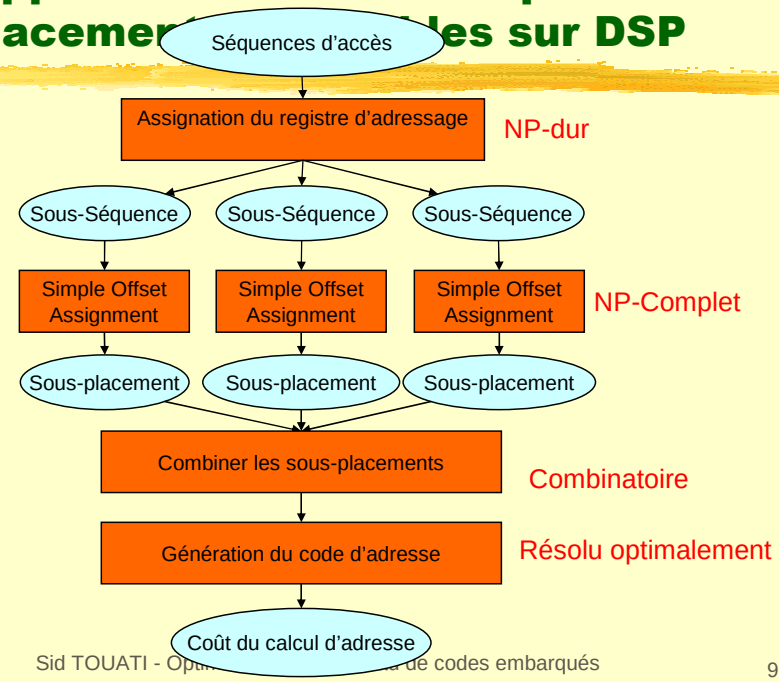
Source: Johnny Huynh

Problème de placement mémoire sur DSP

- Ayant **k** registres d'adressage et une séquence d'instructions accédant **n** variables, trouver un placement mémoire pour les **n** variables qui minimise le coût du calcul d'adresse.
  - Dans quel ordre doit-on placer les variables en mémoire?
  - Quel registre doit adresser quelle variable?

Source: Johnny Huynh

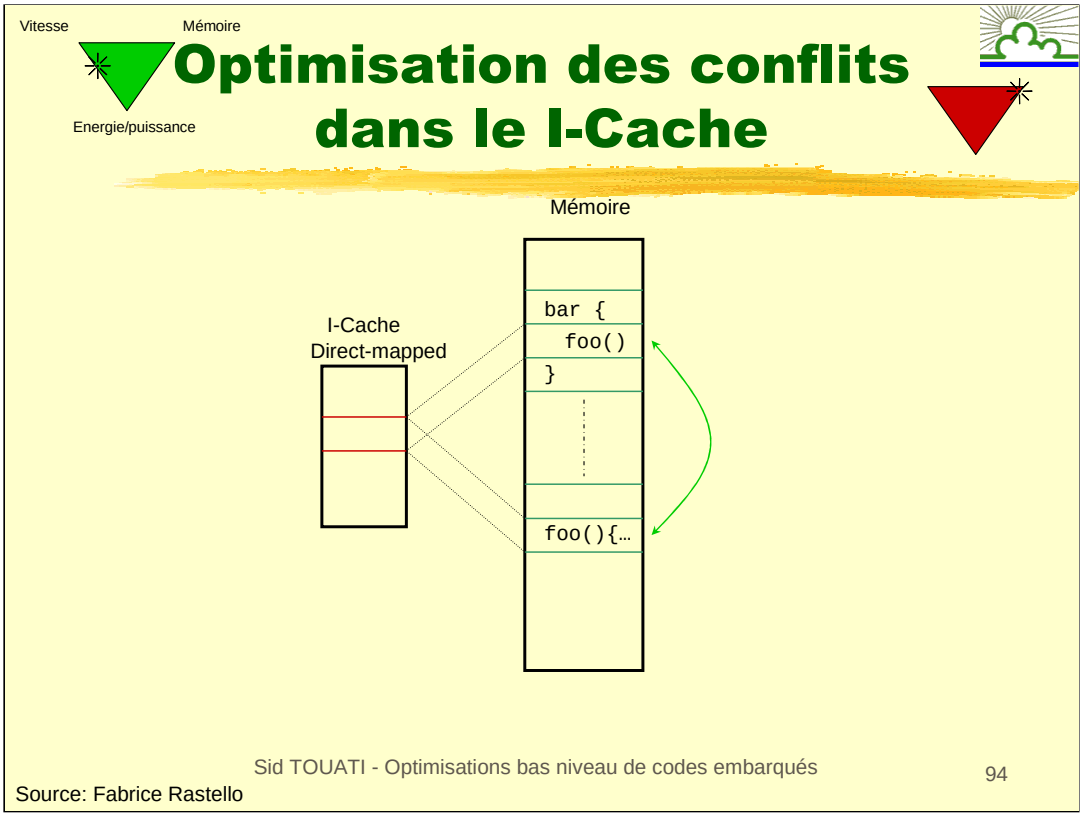
Approche traditionnelle pour le placemer les sur DSP



Source: Johnny Huynh

Plan

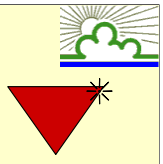
- Enjeux de l'optimisation de codes embarqués
- Optimisations pour la vitesse
- Optimisation pour la mémoire
  - Optimisation de taille de code via compilation
  - Optimisation de taille de code via ISA
  - Optimisation du placement mémoire
    - Exemple d'optimisation de l'espace de données
    - Exemple de technique placement de données sur DSP
    - Exemple de technique de placement de code
- Optimisations pour l'énergie et la puissance

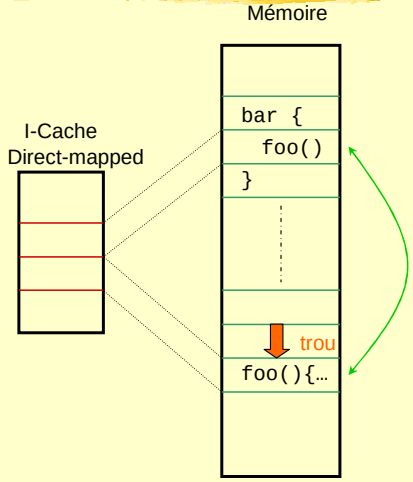


Les conflits en Icache entre les fonctions ne peuvent être déduites qu'en connaissant les adresses des instructions. Cette information n'est pas disponible pendant la compilation, mais après l'édition de lien.

C'est pour cela que cette technique de placement de code est effectuée non pas durant la compilation, mais à l'édition de lien.

Optimisation des conflits dans le I-Cache



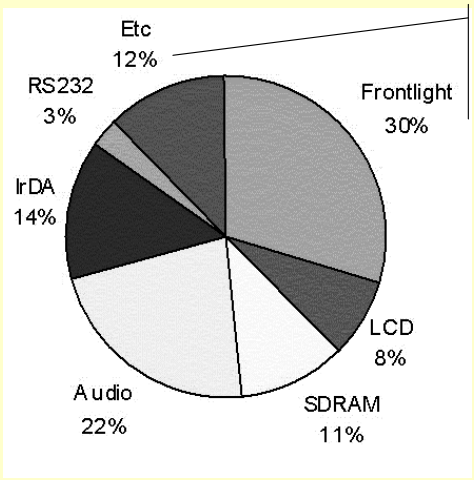


Source: Fabrice Rastello

Plan

- Enjeux de l'optimisation de codes embarqués
- Optimisations pour la vitesse
- Optimisation pour la mémoire
- Optimisations pour l'énergie et la puissance

Exemple de consommation d'énergie



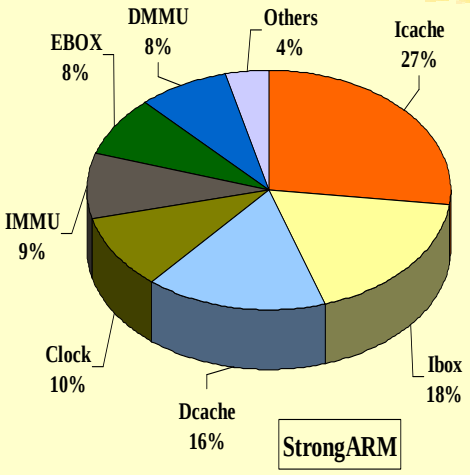
La consommation du CPU est dans la partie « etc. »

Profilage de la consommation d'énergie dans un PDA iPAQ

Source: Josh Fisher

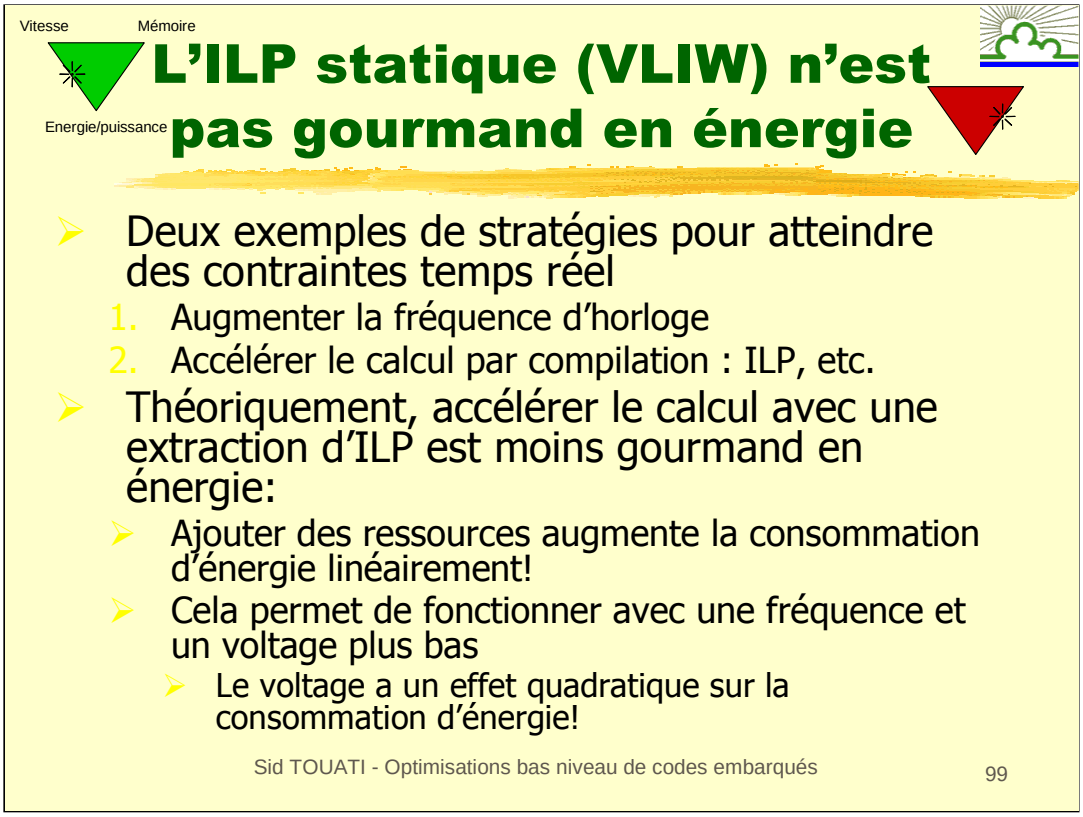
Le processeur n'est pas le composant le plus gourmand en énergie dans un système embarqué typique.

Consommation d'énergie dans un processeur



- Unité entière (EBOX): 8%
- Décode et issue (IBOX): 18%
- Mémoires: Plus que 55%
- Une implémentation VLIW du même ISA donnerait:
  - Augmentation de EBOX
    - Plus d'unités fonctionnelles
  - Augmenter CLOCK et IBOX
    - Plus de registres et ports
  - Augmentation marginale de ICACHE
    - Lignes de cache plus larges
- Exemple simple (4-way VLIW)
  - EBOX : 4x
  - CLOCK et IBOX : 2x
  - ICACHE : 30%
  - Consommation totale d'énergie supplémentaire: ~60%

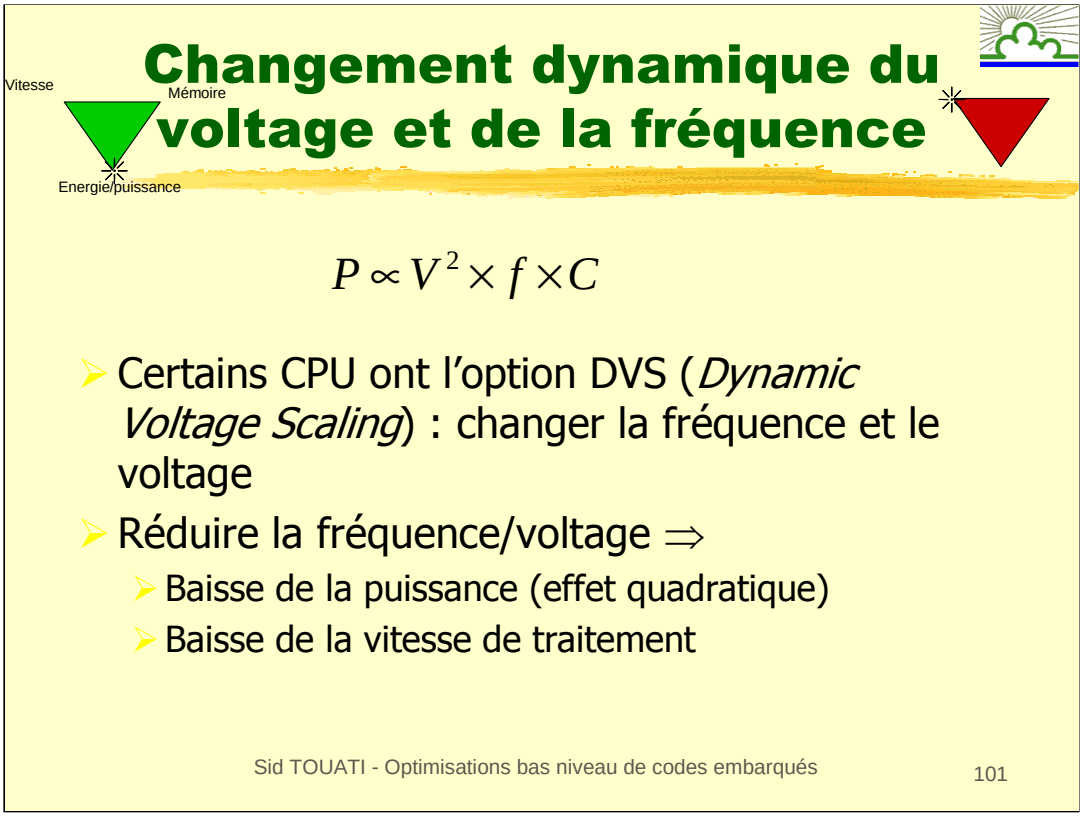
Source: Josh Fisher, from IEEE JSSC, Nov. 96



A vrai dire, si le processeur contient des mécanismes de spéculation, l'extraction d'ILP peut devenir gourmande en énergie.

La compilation et l'économie d'énergie

- En pratique, les techniques de restructuration de code spécialement conçues pour l'économie d'énergie n'ont pas encore donné de résultats spectaculaires
  - Souvent, l'optimisation d'énergie est un effet de bord des techniques d'optimisation de temps : réduire les défauts de caches, etc.
- Les compilateurs manquent de modèles précis et utilisables pour orienter l'optimisation d'énergie



E : énergies consommée par le processeur

V : voltage

f : fréquence de l'horloge du CPU

C : terme qui dépends de la capacitance des composants électronique x activité dynamique des transistors (switching activity)

DVS : quelle granularité ?

- Système entier : intervention manuelle de l'utilisateur
- Système d'exploitation (par service système) : programmes, processus, etc.
- Intra-programme (par compilation) : fonctions, régions, nids de boucles

Un modèle analytique : mémoire et CPU asynchrones

- Hypothèses
  - La mémoire est asynchrone avec le CPU.
  - Le calcul peut être effectué à n'importe quelle fréquence CPU et à n'importe quelle granularité.
  - Chaque opération peut s'exécuter à sa propre fréquence.
  - Il n'y a pas de pénalité (en terme d'énergie ou de latence) associée à un changement de paire *(V,f)*
- Combien de valeurs de tensions possibles ?
  - Cas de voltage continu
  - Cas de voltage discret

Dans un tel modèle analytique, on suppose également que le temps d'exécution d'une opération CPU dépend linéairement de la fréquence d'horloge. Ceci n'est pas toujours le cas, mais on peut l'admettre dans un modèle d'exécution séquentiel simple.

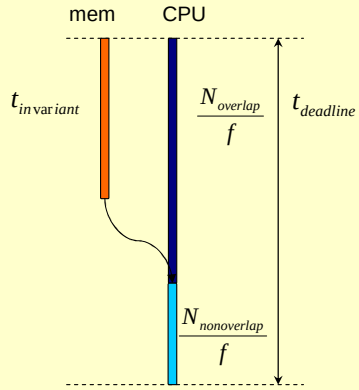
Cas de voltage continu : le voltage et la fréquence peuvent avoir n'importe quelle valeur dans un intervalle

Cas de voltage discret : l'ensemble des valeurs possibles est donné (nombre fini de valeurs possibles).

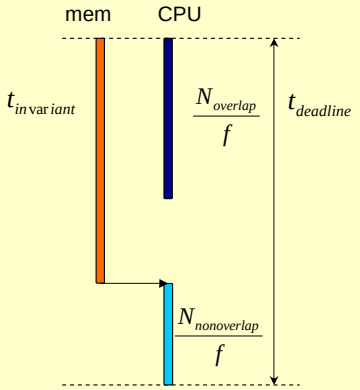
Calculs et accès mémoire chevauchés

- Les accès mémoires et les défauts de caches offrent des opportunités pour appliquer le DVS en intra-programme.
- Baisser la fréquence CPU peut économiser de l'énergie sans impacter les performances globales.

Calculs et accès mémoire chevauchés



(a) Cas *compute-bound*



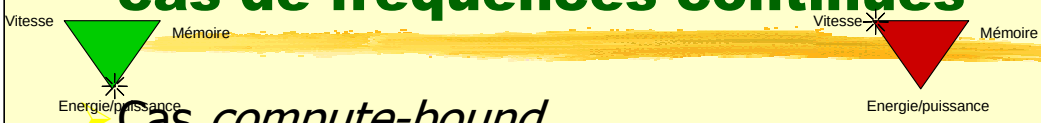
(b) Cas *memory-bound*

Source: Rajeev Gupta

Réduire la fréquence fait augmenter linéairement les temps de traitement de Noverlap et Nnonoverlap.

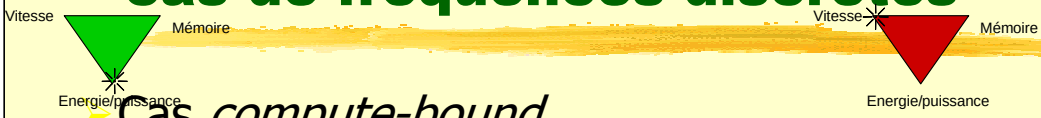
1. $N_{overlap}$: le nombre de cycles CPU d'exécution des opérations qui peuvent se chevaucher avec les accès mémoire.
2. $N_{nonoverlap}$: le nombre de cycles CPU d'exécution des opérations qui ne peuvent pas se chevaucher avec les accès mémoire.
3. $t_{invariant}$: le temps d'accès mémoire.
    - La mémoire est asynchrone vis-à-vis du processeur – ce temps est indépendant de la fréquence CPU

Résultats d'optimalité pour le cas de fréquences continues



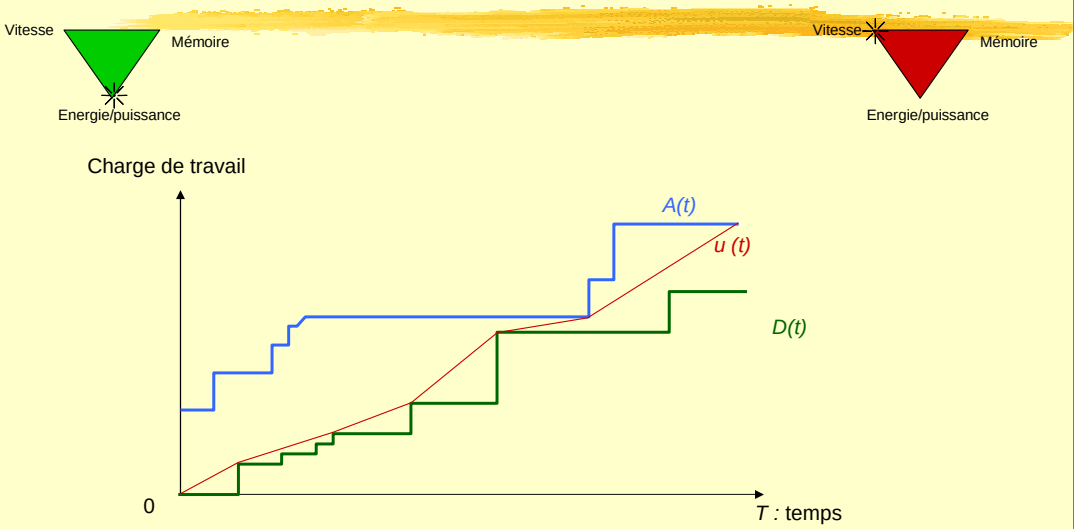
- Cas *compute-bound*
  - Une fréquence unique donne une économie optimale d'énergie.
- Cas *memory-bound*
  - Une économie optimale d'énergie requiert l'utilisation de deux fréquences : l'une pour le calcul chevauché ($N_{overlap}$) et l'autre pour le calcul non chevauché ($N_{nonoverlap}$)

Résultats d'optimalité dans le cas de fréquences discrètes



- Cas *compute-bound*
  - Deux valeurs de voltage qui sont les plus proches valeurs voisines de la solution optimale du cas continu.
- Cas *memory-bound*
  - Quatre voltages : les deux proches valeurs voisines aux deux valeurs optimales du cas continue.

Une autre modélisation analytique : fonctions escaliers



La courbe décrivant les fréquences optimales en terme d'économie d'énergie est cella du plus court chemin de 0 à T (cas de fréquences continues)

Source: Bruno Gaujal

•Pas de distinction entre calculs et accès mémoire

•La charge de travail à accomplir varie selon le temps

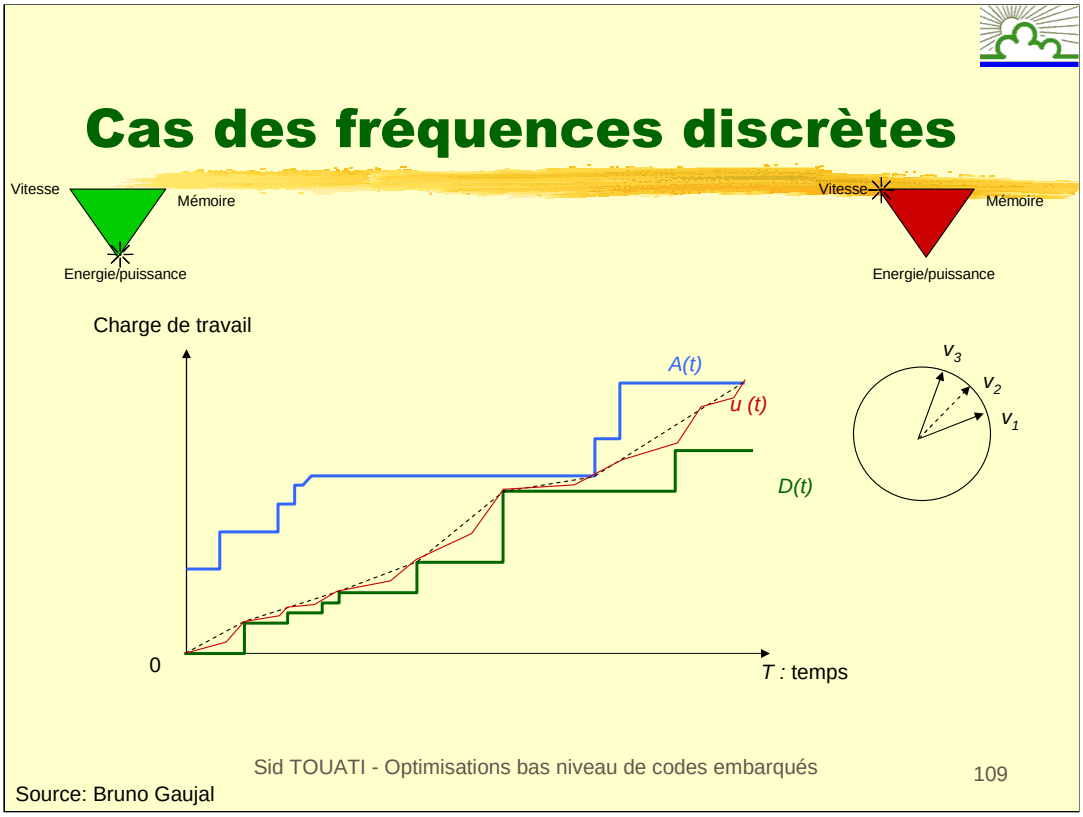
•La charge de travail est composée de plusieurs étapes, chacune a ses contraintes min et max de charge de travail (déduites des contraintes temps réel).

D(t) : Borne inf de la charge de travaille (déduite des dates d'arrivée des jobs)

A(t) : Borne sup de la charge de travaille (déduite des deadline des jobs)

u(t) : la courbe qui décrit la charge de travaille optimale minimisant la consommation d'énnergie (les fréquences optimales sont les différentes pentes de la courbe).

L'énergie consommée $e(t)=g(u(t))$, où g est un modèle analytique de l'énergie en fonction de la fréquence $u(t)$. Le but du problème de minimisation de la consommation d'énergie est de minimiser l'intégrale de e(t). Si g est une fonction convexe, alors la solution optimale pour $u(t)$ est décrite par le plus court chemin de 0 à T.



D(t) : Borne inf de la charge de travaille (déduite des dates d'arrivée des jobs)

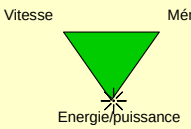
A(t) : Borne sup de la charge de travaille (déduite des deadline des jobs)

u(t) : la courbe qui décrit la charge de travaille optimale minimisant la consommation d'énnergie (les fréquences optimales sont les différentes pentes de la courbe). La courbe en pointillée est la courbe optimale en cas de fréquences continues

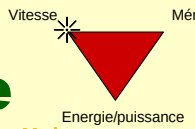
Il existe une variante à ce problème d'optimisation de l'énergie : en effet, on peut considérer le même problème mais en ajoutant la contrainte de minimiser le changement de fréquences (afin de ne pas abîmer la batterie, le ventilateur, etc.). On sait calculer une solution optimale à ce problème.

Considérations pratiques pour le DVS

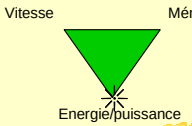
- Vitesse de la transition vers de nouvelles fréquences et tensions:
  - Centaines de microsecondes
  - Milliers d'instructions
- Granularité arbitraire fine
  - La même instruction ne peut pas s'exécuter à différentes fréquences
  - Les régions d'un programmes définissent la granularité : nids de boucles, régions avec points d'entrée/sortie uniques.
- Maximiser la durée de vie de la batterie
  - Minimiser la variance de la décharge de la batterie
- Energie CPU vs. Énergie du système
  - Faire tourner le CPU à sa fréquence maximale peut être le meilleur choix.



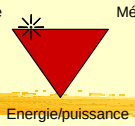
DVS statique



- Détermination statique de la meilleure fréquence/tension
  - L'accélération de traitement qui résulte des techniques usuelles d'optimisation de code peut être reconverti en un budget pour économiser l'énergie
  - Le programme s'exécute à une fréquence/tension unique, calculée statiquement.



DVS dynamique



- Le compilateur insère dans le code des instructions de changement de fréquence/tension. Celles-ci changent durant l'exécution.
- Régions: nids de boucles
- Modèles linéaires :
  - Estimation de l'énergie consommée par le i^e niveau de boucle si une certaine tension/fréquence est utilisée.
  - Estimation du temps d'exécution du i^e niveau de boucle si une certaine tension/fréquence est utilisée.
- Coût de transition entre fréquences = constante

DVS dynamique

- Une formalisation du problème par PLNE est possible
- But
  - Trouver les points dans les nids de boucle où le CPU peut être ralenti avec une perte acceptable de vitesse d'exécution.
  - Ayant un programme *P*, trouver une région *R* et une fréquence *f* tel que si *R* est exécutée à une fréquence *f* et *P-R* est exécutée à une fréquence f_{max}, alors le temps d'exécution est augmenté au plus par *r%* et la consommation d'énergie globale est minimisée.

PLNE : programmation linéaire en nombres entiers

Conclusion

- Pour plus d'informations (références, compléments d'informations, etc.) :
  - Sid.Touati@uvsq.fr
  - Sid.Touati@inria.fr

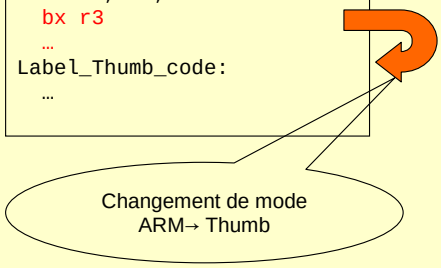
Pot pourri

Mode ARM, mode Thumb

- Autoriser le changement d'état du processeur
- Si l'instruction *BX Rm* est exécutée, alors le processeur se branche à l'adresse spécifiée par `Rm[31…1]`, et fait le changement de mode:
  - Si Rm[0]=1⇒mode Thumb
  - Si Rm[0]=0 ⇒mode ARM

```
Example:

Label_ARM_code:
  …
  mov r3, label_Thumb_code
  or r3, r3, #1
  bx r3
  …
Label_Thumb_code:
  …
```



Changement de mode
ARM→ Thumb

Il y a plusieurs méthodes pour passer du mode ARM à un mode thumb et vice versa. La méthode la plus usuelle est d'utiliser l'instruction de branchement spécial BX (Branch and eXcange).

Etant donné que les instructions ARM et Thumb sont alignés sur des frontières de 16 ou 32 bits, le bit 0 du PC n'est jamais utilisé par les instructions de branchements standards. Ainsi, l'instruction BX se branchera à l'adresse spécifiée dans le registre argument *Rm* en passant en mode Thumb si le bit Rm[0] est égal à 1, en mode ARM sinon.