Compilation et gestion mémoire basse consommation

Olivier Zendra

▶ **To cite this version:**

Olivier Zendra. Compilation et gestion mémoire basse consommation. Ecole thématique "Conception faible consommation de système temps réel" (ECoFac 2006) Nice, France, 2006. inria-00001231

HAL Id: inria-00001231

https://cel.hal.science/inria-00001231v1

Submitted on 11 Apr 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Compilation et gestion mémoire basse consommation

Olivier Zendra
TRIO
INRIA-Lorraine / LORIA
(Nancy)

olivier.zendra@loria.fr

http://www.loria.fr/~zendra

Introduction: compilation

- Traduction code source en exécutable
  - Avec optimisation
- Statique (hors ligne): cc, gcc
- Dynamique (en ligne)
  - «Début» d'exécution: *Just In Time* (JVMs)
    - D'un coup
    - Au fur et à mesure
  - En cours d'exécution, avec recompilations: JVMs optimisantes (à la HotSpot)

Introduction: compilation

- Architecture figée
- Programmes inconnus *a priori*
  - Optimiser pour exécuter au mieux
- Optimisation par le matériel
  - Logique en ligne
  - Circuits dédiés
  - Surcoût à l'exécution (E et T)

Introduction: compilation

- Optimisation logicielle (à la compilation statique)
  - Logique hors ligne
  - Pas de surcoût à l'exécution
  - Ressources disponibles supérieures (T, RAM)
  - Contexte beaucoup plus large possible
  - Comportement exact à l'exécution plus difficile à saisir

Introduction: compilation

- Consommation: neuf en compilation. Historiquement, taille & vitesse.
- Optimisations vitesse et énergie
  - Souvent liées [Lee1997]
  - Pas toujours: mise hors du chemin critique est positive en temps, mais en énergie ?
- Optimisation en énergie != en densité de puissance (points chauds)

Introduction

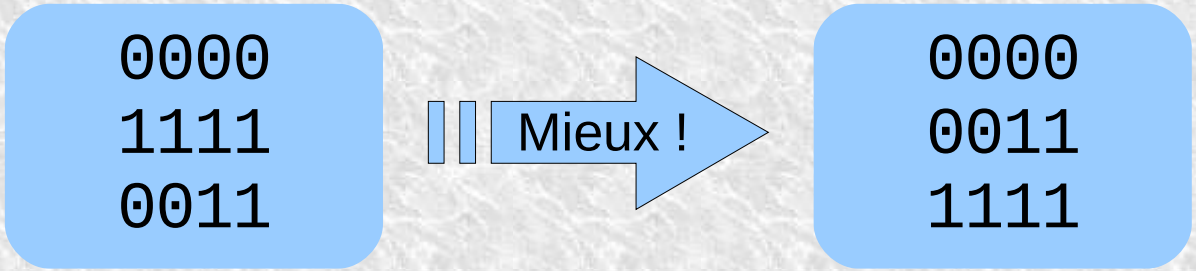
- Survol de techniques et solutions
- Quelques points spécifiques
- Point de vue logiciel (compilateur, application)
- Gestion mémoire

Introduction

- Bas niveau puis haut niveau
- «bas» et «haut» niveau dépendent du point de vue
  - Optimisations «haut niveau» prennent en compte un contexte plus large

1) Transitions et commutations

- Transitions entre instructions successives: coûtent de l'énergie
- Compilateur réordonnance les instructions pour minimiser ce coût [Graybill2002 p193]



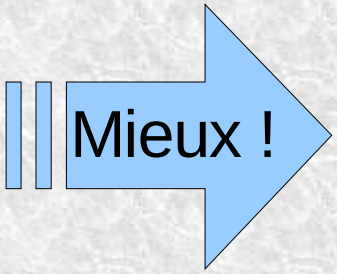
1) Transitions et commutations

- Renommage de registres pour diminuer commutations sur le nom de registre
  - Activité de commutation sur ce champ -11% [Kandemir2000]
- Impact global ?

2) Boucles

- Très nombreux travaux (Catthoor,...)...
- Historiquement pour la vitesse
- Ex.: dépliage de boucles (*loop unrolling*)
  - 1 boucle de longueur *n* exécutée *i* fois devient 1 boucle de longueur *n*x* exécutée *n/x* fois

```
for(i=0;i<10000;i++){
   a();b();
}
```




```
for(i=0;i<5000;i++){
   a();b();
   a();b();
}
```

2) Boucles

- Impact du dépliage de boucles:
  - Instructions statiques dupliquées
    - Taille de code ++
    - Énergie ++
  - Moins d'instructions dynamiques (pour le contrôle)
    - Temps--
    - Énergie --
  - Balancer surcoût et gain !
- D'autres plus loin (mémoire, modes)...

3) Un mot sur les modes

- CPU: DVS/DFS (Dynamic Voltage Scaling / Dynamic Frequency Scaling)
  - $P = C.V^2.f$; $E = P_{moy}.Temps$
  - P dynamique
- Autres: modes repos, hibernation
  - Tendre vers 0 (ressource inutilisée)
  - P dynamique et statique

3) Un mot sur les modes

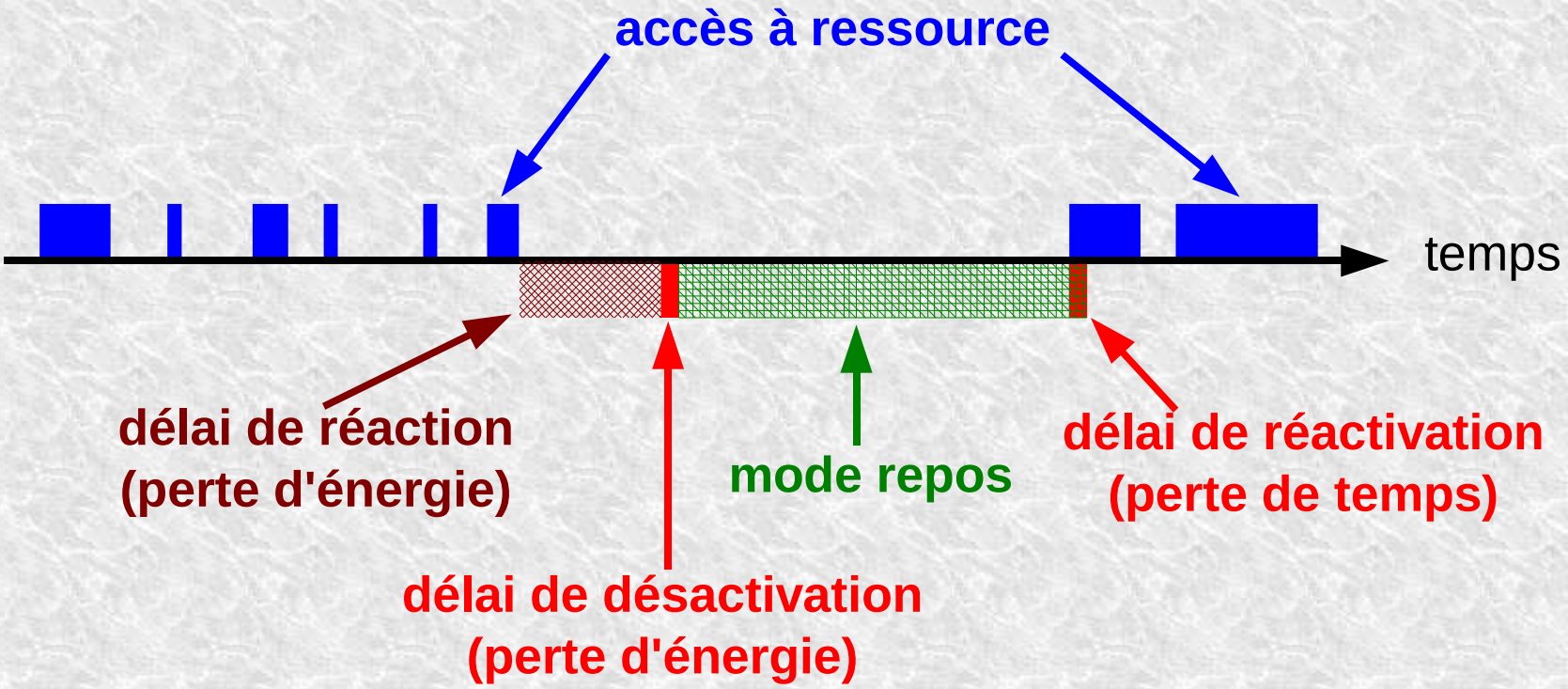
- Très efficaces pour diminuer énergie
- Pour plus (DVS/DFS): voir présentations de V. Rao et C. Belleudy
- Ici: rôle de la compilation et de la gestion mémoire pour tirer parti des modes repos

3) Modes et compilation

- Matériel détecte les phases de faible utilisation d'une ressource
  - Facilement *a posteriori*
  - Prédiction plus difficile, moins sûre
  - Donc retard inutile avant action appropriée

3) Modes et compilation

- Matériel:

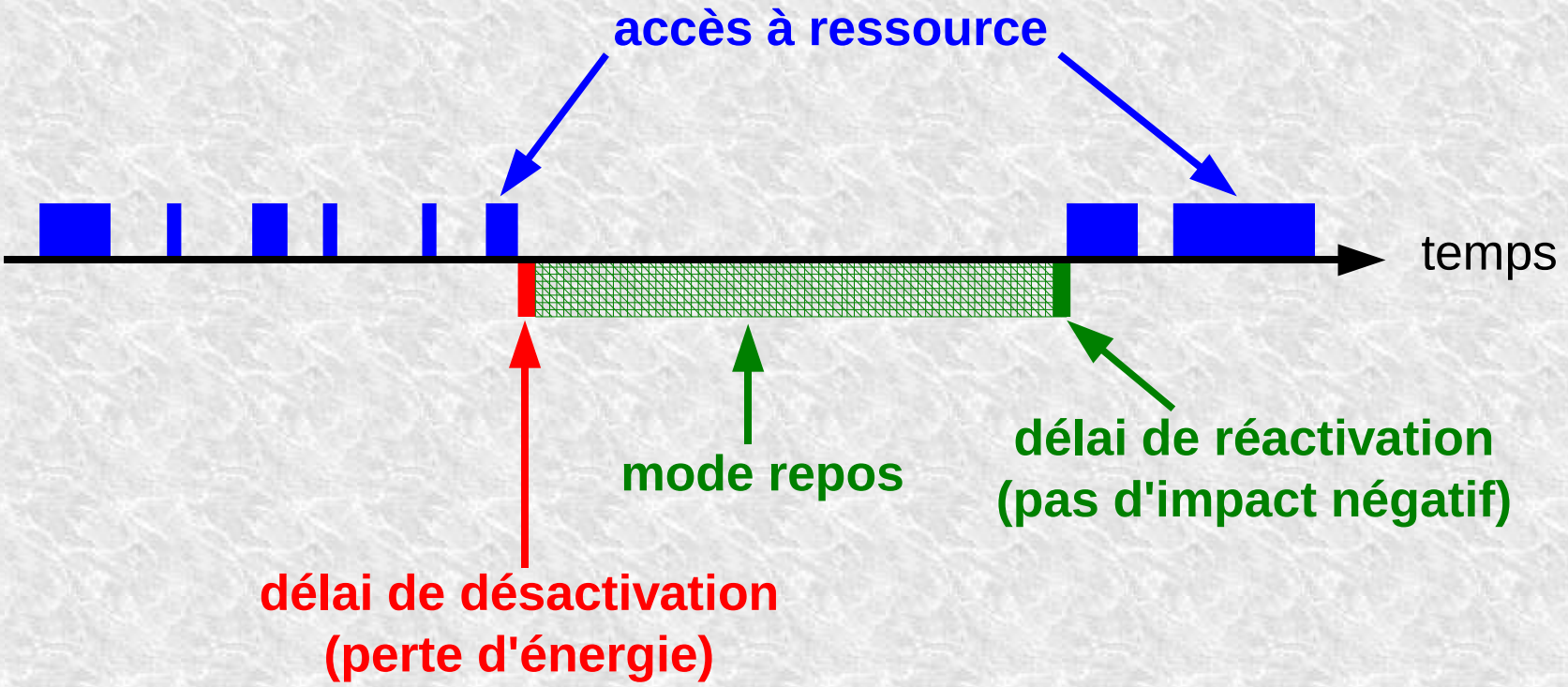


3) Modes et compilation

- Compilateur connaît points où ressource sera inutilisée
  - Libérable immédiatement
  - Peut «prévenir» d'un repos futur
    - Et d'un redémarrage
  - Pas délai inutile en endormissement ou réveil
    - Mieux en E
    - Mieux en T

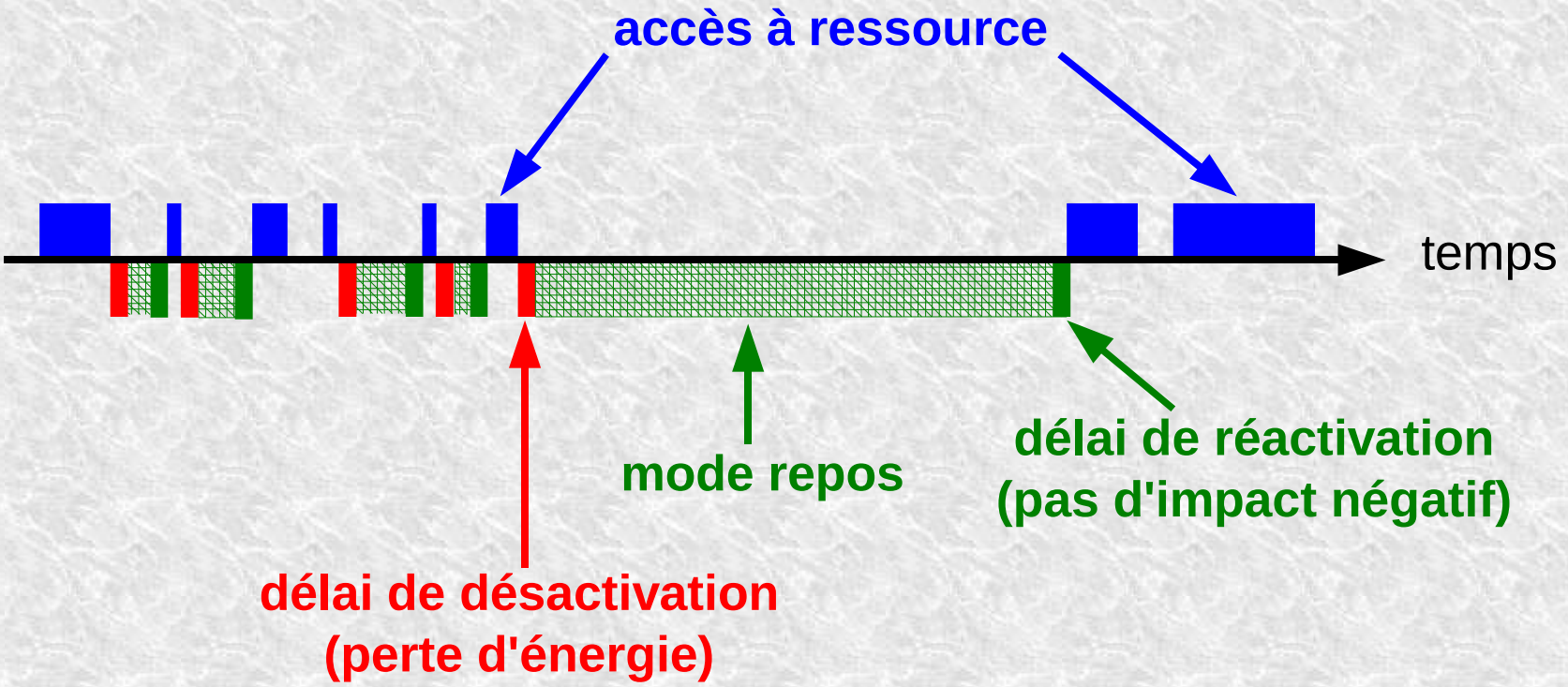
3) Modes et compilation

- Compilateur:



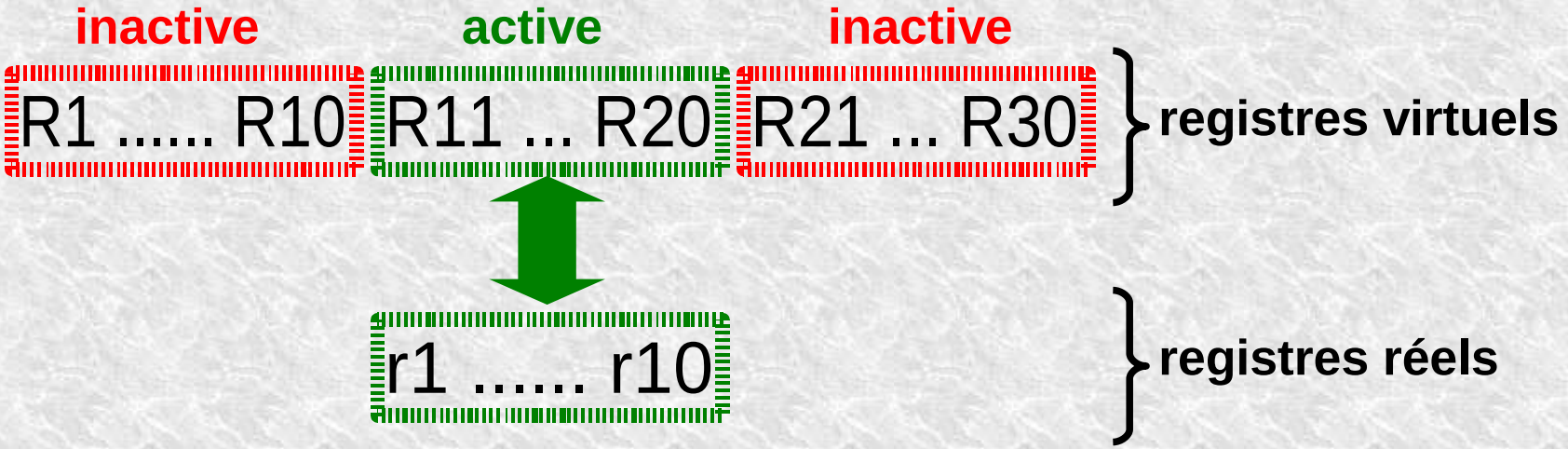
3) Modes et compilation

- Compilateur:



4) Fenêtres de registres: principe

- Plus de registres virtuels que réels
- RV répartis en *n* « fenêtres »
- 1 seule fenêtre RV active à un instant



4) Fenêtres de registres: principe

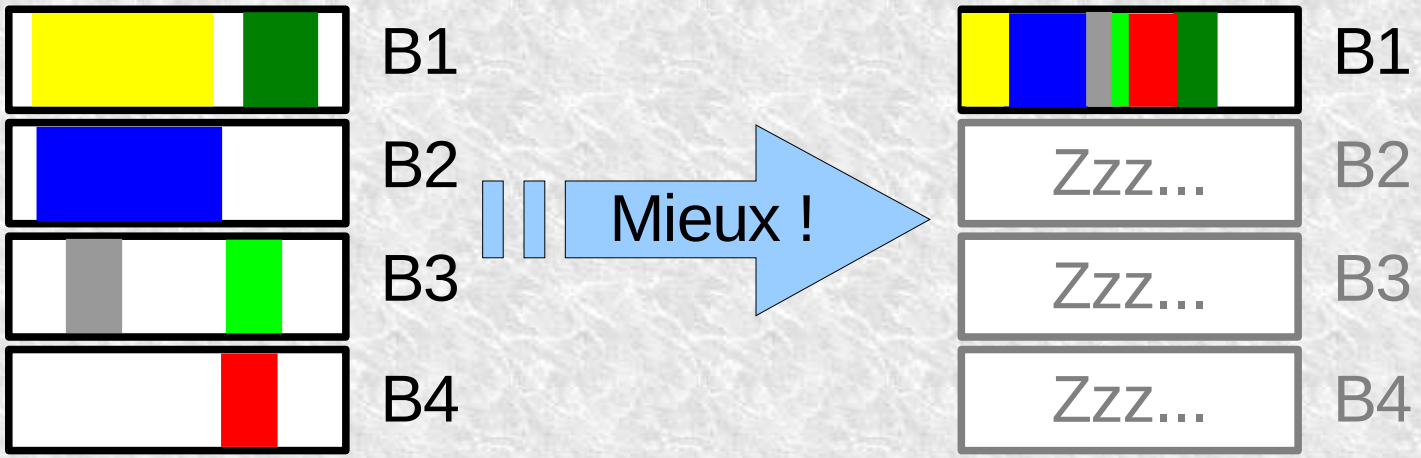
- Changement de fenêtre selon phase du programme
  - «Une phase tient dans une fenêtre»
- Réduit débordements en mémoire quand pas assez de registres (*register spill*)
- Surcoût de gestion (changements de fenêtre: échange registres ↔ RAM)

4) Fenêtres de registres: impact

- Travail en registres plutôt qu'en mémoire:
  - Transferts--
  - (Mode repos)++
  - Vitesse++ (créé pour cela)
    - [Ravindran2005] +11%
  - Énergie--
    - [Ravindran2005] -25%

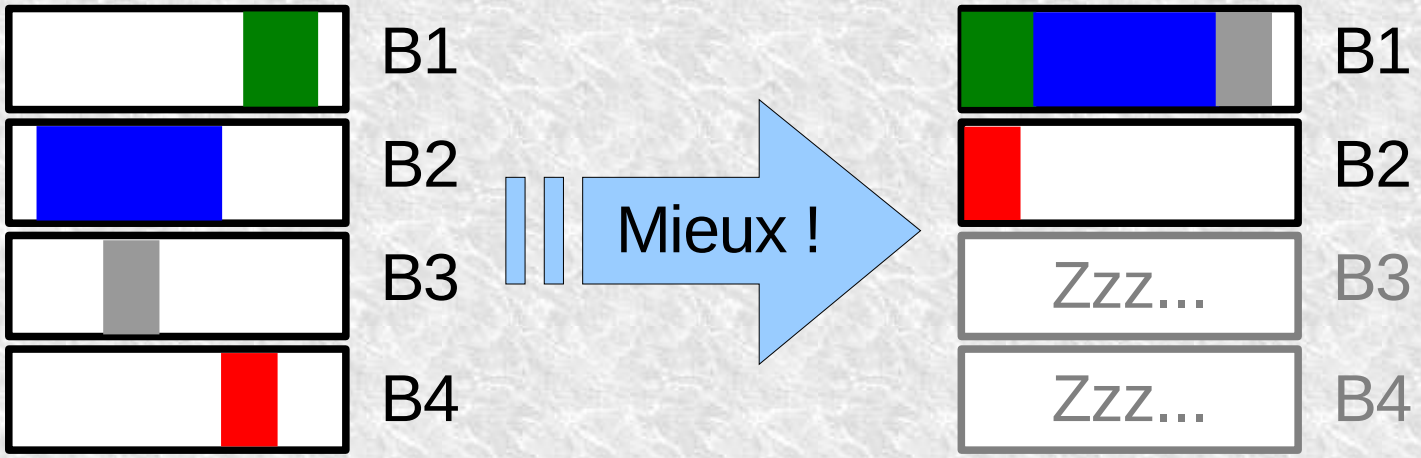
5) Compactage: idée

- Compactage = Moins de place
  - Moins d'énergie
  - Opportunités de mise en repos (bancs mémoire)



5) Compactage: contiguïté

- Allouer et conserver les données dans un minimum de zones bien remplies
  - Éventuels déplacements (éviter fragmentation)
  - Parfois contraire à vitesse (qui peut préférer des accès en parallèle à différents bancs)

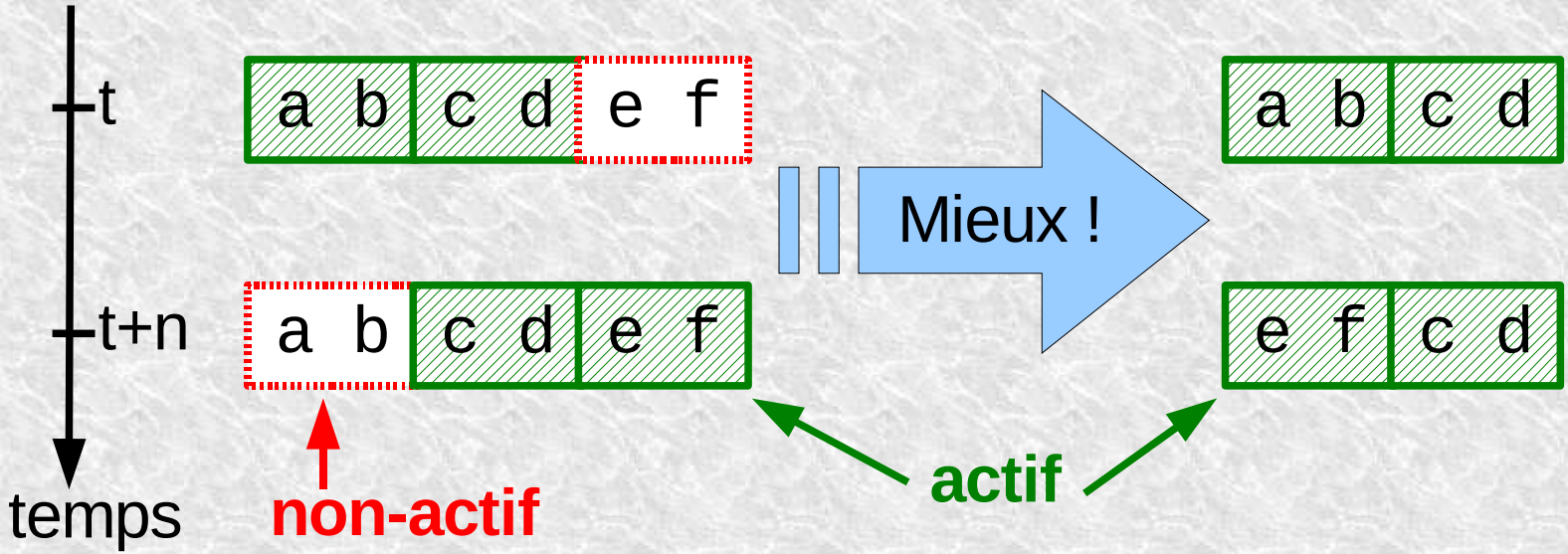


5) Compactage: *coalescing*

- Fusion de variables (*coalescing*): *n* «petites» données dans 1 emplacement
  - *Subword data*
    - *Bitwidth aware register allocation*

5) Compactage: *coalescing*

- Analyse de durée de vie: données qui ne coexistent pas au même emplacement



5) Compactage: compression

- Compression de données (plus large échelle):
  - Attention au surcoût potentiel
  - Fortes opportunités
    - Taille données --
    - Mises en repos
    - Vie longue, accès rares

5) Compactage: impact

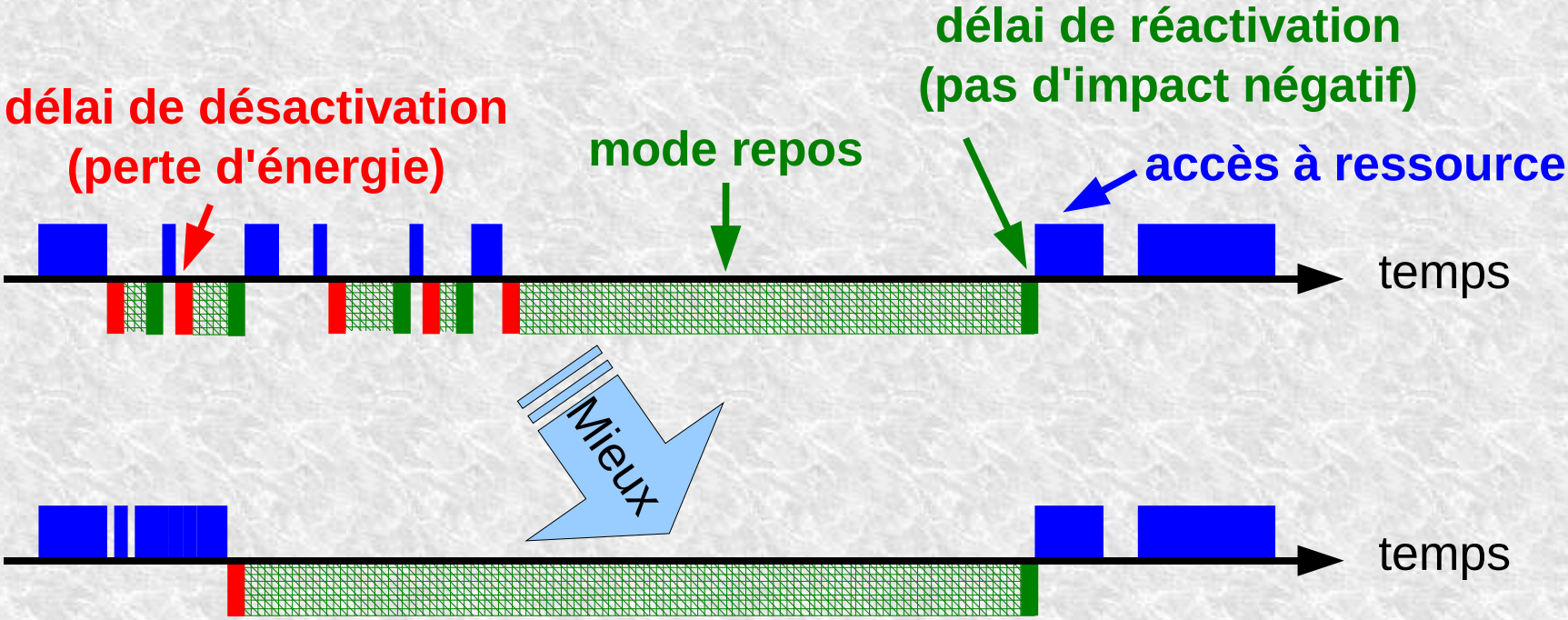
- Sur variables [Zhuang2003]:
  - Cycles -3%
  - Pile -69%
- Sur registres [Tallam2003]: -10 à -50% de registres
- Sur données (champs) [Zhang2002]:
  - Tas: -25%
  - Énergie: -30%
  - Temps d'exécution: -12%
    - avec *Data Compression eXtensions* sur l'ISA: -30%

6) Ré-ordonnancement d'accès: principe

- Améliorer la localité
  - Grouper les accès aux ressources
- Augmenter les périodes où une ressource n'est pas utilisée
- Favorise les modes repos

6) Ré-ordonnancement d'accès: niveau code

- Changements sur le code
  - Avancer certains accès:

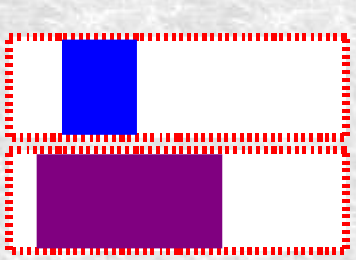


6) Ré-ordonnancement d'accès: niveau boucles

- Division de boucles (*loop fission*)
  - 1 boucle devient n boucles
  - Travaillent sur des données différentes (tableaux...): meilleure localité, mises en repos
  - [Graybill2002,ch10] Énergie-- sur boucle la + coûteuse > énergie++ sur les autres (contrôle)

6) Ré-ordonnancement d'accès: division de boucles

```
for(i=0;i<10000;i++){
        ...a[...];
        ...b[...];
}
```



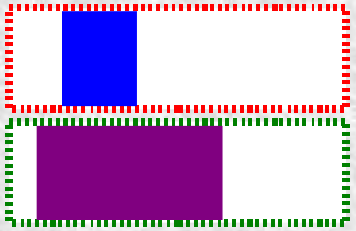
B1

B2

actifs

Mieux !

```
for(i=0;i<10000;i++){
        ...a[...];
}
```

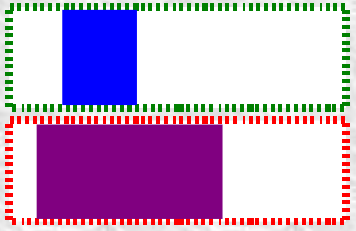


B1 **actif**

B2 **repos**

puis

```
for(i=0;i<10000;i++){
        ...b[...];
}
```



B1 **repos**

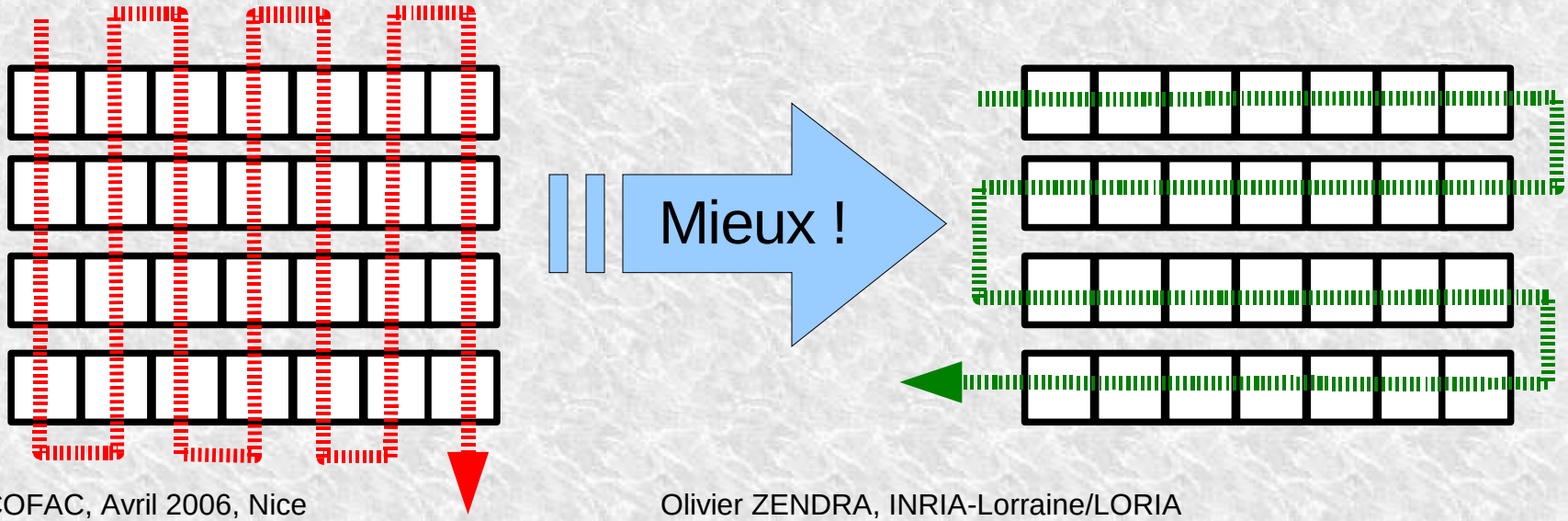
B2 **actif**

6) Ré-ordonnancement d'accès: niveau données

- Changement de *data layout*
- Dual de changements sur code
- Très intéressant sur tableaux

6) Ré-ordonnancement d'accès: niveau données

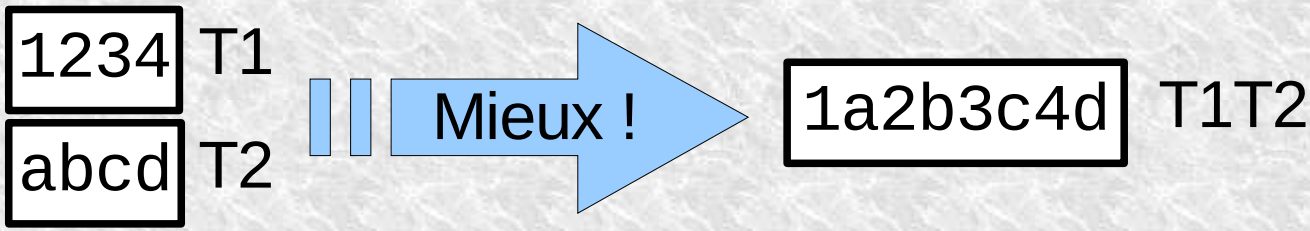
- Tableaux: accès selon stockage
  - Énergie -10% / contrôle de mode de base [Athavale2001]



6) Ré-ordonnancement d'accès: niveau données

- Entrelacement de tableaux (accédés concomitamment):
  - Energie -8% / contrôle de mode de base [Athavale2001]

```
for(i=0;i<10000;i++){
        ...T1[x];
        ...T2[x];
}
```



7) Mémoire «*scratch-pad*»: motivation

- Caches = vitesse++
- Mais peu adaptés aux systèmes embarqués
  - Taille circuits++ (cache+logique)
  - Consommation énergie++
  - Peu prévisibles: problème en temps réel
- Nombreux systèmes sans cache

7) Mémoire «*scratch-pad*» (SPM): principe

- Petite zone mémoire rapide (SRAM,...)
  - Comme cache
- Gérée directement et explicitement au niveau logiciel
  - Pas de circuiterie pour sa gestion
  - Par le développeur
  - Par le compilateur

7) Mémoire «*scratch-pad*»: avantages / caches

- Taille-- (mémoire sans logique)
  - [Banakar2002] -34% / cache
- Coût--
- Énergie--
  - [Banakar2002] -40% / cache
- +Prévisible

7) Mémoire «*scratch-pad*»: domaines d'application

- TB si accès aux données réguliers et connus
  - Multiplications matricielles, algorithmes de compressions audio-vidéo, filtrage...
- B (>cache) si *mapping* en SPM optimal basé sur probabilités d'accès
  - Listes, arbres n-aires à topologie peu variable [Absar2006]

7) Mémoire «*scratch-pad*»: gestion statique

- Choix (placements) faits entièrement hors ligne (à la compilation)
  - Pas de déplacement
  - Prise en compte d'informations de l'exécution avec profils d'exécution (*profiling*)
- Bonnes performances
- Bonnes caractéristiques temps réel

7) Mémoire «*scratch-pad*»: gestion statique: exemple

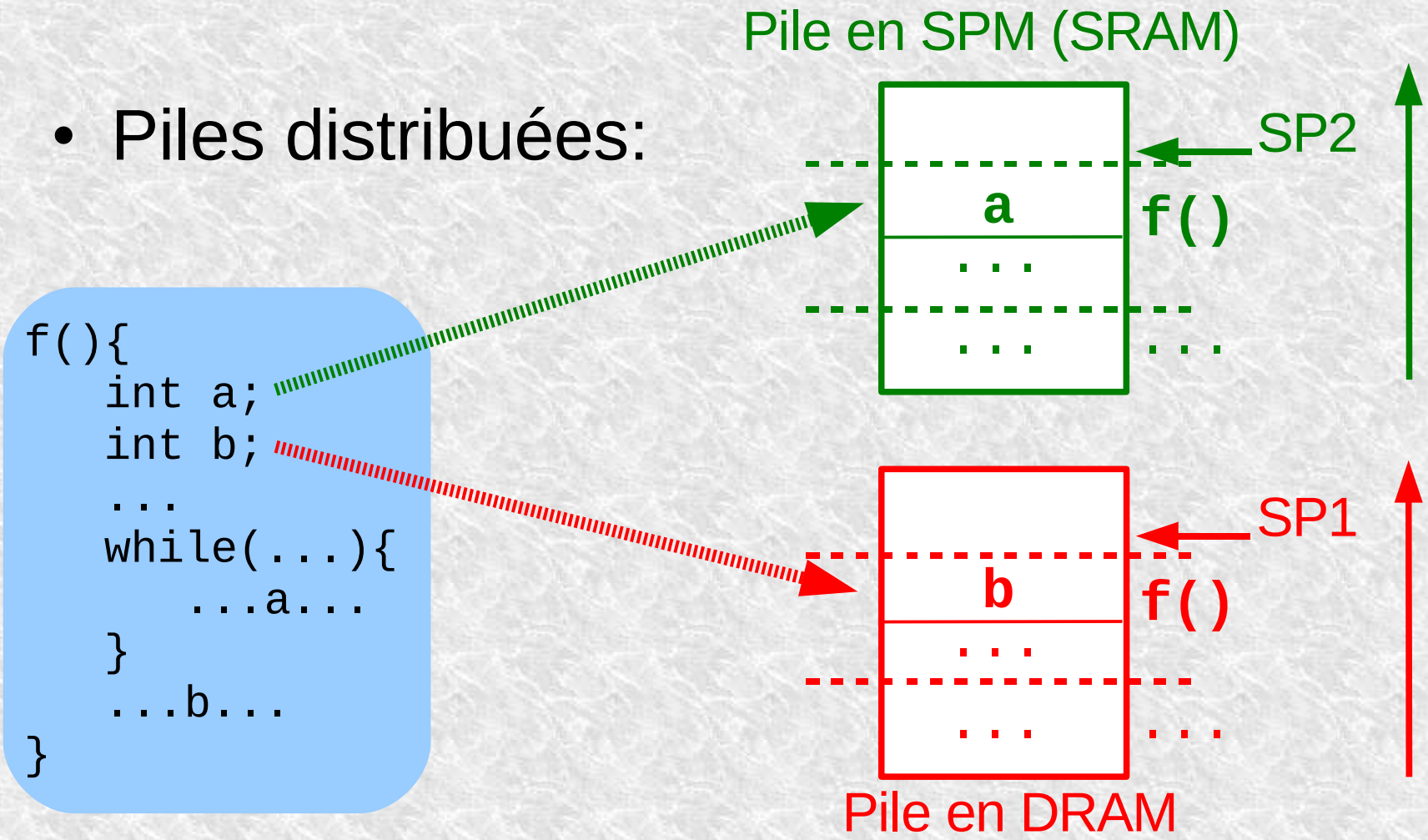
- [Avissar2002]
- N niveaux mémoire
- 3 types d'allocation statique:
  - Avide & pile unifiée: +petits en SPM & pile unique en DRAM (classique)
  - Avide & pile distribuée: ... & pile en plusieurs morceaux (SPM, DRAM...)
  - (le meilleur) Formulation linéaire & pile distribuée: optimisation sur le modèle & ... : temps d'exécution -56% / tout en DRAM

7) Mémoire «*scratch-pad*»: gestion statique: exemple

- Avec profils: fréquences d'accès
- Piles distribuées
  - Variables locales les plus «chaudes» en SPM
  - Plusieurs pointeurs de pile (SPx) à mettre à jour en entrée/sortie de procédure
    - Mieux avec 1 seul pour procédures courtes (toutes variables en même pile)
  - Temps d'exécution -44% / pile unique !
  - Optimum si pas de récursivité

7) Mémoire «*scratch-pad*»: gestion statique: exemple

- Piles distribuées:



7) Mémoire «*scratch-pad*»: gestion statique: exemple

- Optimisation sur formulation linéaire
  - Temps d'exécution -11%
- 20% des données en SRAM presque aussi efficace que 100% sur tous tests
  - Sur certains tests, idem avec <=5% en SRAM
  - Éviter DRAM seule
- Optimal pour globales
- Pas le tas...

7) Mémoire «*scratch-pad*»: gestion statique: exemple

- Marche aussi en multi-programme
- Partitionner toute la SRAM entre les programmes à partir des profils en exécution collective
  - Optimal si exécutés ensemble

8) Mémoire «*scratch-pad*»: gestion dynamique: principes

- Allocation dynamique (exécution) mais décidée à la compilation
- (Dé)placements faits à l'exécution
- Régions, phases du programme, au lieu de programme entier
- Plus complexe, plus récent

8) Mémoire «*scratch-pad*»: gestion dynamique: principes

- Choix d'allocation basé sur
  - Fréquence d'utilisation
  - Coûts de transfert
  - Taille

8) Mémoire «*scratch-pad*»: gestion dynamique: exemple

- Version statique:

```
int a[800];          // SPM (1000)
int b[800];          // DRAM
...
while (i<100000) a[...]... // OK

while (i<100000) b[...]... // bof
...a...b...
```

8) Mémoire «*scratch-pad*»: gestion dynamique: exemple

- Version dynamique:

```
int a[800];          // SPM (1000)
int b[800];          // DRAM
...
while (i<100000) a[...]... // OK
// copie a->DRAM, puis b->SPM
while (i<100000) b[...]... // OK
...a...b...
```

8) Mémoire «*scratch-pad*»: gestion dynamique: avantages

- Meilleure (ré)utilisation de la mémoire
  - Fin d'utilisation (temporaire) = libération immédiate du SPM possible
- Meilleure sur situations plus complexes
  - Création dynamique de tâches, taille de données variable, etc. (MPEG21, MPEG4)

8) Mémoire «*scratch-pad*»: gestion dynamique: inconvénients

- Temps réel +difficile
- Taille++ (logique)
- Surcoût de gestion en T et E / statique
  - Logique
  - Transferts SPM-RAM
    - Coût diminué par support DMA [Francesco2004]
    - Allocation directe en SPM possible
      - Coût de transfert = 0

8) Mémoire «*scratch-pad*»: gestion dynamique: algo

- [Dominguez2005]: Première méthode de compilation permettant une gestion dynamique du SPM qui alloue des données du tas en SPM
  - Taille allouée à un *site* (malloc / new) inconnue
    - Taille <= Taille(SPM) ?
  - Déplacement de mémoire = pointeurs invalides
    - Mises à jour chères

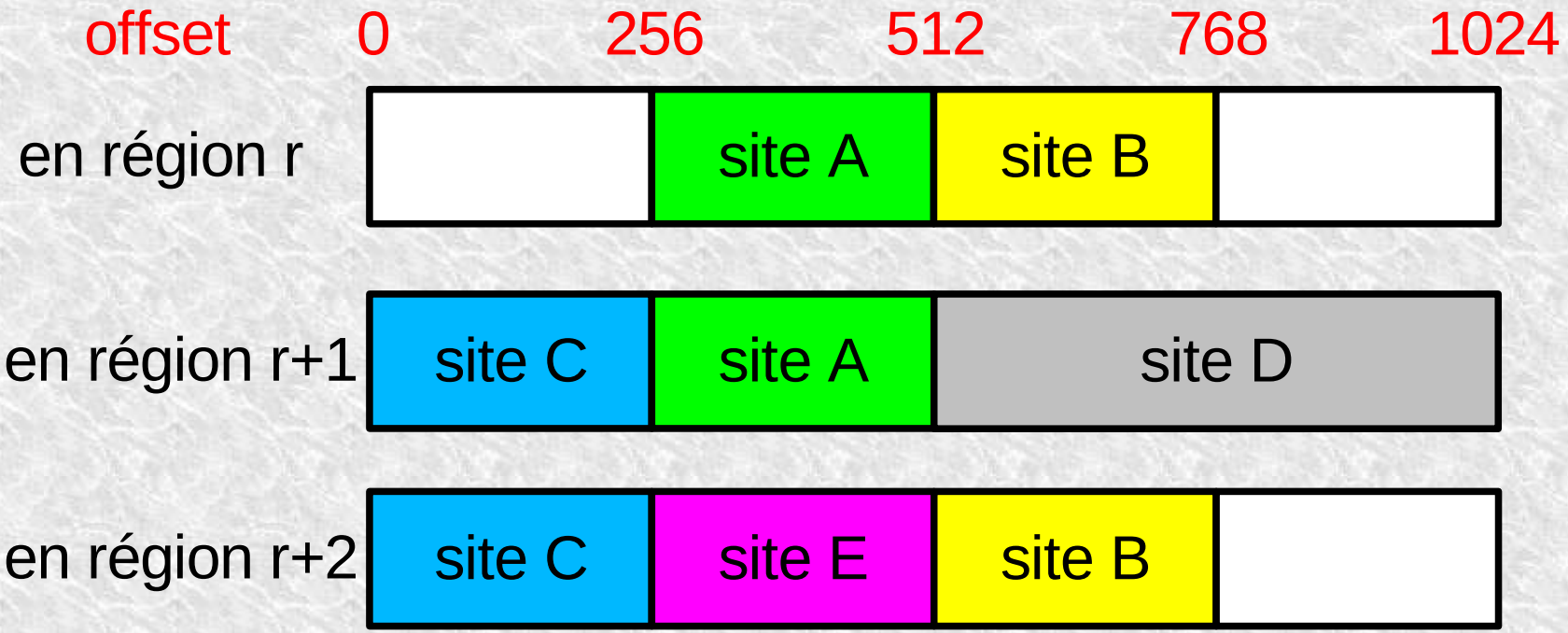
8) Mémoire «*scratch-pad*»: gestion dynamique: algo

- 3 étapes:
  - Partitionne le programme en régions (début-fin de procédure et boucle)
  - Détermine ordre d'exécution entre régions
  - Insère du code en début de région *r* pour copier des parties du tas (*bins*) vers/hors SPM (selon utilisation en *r*)

8) Mémoire «*scratch-pad*»: gestion dynamique: algo

- Un *bin* = un sous-ensemble de taille fixe des objets alloués à un site
  - Taille(*bin*) <= Taille(SPM) est garanti
  - Offset(*bin*) figé (si *bin* présent en région)
    - Pointeurs restent valides après allers-retours entre SPM et DRAM
  - Heuristique basée sur modèle de coût pour décider taille et contenu des *bins*
    - Sites où fréquence d'accès par octet > ont *bin* + grand

8) Mémoire «*scratch-pad*»: gestion dynamique: algo



8) Mémoire «*scratch-pad*»: gestion dynamique: résultats

- Temps d'exécution: -35% par rapport à un placement statique (sauf du tas) en SPM
- Énergie: -40% par rapport à un placement statique (sauf du tas) en SPM

9) Importance d'une analyse globale du système

- = optimisations inter-programmes
- Ordonnancement: intrinsèque
- Matériel: tous programmes considérés, mais pas ensemble
- Gestion mémoire:
  - OS: multi-programme
  - Application: mono-programme

9) Importance d'une analyse globale du système

- Compilation: plutôt mono-programme
  - Surtout compilation statique
  - Compilation dynamique: multi-programme (JVMs...)
- Crucial pour maximiser les gains
  - Ex. taille tampon mémoire et groupement accès disque: énergie -7% à -49% avec optim multi par rapport à optim mono [Hom2005]

Conclusion et perspectives

- Complémentarité matériel / compilation:
  - Compilation: contexte beaucoup plus large possible (beaucoup de ressources)
  - Mais comportement exact à l'exécution plus difficile à saisir
  - Essayer d'avoir les deux à la foi
    - Machines virtuelle optimisante fait cela. Mais coûteux en ressources à l'exécution !

Conclusion et perspectives

- Besoin de support pour interface matériel-logiciel (compilateur) au niveau ISA: synergies
  - Gestion «directe» de ressources par compilateur
  - Co-optimisations compilateur + matériel, avec transmissions d'informations de l'un à l'autre

Conclusion et perspectives

- Processeurs VLIW, EPIC: fort potentiel avec parallélisme
  - Vitesse++
  - Intéressant en énergie
  - Compilateur doit fournir le parallélisme
    - Gros travail (pas encore là pour processeurs génériques)

Conclusion et perspectives

- Importance de la mémoire et de son utilisation: 70% à 90% de l'énergie en 2010 [ITRS]
  - SPM
  - *Energy-aware Garbage Collectors ?*

Références

- [Absar2006] Mohammed Javed Absar, Francky Catthoor: *Compiler-Based Approach for Exploiting Scratch-Pad in Presence of Irregular Array Access*, DATE 2005, IEEE Computer Society
- [Athavale2001] R. Athavale, Narayanan Vijaykrishnan, Mahmut T. Kandemir, Mary Jane Irwin: *Influence of Array Allocation Mechanisms on Memory System Energy*. IPDPS 2001: 3.
- [Avissar2002] O. Avissar, R. Barua D. Stewart: *An Optimal Memory Allocation Scheme for Scratch-Pad Based Embedded Systems*. ACM Transactions on Embedded Computing Systems (TECS), 1(1),pp. 6-26, November 2002.
- [Banakar2002] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, Peter Marwedel: *Scratchpad memory: design alternative for cache on-chip memory in embedded systems*. CODES 2002: 73-78

Références

- [Dominguez2005] Angel Dominguez, Sumesh Udayakumaran, Rajeev Barua: *Heap Data Allocation to Scratch-Pad Memory in Embedded Systems*. Journal of Embedded Computing, 1(4), 2005, IOS Press.
- [Francesco2004] Poletti Francesco, Paul Marchal, David Atienza, Luca Benini, Francky Catthoor, Jose Manuel Mendias: *An integrated hardware/software approach for run-time scratchpad management*. DAC 2004: 238-243
- [Graybill2002] Robert Graybill, Rami Melhem: *Power aware computing*, 2002, Kluwer Academic Publishers.
- [Hom2005] Jerry Hom, Ulrich Kremer: *Inter-program optimizations for conserving disk energy*. ISLPED 2005: 335-338.
- [ITRS] International Technology Roadmap for Semiconductors. http://public.itrs.net/

Références

- [Kandemir2000] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, W. Ye, I. Demirkiran: *Register relabeling: A post-compilation technique for energy reduction*, Workshop on Compilers and Operating Systems for Low Power, October 2000.
- [Lee1997] M. Lee, V. Tiwari, S. Malik, M. Fujita: *Power analysis and minimization techniques for embedded DSP software.* IEEE Trans. Very Large Scale Integration, vol. 5, pp. 123--135, Mar. 1997.
- [Ranvindran2005] R.A. Ravindran, R.M. Senger, E.D. Marsman, G.S. Dasika, M.R. Guthaus, S.A. Mahlke, R.B. Brown: *Partitioning variables across register windows to reduce spill code in a low-power processor*, IEEE Transactions on Computers, Vol. 54, Issue 8, 2005, pp. 998-1012.
- [Tallam2003] Sriraman Tallam, Rajiv Gupta: *Bitwidth aware global register allocation*. POPL 2003: 85-96.

Références

- [Zhang2002] Youtao Zhang, Rajiv Gupta: *Data Compression Transformations for Dynamically Allocated Data Structures*. CC 2002: 14-28.
- [Zhuang2003] Xiaotong Zhuang, ChokSheak Lau, Santosh Pande: *Storage assignment optimizations through variable coalescence for embedded processors*. LCTES 2003: 220-231

benchmarks utilisés

- [Avissar2002]: Consortium Trimaran (bmm, fir), Rutter (btoa), MiBench (crc32, djikstra), UTDSP (fft, iir, latnrm)
- [Athavale2001]: adi, amhmtm, btrix, dtdtz, eflux, tomcat, tsf, vpenda
- [Dominguez2005]: Huff, Drystone, Susan, Gsm, KS
- [Hom2005]: mpeg_play, mpg123, sftp
- [Ravindran2005]: fir, rawd, sha, g721enc, g721dec, gsmenc, gsmdec, epic, unepic, cjpeg, djpeg, rijndael, pgpenc, pgpdec (Mediabench & MiBench)
- [Tallam2003]: Mediabench (adpcm, g721), NetBench (crc, dh), Bitwise du MIT (SoftFloat, NewLife, MotionTest, Bubblesort, Histogram), thres
- [Zhang2002]: treeadd, bisort, tsp, perimeter, health, mst
- [Zhuang2003]: epic, gsm, g721, Mpeg2d, Mpeg2e, Bzip2, Gzip, Mcf, Twolf, Vpr (MediaBench & Spec2000int)