Systèmes intégrés multiprocesseur

Applications, architectures et logiciels embarqués

Frédéric Pétrot
Département ASIM/LIP6
Université Pierre et Marie Curie

Plan de la présentation

- Problématique
- Modélisation d'applications en vue d'implantations matérielles/logicielles
- Quelques architectures matérielles
- Implantation d'une spécification
- Logiciel embarqué

Plan de la présentation

- Problématique
- Modélisation d'applications en vue d'implantations matérielles/logicielles
- Quelques architectures matérielles
- Implantation d'une spécification
- Logiciel embarqué

Problématique

Implanter une application sur matériel/logiciel

- Coût minimal pour production de masse
- Temps de conception contraint

⇒ Chemin de la spécification à l'implantation

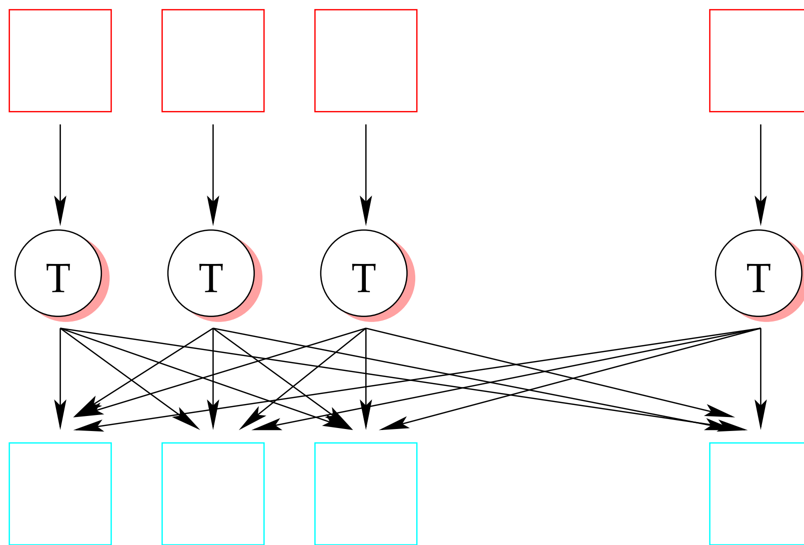
- Sémantique de spécification
- Transformations applicables
- Architecture(s) cible(s) : processeurs, coprocesseurs, noyaux, communications, ...

⇒ Espace des solutions vaste

Problématique

Applications décrites comme :

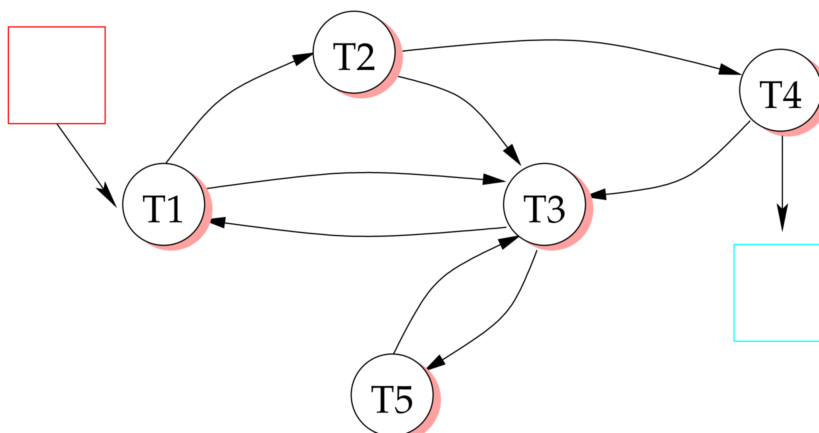
- un programme identique s'exécutant n fois (flot de données)
  $\Rightarrow$ ensemble de tâches indépendantes : Network processing



Problématique

Applications décrites comme :

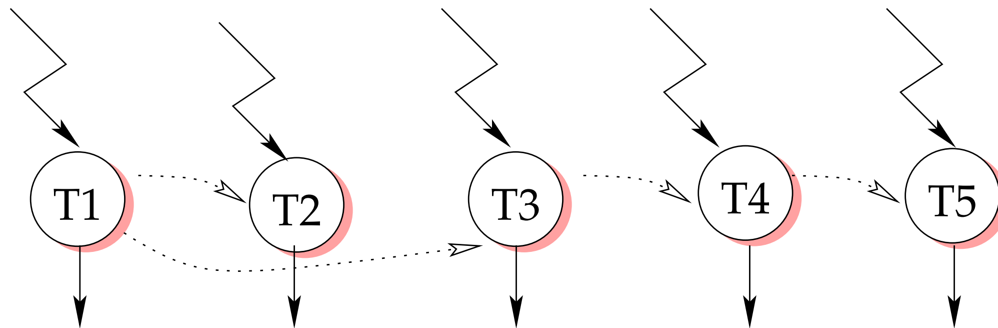
- graphe de tâches communicantes (flot de données)
  $\Rightarrow$ ensemble de tâches se partageant des ressources : Multimédia



Problématique

Applications décrites comme :

- une machine à états (contrôle/commande)
  ⇒ gestion asynchrone d'évènements : Automobile



Problématique

Problèmes à résoudre :

1. Implantation optimisée matériel/logiciel
2. Conservation de la sémantique de haut niveau

Contraintes :

1. Temps de conception
2. Débit, latence, consommation, surface, souplesse
3. Architectures visées : matériel et logiciel existant partiellement

De l'avenir du matériel spécialisé

Source : Dwight W. Decker, Conexant Systems, Inc.

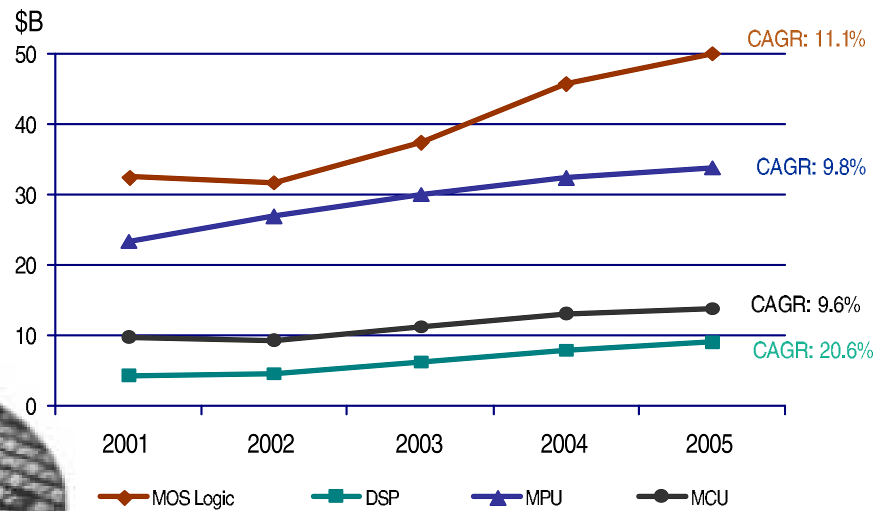
MIDYEAR FORECAST LUNCHEON 2002

Product Segment Forecast

SIA

CREDIT SUISSE FIRST BOSTON Technology Group

MOS Logic, DSP, Microcontroller, and Microprocessor



Source: SIA June 2002 Forecast 02-0270 15

De l'avenir du matériel spécialisé

Source : Dwight W. Decker, Conexant Systems, Inc.

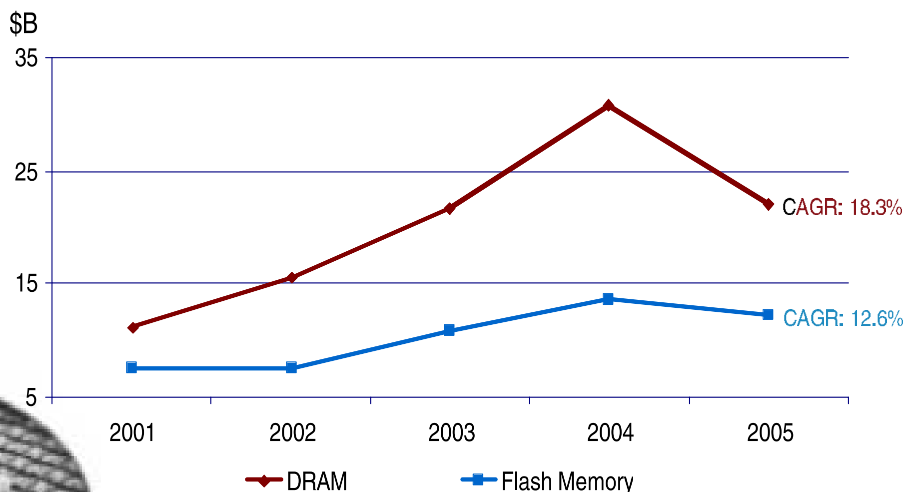
MIDYEAR FORECAST LUNCHEON 2002

Product Segment Forecast

SIA

CREDIT SUISSE FIRST BOSTON Technology Group

DRAM and Flash Memory

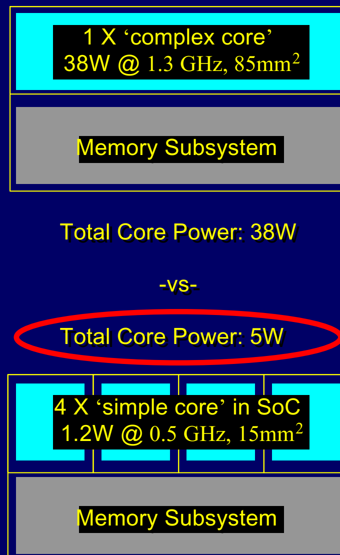


Source: SIA June 2002 Forecast 02-0270 16

De l'avenir du matériel spécialisé

Source : John Cohn, IBM

SoC Architecture for Low Power µP



Problem

- Big uni-processors getting too hot !
- Yield, design cost, TAT also degrading

Solution

- Combine small, fast cores in SoC style
- Exploit multi-thread parallelism
- Dramatically lower power
- While preserving system-level throughput
- .. At cost in uni performance, sw complexity

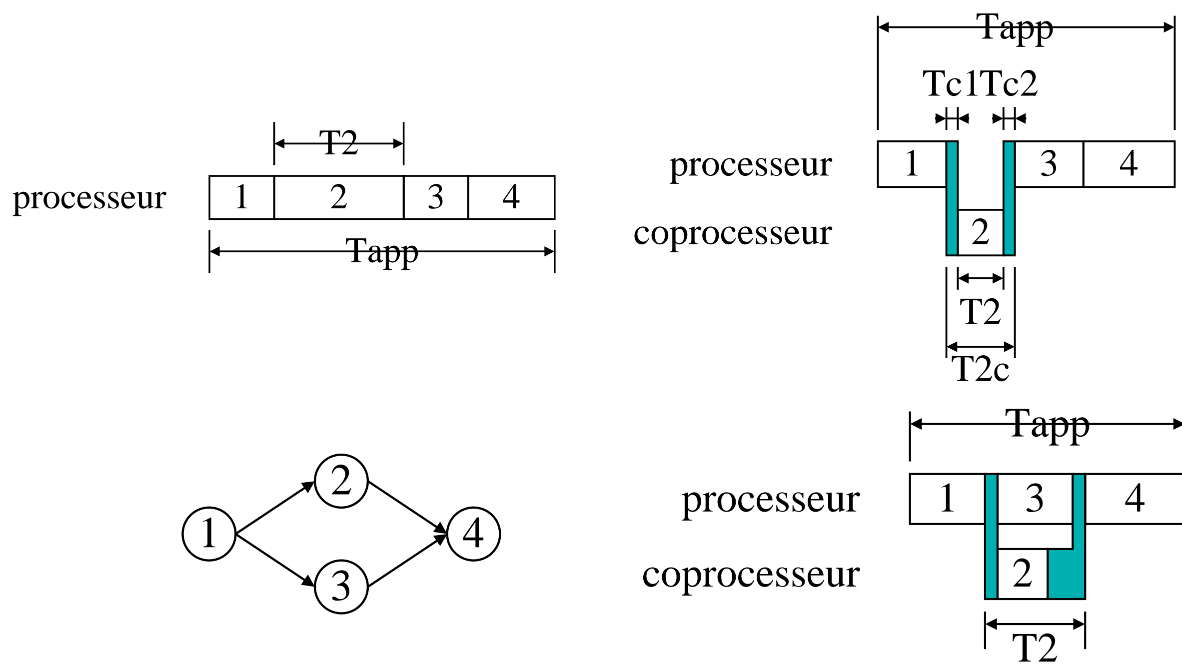
Bonus:

- Improved power, design cost, TAT

Plan de la présentation

- ▶ Problématique
- ▶ Modélisation d'applications en vue d'implantations matérielles/logicielles
- ▶ Quelques architectures matérielles
- ▶ Implantation d'une spécification
- ▶ Logiciel embarqué

Spécification: intérêt du parallélisme



Spécifications

Initialement :

- Application séquentielle
  Langage de programmation classique (C, C++)

Réseaux de « Kahn » :

- Extraction du parallelisme à gros grain
  ⇒ Graphe de tâches qui communiquent à travers des FIFOs
- FIFO:
  - FIFO de taille infinie sans pertes
    Lectures bloquantes
    ⇒ comportement indépendant des dates d'exécution des tâches
  - En réalité : FIFO de taille finie, écritures bloquantes
    ⇒ propriétés conservées
    ⇒ risque de *deadlock*

Modèle de communication

Actions liées à la communication par FIFO

- Transfert des données entre les 2 bouts du tuyau
- Mise à jour de la variable d'état de la FIFO
  $\Rightarrow$ partagée par le producteur et le consommateur
  $\Rightarrow$ pas d'accès simultanée à une ressource partagée
- Suspension de la tâche tentant de consommer d'une FIFO vide (ou de produire dans une FIFO pleine)
- Réveil d'une tâche lorsque des données sont disponibles (ou des cases sont vides)

Implementation $\Rightarrow$ dépend clairement de la nature HW/SW des tâches

Plan de la présentation

- Problématique
- Modélisation d'applications en vue d'implantations matérielles/logicielles
- Quelques architectures matérielles
- Implantation d'une spécification
- Logiciel embarqué

Les CPUs usuels pour l'embarqué

Rappel utile, ...

Chip Category	Number Sold
Embedded 4-bit	2000 million
Embedded 8-bit	4700 million
Embedded 16-bit	700 million
Embedded 32-bit	400 million
DSP	600 million
Desktop 32/64-bit	150 million

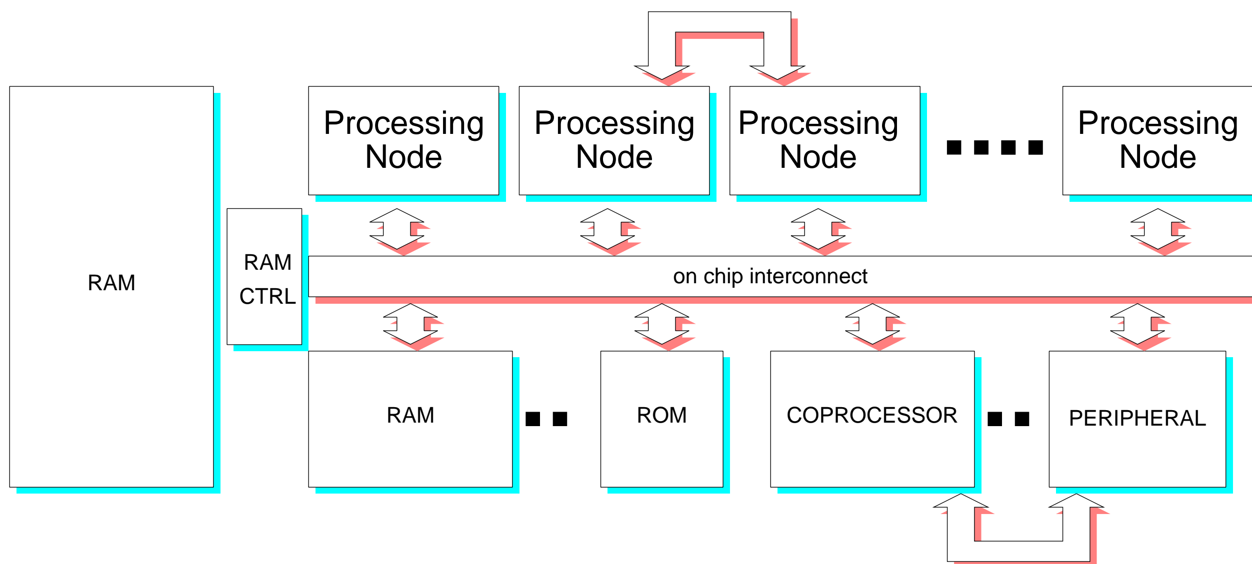
David Tennenhouse (Intel Director of Research). Keynote Speech at the 20th IEEE Real-Time Systems Symposium (RTSS'99), December 1999

32-bit Family	Number Sold
ARM	151 million
Motorola 68k	94 million
MIPS	57 million
Hitachi SuperH	33 million
x86	29 million
PowerPC	10 million
Intel i960	8 million
SPARC	3 million
AMD 29k	2 million
Motorola M-Core	1 million

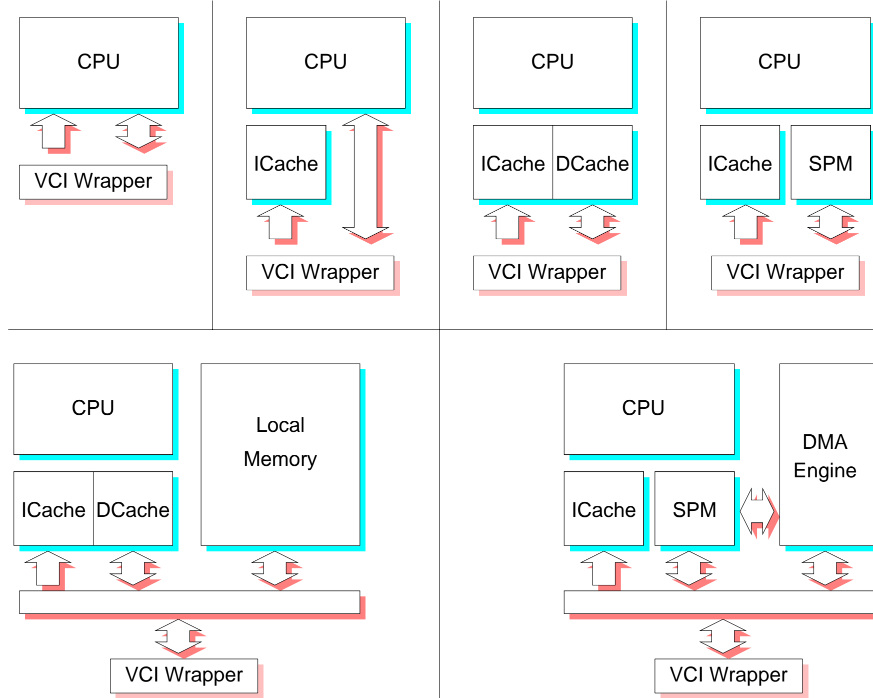
T. R. Halfhill. Embedded Market Breaks New Ground. Microprocessor Report, January 2000

Architecture hautes performances

Plateforme multiprocesseur à mémoire partagée :

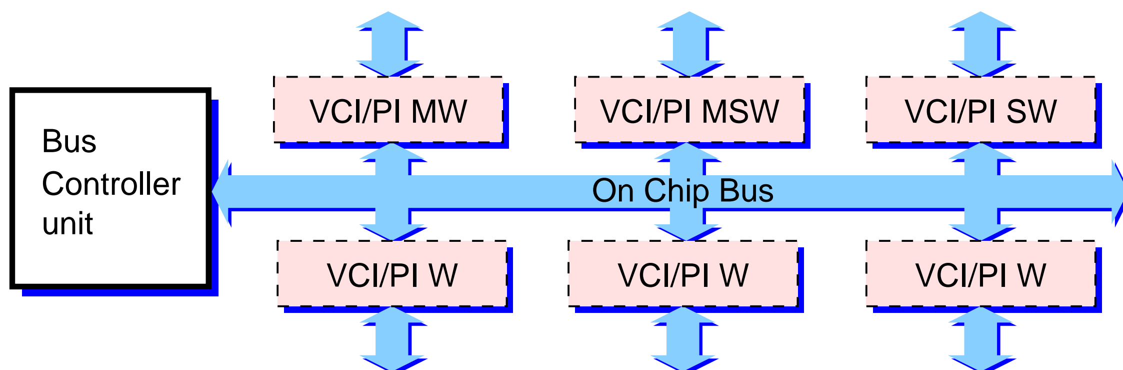


Exemples de nœuds de calculs



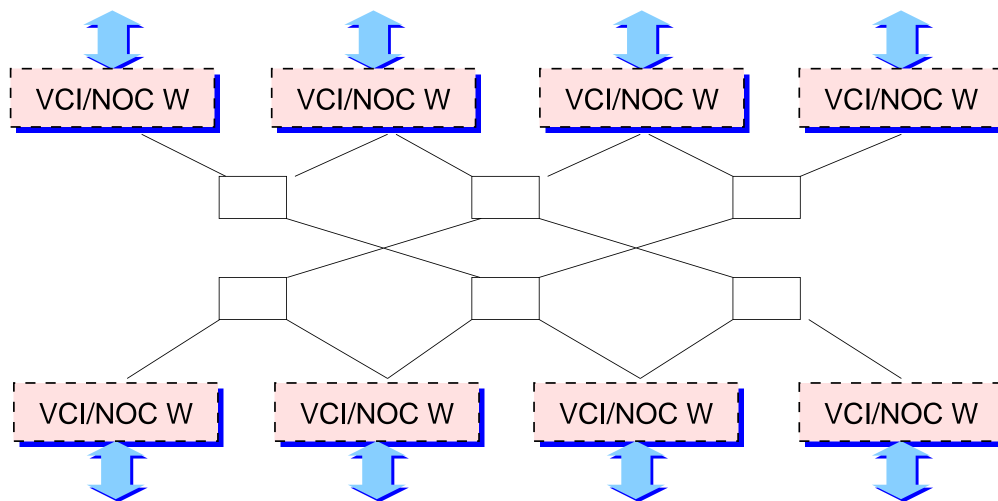
- ▶ Expertise LIP6 : MIPS R3000 (mc), Sparc V8, Caches, DMA, ITC

Exemples d'interconnexions



- ▶ Expertise LIP6 :
  - ⇝ Bus : PI-Bus ⇒ VCI ↔ PI wrappers, BCU ;

Exemples d'interconnexions



- Expertise LIP6 :
  - ⇝ Réseau : Spin ⇒ VCI ↔ Spin wrappers ;

En résumé

Intérêts/particularités du SoC :

- Système à mémoire partagée à large échelle
- Latence moyenne assez faible
- Nombre d'I/O sur core peut être grand
- Nombre de fils peut être grand
- CPU sans mémoire virtuelle
- Séparation user/kernel non significative

Plan de la présentation

- Problématique
- Modélisation d'applications en vue d'implantations matérielles/logicielles
- Quelques architectures matérielles
- Implantation d'une spécification
- Logiciel embarqué

Support d'exécution des tâches

Sur processeur ou coprocesseur spécialisé ?

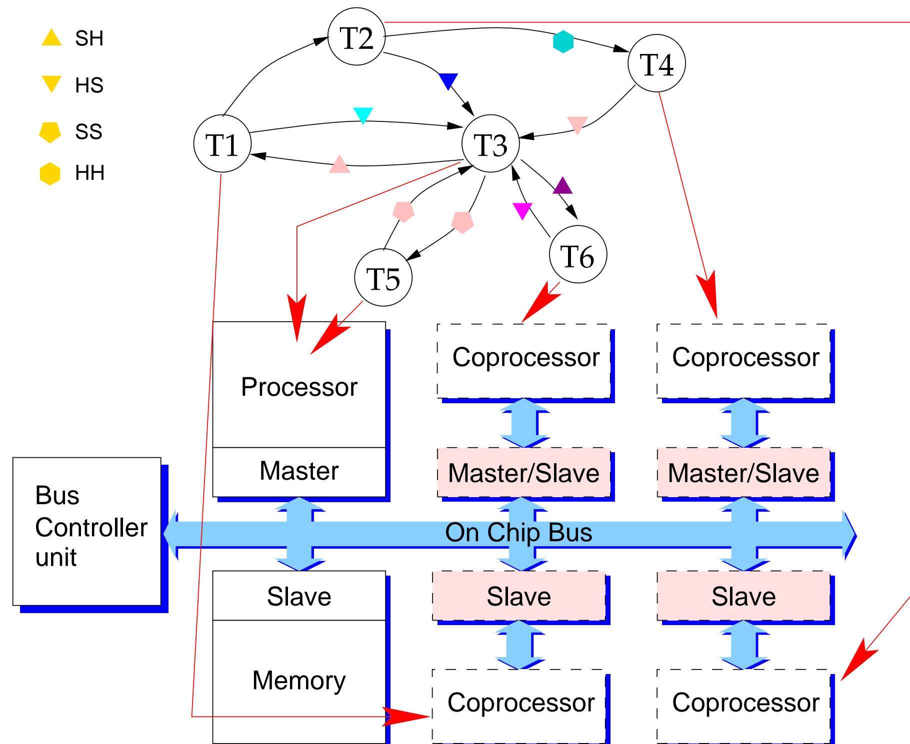
Fondamentalement :

- Performance
- Consommation
- Surface

Mais aussi :

- Temps et coût de développement
- Possibilité de reprogrammer *a posteriori*
- Partage efficace des ressources matérielles

Exemple d'affectation



Implantation des communications

- Assurer le transfert des données sans pertes :
  - ⇝ entre tâches logicielles
  - ⇝ entre une tâche logicielle et un coprocesseur
  - ⇝ entre coprocesseurs
- Minimiser les coûts de synchronisation :
  - ⇝ traitement des interruptions
  - ⇝ bande passante du bus/latence du réseau
  - ⇝ commutation de contexte processeur

Support matériel

Standard bien venu, ...

- Au niveau protocole : VCI, Amba, ...
- Au niveau « sémantique » :
  Pilote spécifique qui assure cela par coprocesseur

En théorie, on aimerait :

- Un module d'interface Maître/Esclave avec interface FIFO
- Un petit nombre de pilotes

En pratique :

- C'est la jungle !

Plan de la présentation

- Problématique
- Modélisation d'applications en vue d'implantations matérielles/logicielles
- Quelques architectures matérielles
- Implantation d'une spécification
- Logiciel embarqué

Logiciel(s) embarqué(s)

- Noyaux : très liés
  - ⇝ au(x) processeur(s)
  - ⇝ à l'interconnect

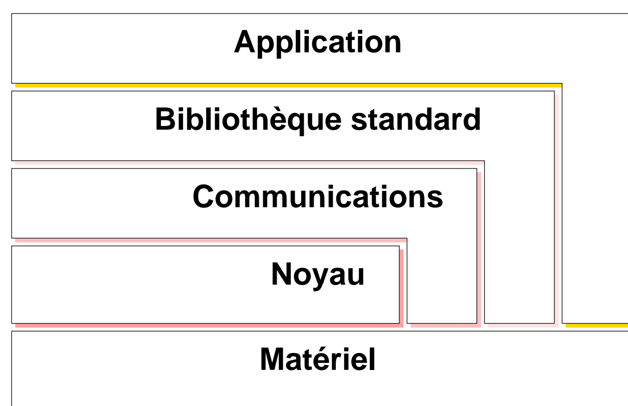
  Ex : *VxWorks, VRTX, pSOS, OSEK/VDX, Chorus, Windows CE, eCos, RTEMS, LynuxOS, Mythos, rtmk, ...*
- Couches de communication : très liées
  - ⇝ au(x) processeur(s)
  - ⇝ à l'interconnect
  - ⇝ au matériel spécialisé
  - ⇝ aux interruptions
- Bibliothèque de fonctions « standards » : dépendent d'une machine virtuelle d'exécution plus ou moins standardisée
- Applicatif

Exemple d'implantation d'un noyau

Services standards des *thread* POSIX

- Multi-tâches
- Multiprocesseur (PRAM)
- Multi bancs mémoire

Exploration d'architecture ⇒ abstraction du matériel délicate



- Pour :
  - ⇒ performance,
  - ⇒ taille,
  - ⇒ adapté au problème
  - ⇒ définition des services
- Contre :
  - ⇒ élégance,
  - ⇒ portabilité

Service de base d'un noyau: initialisation

- création de l'ordonnanceur ;
- création et lancement du *thread* `main` ;
- exécution des tâches.

⇒ masquage des interruptions
⇒ auto-identification des différents processeurs
⇒ allocation mémoire statique (ou dynamique)
⇒ cohérence des accès mémoire partagés
⇒ séquentialisation des accès mémoire partagés

- execution par processeur 0 ;
- attente fin de création ordonnanceur pour processeurs $i \neq 0$;
- execution tâches par processeurs $\forall i$.

École thématique, Roscoff *1 avril 2003*

Initialisation: pseudo-code

$scheduler_created \Leftarrow 0, proc_id \Leftarrow 0$ {Exécuté lors du chargement}
$lock \Leftarrow$ get_lock() {Exécuté par tous les processeurs}
spin_lock($lock$), set_proc_id($proc_id$++), spin_unlock($lock$)
if get_proc_id $= 0$ **then**
  RAZ de la section `.bss`
  positionne une pile pour processeur 0
  création du/des ordonnanceur(s) et du *thread* `main`, $scheduler_created \Leftarrow 1$
  autorise les interruptions
  saut (`goto`) à `main`
else
  positionne une pile pour processeur $i \neq 0$
  while $scheduler_created = 1$ **end while** {Variable `volatile`}
  spin_lock($scheduler$)
  commute() {Processeur disponible pour exécuter un *thread*}
end if

Service de base d'un noyau: changement de contexte

Définition du contexte d'exécution :

- R3000 : `$i, i` $\neq$ `(26, 27, 28)`, `$hi`, `$lo`, `$sr`, `$cr` ;
- Sparc : `%g`, n fenêtres, `%psr`, `%wim`, `%y` ;
- registres flottants et/ou des coprocesseurs (s'il y en a).

⇒ mise à jour de la mémoire (si cache write-back)
⇒ mémoire *scratch pad* fait partie du contexte

Changement de contexte (interruptions globalement masquées) :

- sauvegarde éventuelle du contexte ;
- chargement d'un nouveau contexte.

⇒ contexte peut être sauvé par un processeur et chargé par un autre
⇒ masquage des interruptions identique sur tous les processeurs
⇒ localisation du contexte

Changement de contexte: pseudo-code

save_context($current$) {Sauve le context dans la table $current$}

$current[0] \Leftarrow r_0$

$current[1] \Leftarrow r_1$

$current[i] \Leftarrow r_i$

$current[n] \Leftarrow r_n$

retour à l'appellant

restore_context($current$) {Restore le context à partir de la table $current$}

$r_0 \Leftarrow current[0]$

$r_1 \Leftarrow current[1]$

$r_i \Leftarrow current[i]$

$r_n \Leftarrow current[n]$

retour à l'appellant ... lors du save_context !

Service de base d'un noyau: interruptions

- quel(s) processeur(s) reçoi(ven)t les interruptions ?
- masquage / demasquage global
- masquage / demasquage du timer
- sauvegarde/restoration de l'état : état $\neq$ contexte !
  - ⇝ uniquement les registres temporaires $\neq$ tous les registres
  - ⇝ dans la pile du *thread* $\neq$ dans un tableau unique
    - réentrance
    - entrée sur un proc et sortie sur un autre ordonnancement preemptif
  - ⇝ masquage/demasquage effectué par le matériel
- dépend fortement du matériel (CPU et ITC)
- peut permettre aux CPUs de communiquer

Interruptions: pseudo-code

$tstack = tstack + n + 1$
$tstack[0] \Leftarrow r_0$
$tstack[1] \Leftarrow r_1$
$tstack[i] \Leftarrow r_i$
$tstack[n] \Leftarrow r_n$
Analyse de la cause de l'interruption/exception
Action qui peut suspendre la tâche
$r_0 \Leftarrow tstack[0]$
$r_1 \Leftarrow tstack[1]$
$r_i \Leftarrow tstack[i]$
$r_n \Leftarrow tstack[n]$
$tstack = tstack - n - 1$
retour après l'instruction interrompue

Pointeur de pile : première case vide *à tout moment*

Sous ensemble POSIX retenu

Existant :

- threads: create, exit, real-time task (non-posix)
- spin locks: init, lock, unlock (hw *test & set*)
- mutex: init, lock, unlock
- conditions: init, wait, signal, broadcast
- semaphores: init, wait, post
- join, cancel, yield
- setjmp, longjmp
- signal
- timers: create, delete, settime, gettime, getoverrun

Ordonnancement

Types :
Mutliprocesseur symétrique (SMP)

- unique ordonnanceur centralisé ;
- ordonnanceur exécuté sur tous les processeurs ;
- migration de tâches.

Mutliprocesseur centralisé à tâches affectées
Idem SMP, mais :

- tâches attachées à un processeur⇒ pas de migration de tâches.

Mutliprocesseur distribué à tâches affectées

- autant d'ordonnanceurs que de processeurs ;
- ordonnanceur exécuté par son processeur ;
- tâches attachées à un processeur

Ordonnancement

Préemption :

1. tâche courante pouvant être interrompue sans son accord ;
2. appel système pouvant être interrompu.

Algorithmes d'élection :

FIFO : fair play, sans famine, temps moyen optimisé

Par priorités : permet d'avoir des garanties

- statiques ou dynamiques ;
- caculée en fonction de période, durée, écheance ;
- difficile en cas de partage de ressources ;
- encore plus difficile en multiprocesseur, ...

Ordonnancement

lock : assure l'accès exclusif (inutile en monoprocesseur) ;

stacks : pile(s) spéficique(s), utilisée(s) si CPU en attente ;

wakeup : indique la disponibilité d'une tâche (variable `volatile`) ;

alarm : signal utilisé si dépassement d'échéance ;

date : date courante (en temps absolu) ;

signals : signaux reçus ;

autres : listes de tâches.

Attente CPU

Activée si zéro tâche exécutable :

- deadlocks fonctionnels ;
- attente d'un fin d'exécution matérielle ;
- autres processeurs en cours d'exécution (multipro).

Difficultés :

- plusieurs processeurs en attente simultanément ;
- dernière tâche suspendue peut être réveillée ;
- interruptions matérielles activées.

$\Rightarrow$ incrémentation de $wakeup$ en entrant dans boucle
$\Rightarrow$ décrémentation de $wakeup$ lors du réveil d'une tâche

Attente CPU : pseudo-code

if current **then**
  save_context($current$)
  $current.return_addr \Leftarrow$ adresse de fin de fonction
  pile $\Leftarrow$ pile de l'ordonnanceur
  repeat
    $go \Leftarrow wakeup$, $wakeup{+}{+}$
    spin_unlock($lock$), it_timer_mask, it_global_unmask
    while $wakeup > go$ **end while**
    it_global_mask, it_timer_unmask, spin_lock($lock$)
    $thread \Leftarrow$ scheduler_elect()
  until $thread$ exists
  restore_context($thread$)
end if
fin de fonction

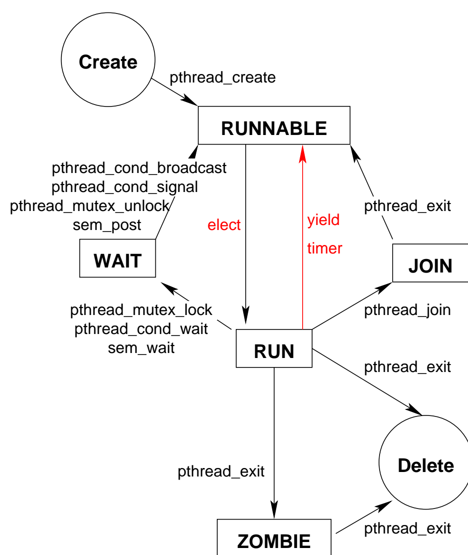
Tâches

Unité d'exécution séquentielle

thread
links
context
father
children
state
signals
handlers
jmpbuf
condition
join
semaphore
mutex

links : pour faire des ensembles de threads ;
context : tout est dit ;
father : *thread* père ;
children : nombre de fils ;
state : état ;
signals : signaux reçus ;
handlers : gestionnaires de signaux ;
jmpbuf : contexte utilisé par `setjmp/longjmp` ;
condition : lien lors d'attente sur condition ;
mutex : lien lors d'attente sur mutex ;
semaphore : lien lors d'attente sur sémaphore ;
join : pointeur sur tâche attendant ma fin ;
attributs : voir la suite !

Tâches



Attributs

- adresse et taille de pile ;
- détachable ?
- héritage de la politique d'élection ?
- politique d'élection (OTHER, FIFO, RR) ;
- paramètres de la politique d'élection ;
- date de début ;
- période ;
- échéance ;
- contexte initial ;
- gestionnaire de dépassement ;
- argument du gestionnaire.

Paramètres de la politique libre

Tâches : allure

▶ tâches classiques :

```
tâche()
begin
   loop
      action
   end loop
end
```

▶ tâches temps réelles :

Pour POSIX $\Rightarrow$ temps réel $\equiv$ priorités

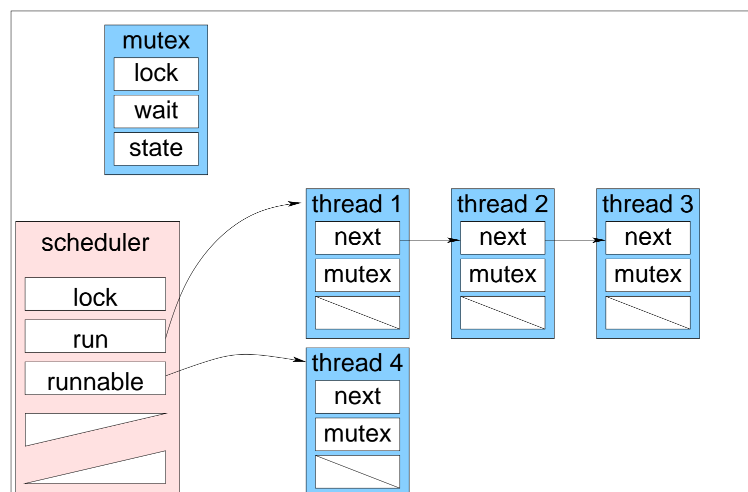
Ne permet pas de :

1. démarrer une tâche à une date donnée ;
2. relancer périodiquement une tâche ;
3. détecter un dépassement d'échéance ;
4. *raz*er une tâche dépassant l'échéance.

```
sauve état initial
loop
   restore état initial ?
   attend début de période
   tâche()
   begin
      action
   end
end loop
```

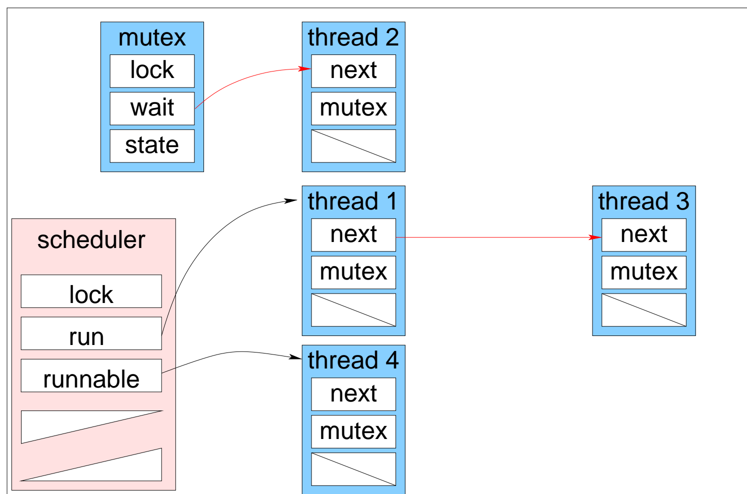

Exemple du mutex : principe

La tâche 1 acquière le mutex



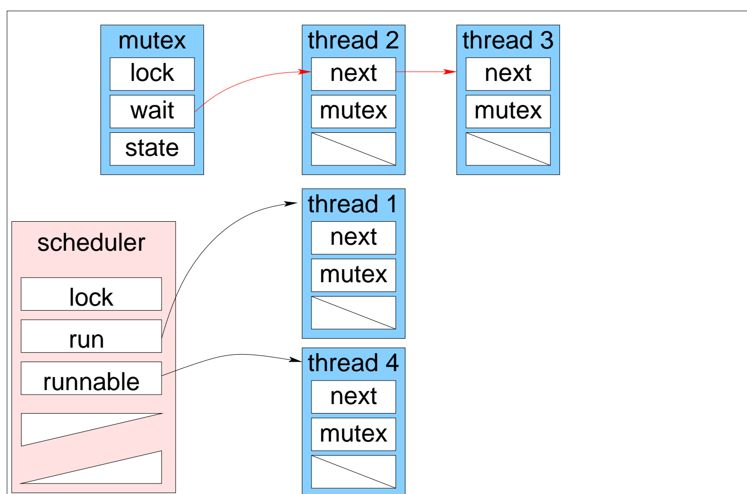
Exemple du mutex : principe

La tâche 2 tente d'obtenir le mutex



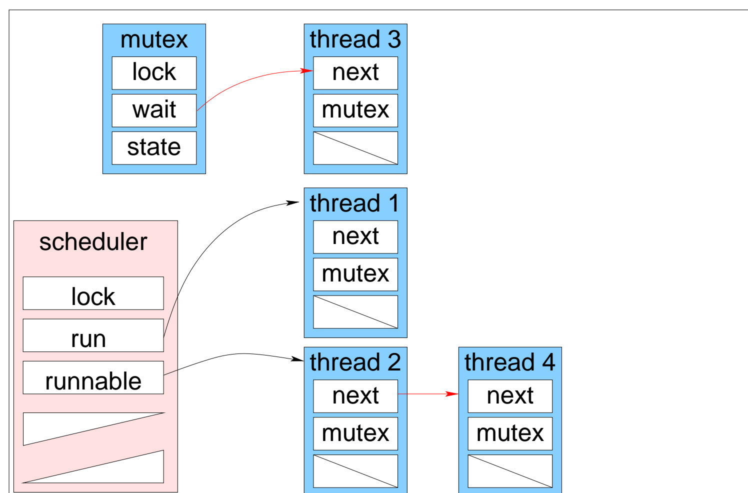
Exemple du mutex : principe

La tâche 3 tente d'obtenir le mutex



Exemple du mutex : principe

La tâche 1 relâche le mutex



Exemple du mutex : le *lock*

```
int _pthread_mutex_lock(pthread_mutex_t *mutex)
{
pthread_t self;
   _pthread_spin_lock(mutex->sem);
   self = _pthread_self();
   while (mutex->mutex == MUTEX_LOCKED) {
      _pthread_spin_lock(scheduler->sem);
      self->mutex = mutex->wait;
      mutex->wait = self;
      _pthread_spin_unlock(mutex->sem);
      enqueue(self, PS_WAIT);
      _pthread_commute(self);
      _pthread_spin_unlock(scheduler->sem);
      _pthread_spin_lock(mutex->sem);
   }
   mutex->mutex = MUTEX_LOCKED;
   _pthread_spin_unlock(mutex->sem);
   return 0;
}
```

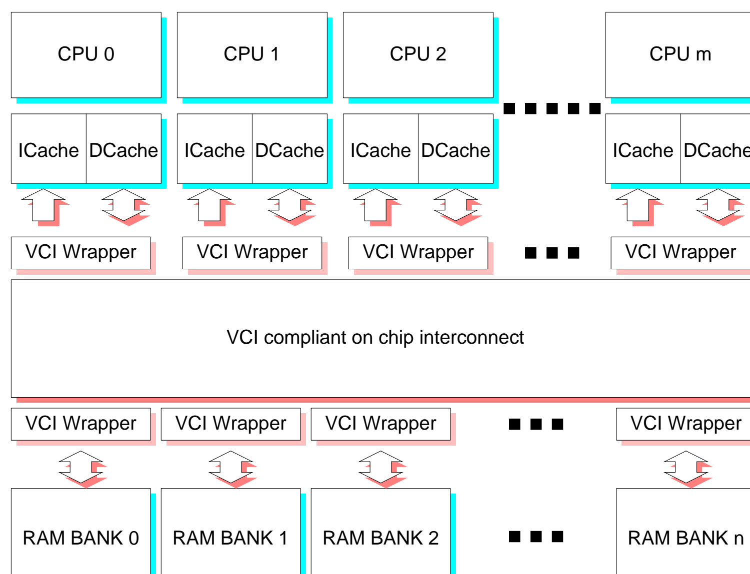
Exemple du mutex : le *unlock*

```
int _pthread_mutex_unlock(pthread_mutex_t *mutex)
{
pthread_t thread;
   _pthread_spin_lock(mutex->sem);
   if ((thread = mutex->wait) != NULL) {
      _pthread_spin_lock(scheduler->sem);
      mutex->wait = thread->mutex;
      enqueue(remove(thread), PS_RUNNABLE);
      scheduler->wakeup--;
      _pthread_spin_unlock(scheduler->sem);
   }
   mutex->mutex = MUTEX_UNLOCKED;
   _pthread_spin_unlock(mutex->sem);
}
```


Allocation/Placement mémoire

Architecture adaptée :

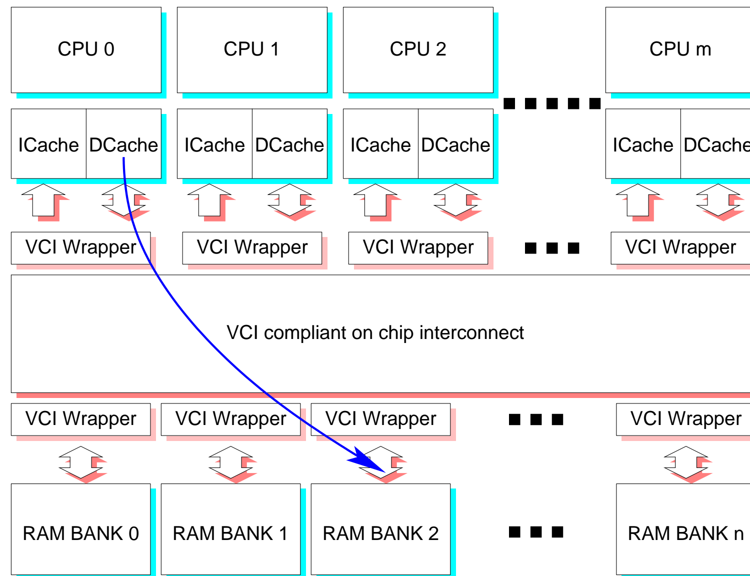
- requêtes simultanées possibles sur l'interconnexion ;
- mémoire logiquement unifiée physiquement distribuée.



Allocation/Placement mémoire

Architecture adaptée :

- requêtes simultanées possibles sur l'interconnexion ;
- mémoire logiquement unifiée physiquement distribuée.

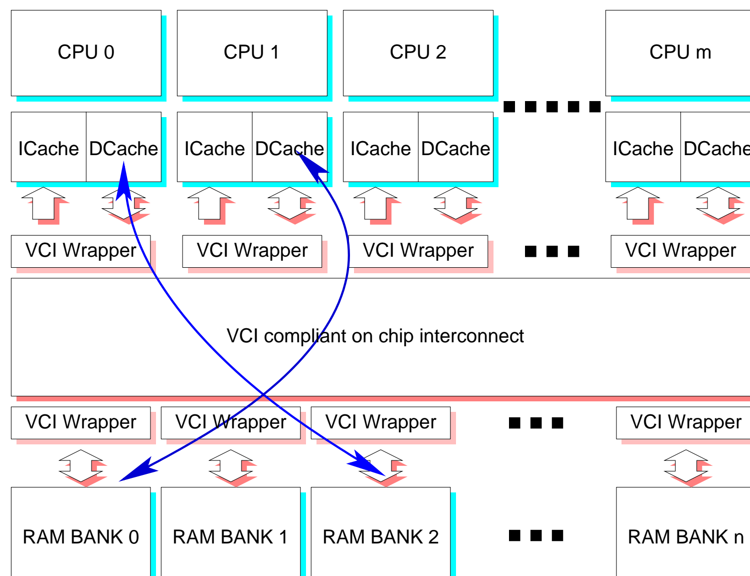


École thématique, Roscoff *1 avril 2003*

Allocation/Placement mémoire

Architecture adaptée :

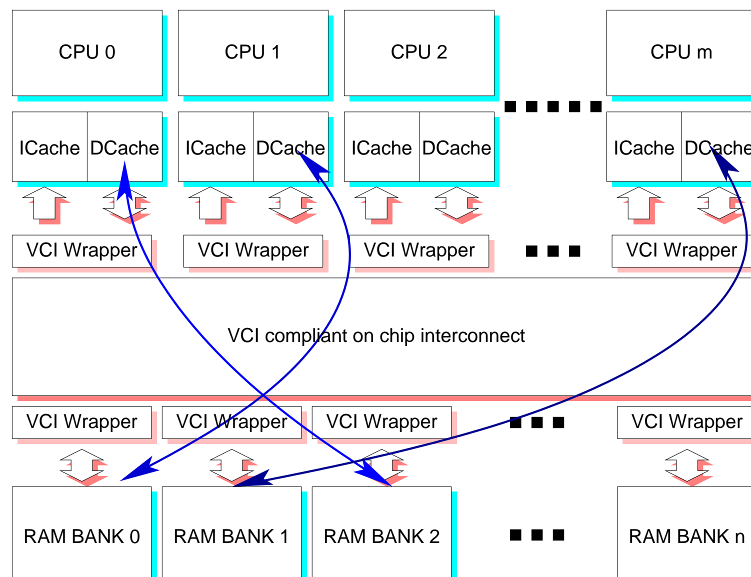
- requêtes simultanées possibles sur l'interconnexion ;
- mémoire logiquement unifiée physiquement distribuée.



Allocation/Placement mémoire

Architecture adaptée :

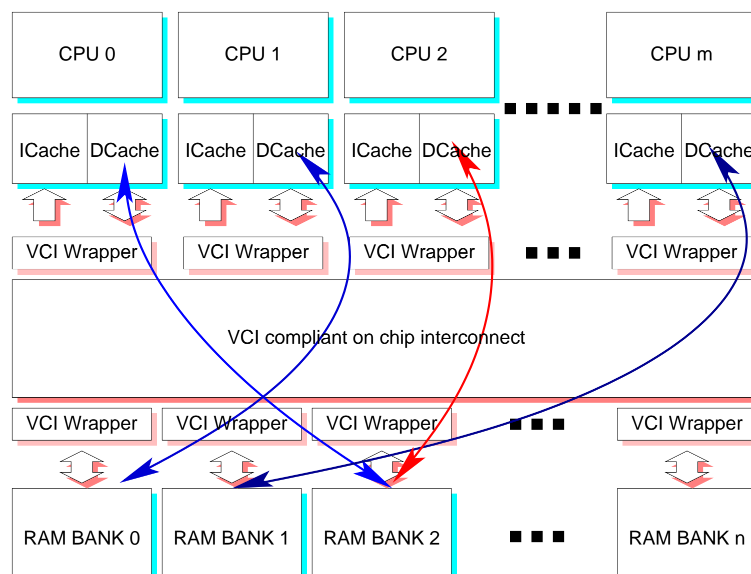
- requêtes simultanées possibles sur l'interconnexion ;
- mémoire logiquement unifiée physiquement distribuée.



Allocation/Placement mémoire

Architecture adaptée :

- requêtes simultanées possibles sur l'interconnexion ;
- mémoire logiquement unifiée physiquement distribuée.



Allocation/Placement mémoire

Influences :

- type d'ordonnancement (SMP, tâches affectées, ...) ;
- accès mémoire (UMA vs NUMA) ;
- degré de parallélisme des accès ;
- granularité des données ;
- sémantique d'accès aux données :
  1. FIFO ⇒ seulement 2 accès ;
  2. rwlocks ⇒ plusieurs lecteurs, unique écrivain ;
  3. shared memory ⇒ ???
- ...

Allocation/Placement mémoire : Implémentation

Visible du programmeur : introduction de `pmalloc`

```
void *__ma__[] = {
    n - 1,
    start-addr0, end-addr0,
    ..., ...,
    start-addrn, end-addrn
};

void *pmalloc(int pool, size_t nbytes);
```

`__ma__`
: description de l'architecture mémoire du système ;

`pool`
: index *logique* d'une zone de mémoire
fonctions ne connaissent pas le nombre de bancs *a priori*.

Allocation/Placement mémoire : Implémentation

Non *thread-safe* :

```
pthread_spin_lock(sem);
x = pmalloc(pool++, n);
pthread_spin_unlock(sem);
```

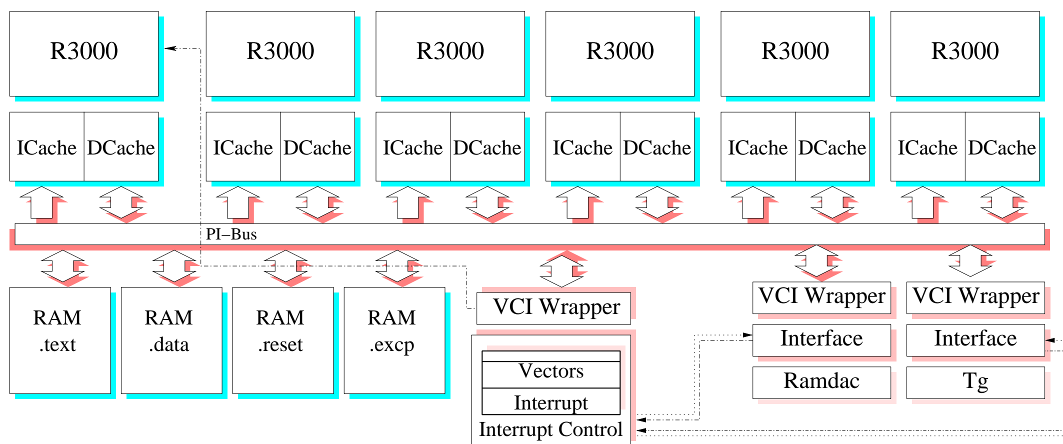
Utilisé dans les fonctions de bibliothèques :

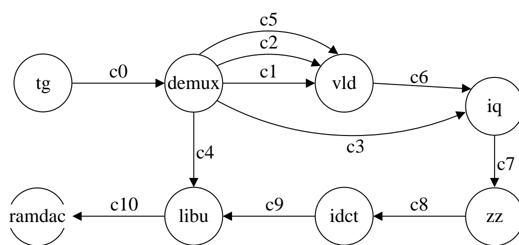
`pthread_create`
: alllocation du *thread* et de sa pile dans le même pool
  nouveau *pool* par nouveau *thread* ;

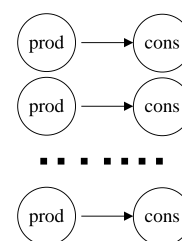
`ChannelInit`
: allocation du canal et de ses attributs dans un pool.

Difficile de spécifier le pool car toute fonction allouant de la mémoire doit posséder ce paramètre

Plateforme expérimentale







Types de noyaux implantés

- Symmetric Multiprocessor. n procs, 1 sched, migration de tâches
- Centralized Non SMP. n procs, 1 sched, pas de migration
- Distributed Non SMP. n procs, n scheds, pas de migration

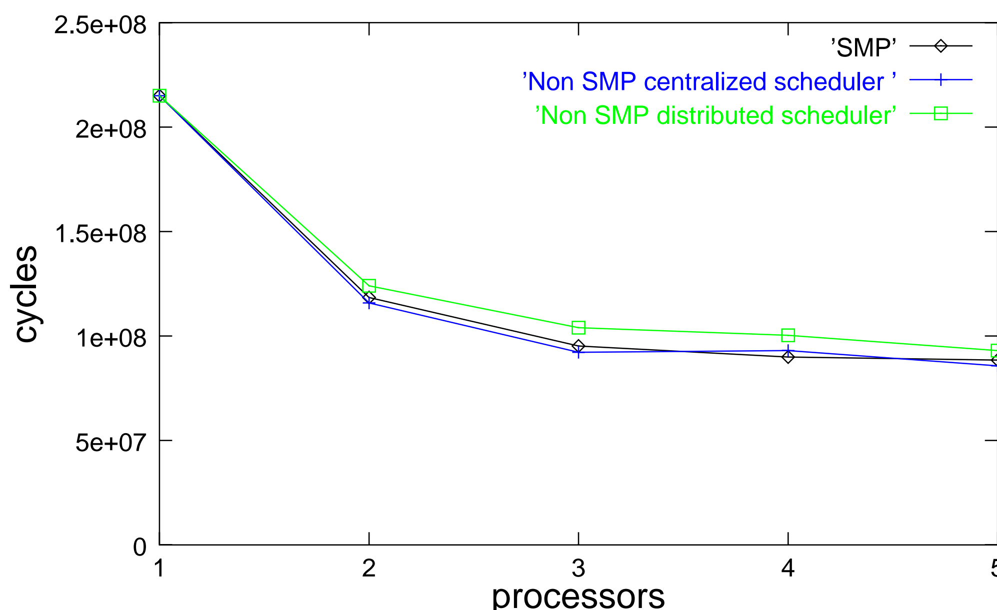
Organisation	SMP	NON_SMP_CS	NON_SMP_DS
Taille du code	7500	9500	10200 + 80 $\times n$

Temps d'exécution de quelques primitives du noyau

Number of CPUs	1	2	4	6
Context Switch	172	187	263	351
Mutex Lock (acquis)	36	56	61	74
Mutex Unlock	30	30	31	34
Mutex Lock (suspendu)	117	123	258	366
Mutex Unlock (réveil un thread)	-	191	198	218
Thread Creation	667	738	823	1085

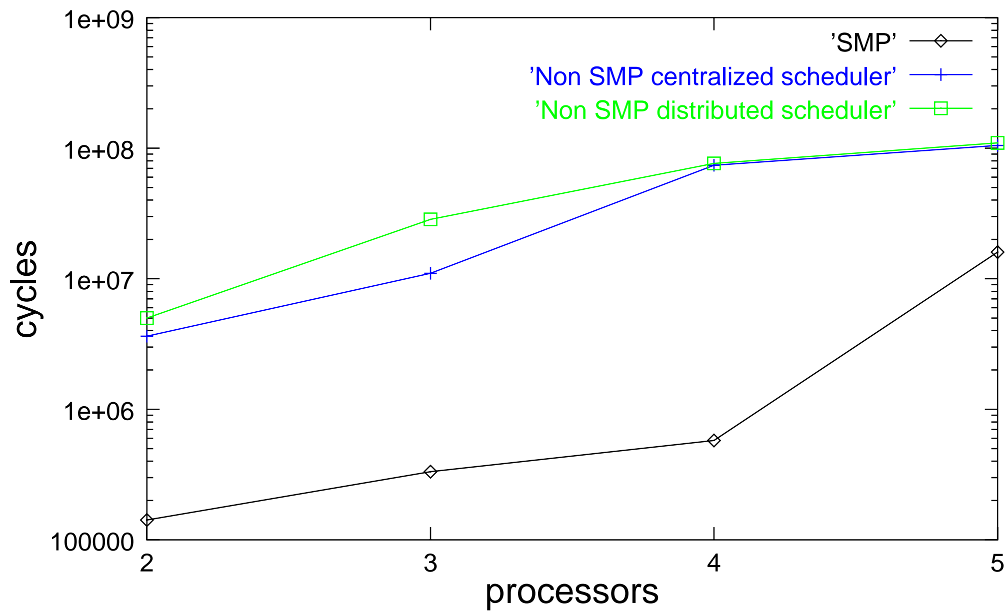
Résultats

Décodeur Multi-JPEG : temps de décodage



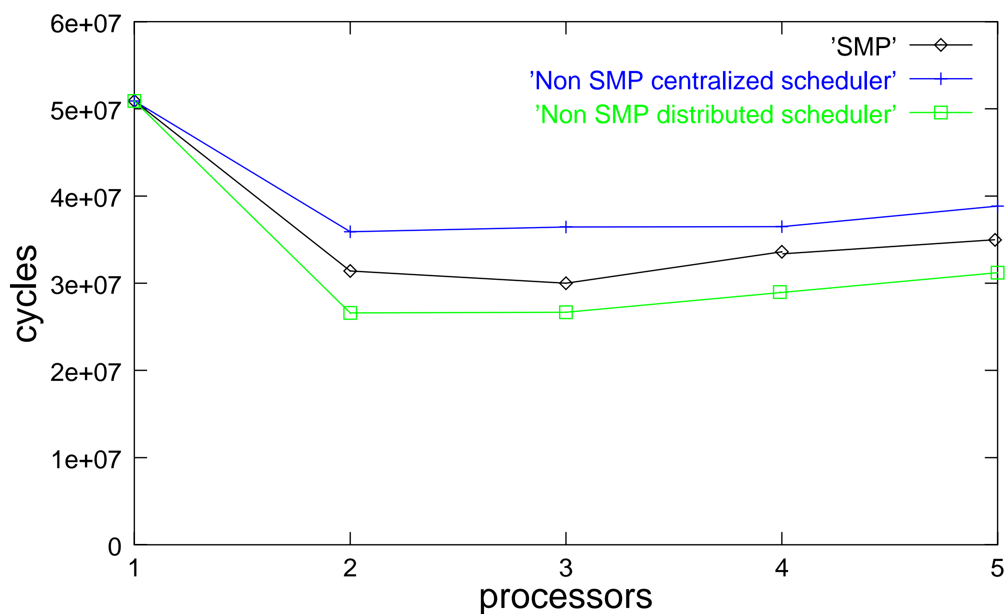
Résultats

Décodeur Multi-JPEG: cycles oisifs cummulés



Résultats

Application COMM : temps d'exécution



Conclusion

- SoC $\neq$ GPC $\Rightarrow$ SoC multiprocesseur $\neq$ multiprocesseur

Noyau POSIX :

- Simple, généraliste et relativement minimal
- Multiprocesseur
- Stratégies dépendantes de l'application

A résoudre :

- Abstraction du matériel
- Prise en compte de choix architecturaux/systèmes :
  - Scratch pad
  - NUMA
  - Optimisation des communications système