Combien de temps met le fût du canon à se refroidir ?

(ou comment mesurer le temps d'un calcul?)

Bernard Goossens

ELIAUS/DALI

Université de Perpignan

Archi'09 Pleumeur-Bodou, 30 mars au 3 avril 2009

Plan du cours

- Les compteurs matériels (RTC, HPET, APIC, TSC, PMC).
- Les dates (gettimeofday(), gmtime(), ctime()).
- La comptabilité du système (HZ, jiffy, tick, utime, stime, times()).
- Les temps d'exécution avec TSC (RDTSC, CPUID, RDTSCP, comparer sadd() à vadd()).
- Les temps d'exécution avec CTR1 (RDMSR, module noyau, comparer sadd() à vadd()).
- Conclusion.

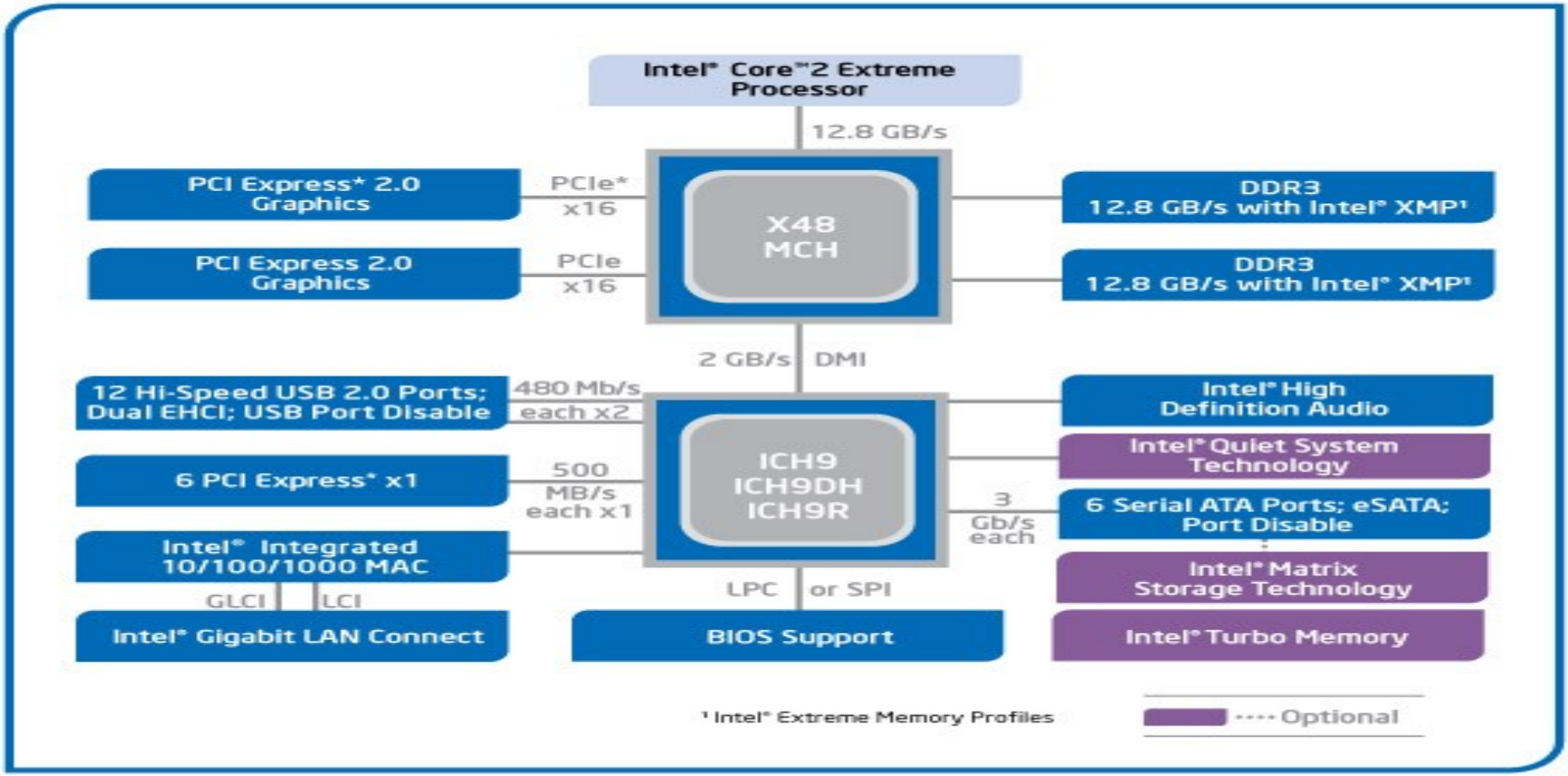
Les organes matériels pour gérer le temps en machine

- L'horloge temps réel (**RTC**): la seule référence conservée quand la machine est éteinte, période de 100000 cycles CPU.
- Le compteur de temps global (**HPET**): référence externe et commune à tous les cœurs, période de 200 cycles CPU.
- Le compteur de temps local (**APIC**): référence interne et propre à un cœur, période de quelques cycles CPU (un cycle bus FSB).
- Le compteur de cycles global (**TSC**): référence interne commune à tous les cœurs, période d'un cycle bus FSB.
- Le compteur de cycle local (**PMC/CTR1**): référence interne propre à un cœur, période d'un cycle CPU (durée variable).

L'horloge temps réel (RTC)

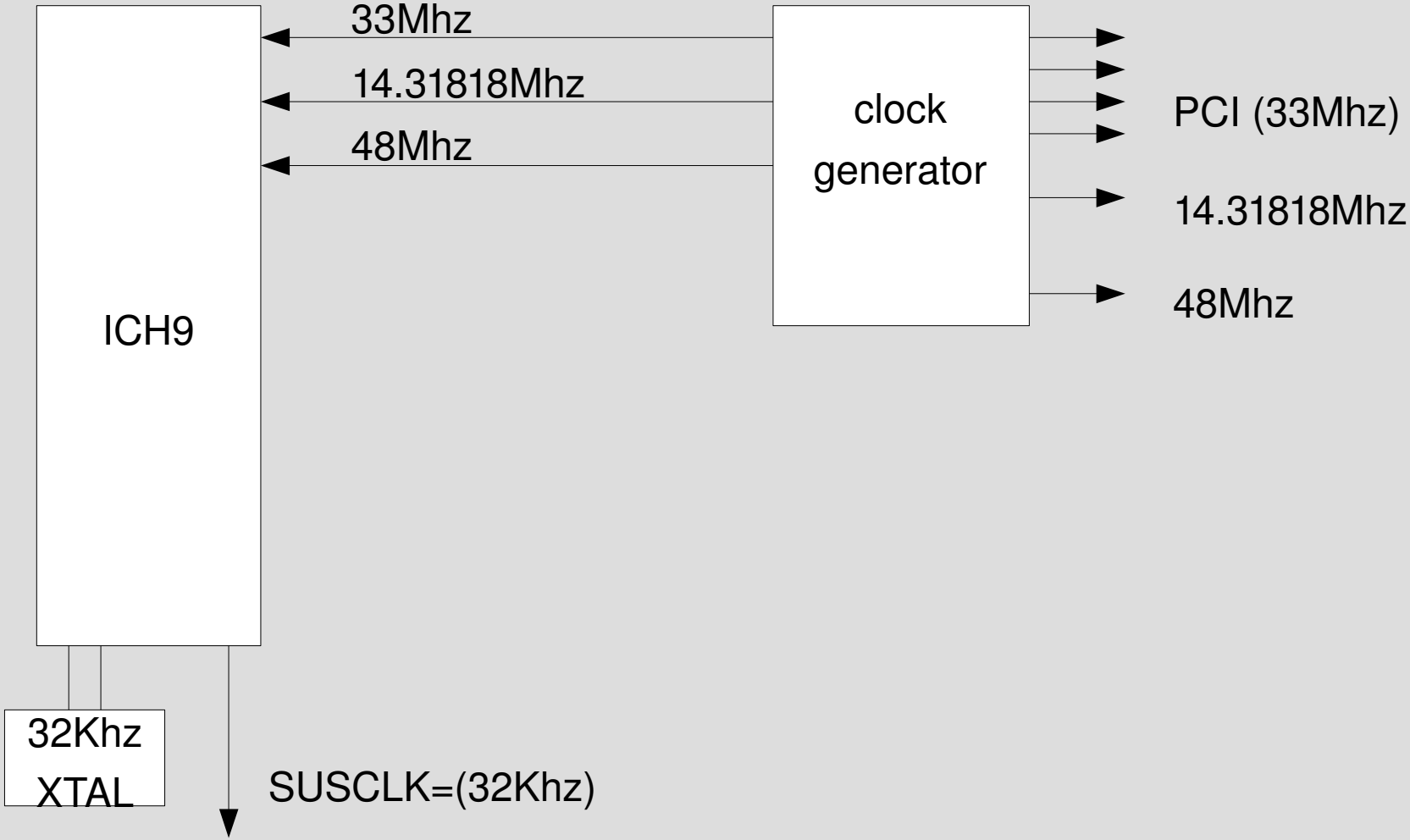
- RTC ou horloge CMOS. MC146818A intégré au ICH/SouthBridge.
- Maintient une date aa/mm/jj/hh/mm/ss dans 256 octets de RAM.
- Quartz 32,768 Khz (période 30,52µs).
- L'horloge et la mémoire sont alimentées par une batterie longue durée (RTC fonctionne même machine éteinte).
- Faible précision de la représentation (1s).
- Faible précision de la mesure (erreur de +/-13 min/an, soit +/-2 s/j).

Position d'ICH



Intel® X48 Express Chipset Block Diagram

Position de RTC



Fonctionnement de RTC

- A chaque début de cycle de l'horloge RTC (période de 30,52µs), un code interne à ICH met à jour les compteurs de date.
- RTC n'est utilisée par linux que pour initialiser la date système.
- Au démarrage, on lit RTC (*get_cmos_time()*). On initialise la variable système *xtime* de type *timespec* (*xtime.tv_sec* pour les secondes et *xtime.tv_nsec* pour les nano-secondes).
- *xtime.tv_sec* reçoit le nombre de secondes séparant la date lue dans RTC de l'*Epoch* (01/01/1970, 0h00 UTC).
- *xtime.tv_nsec* est initialisé de façon à synchroniser le changement de seconde avec RTC (attente de RTC++).

Le compteur de temps HPET

- High Performance Event Timer. Incorporé à ICH.
- Cadencé par un quartz à 14,31818Mhz (69,8413ns).
- Il contient un compteur principal qui s'incrémente sans cesse et 4 limites de déclenchement d'interruption. Linux utilise une limite initialisée pour produire une interruption de période *HZ* (limite=14316 pour *HZ*=1ms). Quand l'interruption arrive, le HPET augmente automatiquement la limite: limite+=14316 (HPET conserve la limite programmée initialement par le système).
- L'horloge et les compteurs sont alimentés par le secteur (ne fonctionne pas machine éteinte).
- Linux définit la constante noyau *HZ* qui est l'intervalle de temps entre deux interruptions de HPET.

HZ et Jiffy

- A la configuration du noyau, on fixe *HZ* à 100, 250 ou 1000.
- La durée de l'intervalle entre deux interruptions est environ de 1/*HZ* s (10ms pour *HZ*=100, 4ms pour *HZ*=250 et 1ms pour *HZ*=1000).
- La durée exacte est une valeur approchée (quand HZ=1000, 999848ns pour une période fixée à 14316 cycles HPET).
- Le *jiffy* est l'instant de durée *HZ*.
- La variable système *jiffies* tient à jour le nombre d'interruptions HPET depuis le démarrage du système (initialisé à 0xfffb6c20).

Les compteurs de temps locaux (APIC timer)

- LAPIC (Local Advanced Programmable Interrupt Controllers).
- Contrôleur d'interruption basé sur le 8259.
- Un contrôleur par cœur et un compteur de temps par contrôleur.
- Cadencé à la fréquence du bus système (FSB) divisé par 1, 2, 4, 8, 16, 32, 64 ou 128.
- Un compteur est initialisé puis décrémenté à chaque cycle. Le cœur est interrompu quand le compteur passe à 0.
- Au démarrage, on calibre le compteur (*calibrate_APIC_clock()*) pour une interruption de période *HZ* (*setup_APIC_timer()*).

L'interruption du HPET

- Elle correspond à 1 *jiffy* (ou *tick* ou *HZ*).
- Elle met à jour la variable système *jiffies* (on calcule le nombre de *ticks* depuis la dernière mise à jour en confrontant le compteur HPET actuel lu par le système au compteur précédent, lu lors de l'interruption servie précédemment: (*nticks = (actuel-précédent)/14316; jiffies += nticks*).
- Elle met à jour la variable système *xtime* (ajouter *nticks*NSECS_PER_TICK* (ns par *tick*)).
- Elle synchronise l'horloge système avec une horloge externe par *ntpd* toutes les 11 minutes (*set_rtc_mmss()*) en réinitialisant RTC.

Deux objectifs, trois outils

- Le système doit fournir des dates, par exemple pour assurer un ordonnancement chronologique d'événements possiblement très proches.
- Le système doit fournir des durées, par exemple pour mesurer le temps d'une action.
- Le matériel et le système fournissent une date à la seconde près grâce à RTC et *ntpd*.
- Le matériel et le système fournissent des durées à la milli-seconde près grâce au HPET et à l'interruption de période *HZ*.
- Le matériel et le système fournissent une précision des dates et des durées améliorée grâce à la lecture du compteur HPET (précision maximum de 69,8413ns).

Les dates: l'appel système *gettimeofday()*

- Il retourne, dans une structure de type *timeval*, le nombre de secondes (champ *tv_sec*) et le nombre de micro-secondes (champ *tv_usec*) écoulées depuis *Epoch*.
- Il lit la date système *xtime* établie lors du dernier *tick*, augmentée du nombre de nano-secondes écoulées depuis, déduites de la valeur actuelle du HPET.
- Il s'agit d'un temps réel, incluant tous les temps parasites.
- La précision maximale des dates est celle du HPET, soit 69,8413ns (il n'existe pas de date entre *d1* et *d2* quand *d2-d1=69,8413ns*).
- La date *xtime* établie à partir de RTC, de *ntpd* et de HPET est elle-même incertaine, ce qui réduit la précision des dates à la micro-seconde.

Mesure de temps réel écoulé

- En encadrant un calcul par deux appels à *gettimeofday()*, on mesure le temps réel de calcul d'une portion de code (CPU, kernel et processus concurrents, donc E/S).
- La finesse de la mesure est la micro-seconde (3 ordres de grandeur par rapport au cycle CPU).
- L'appel système *gettimeofday()* garantit que la succession des valeurs retournées est une suite croissante (le temps ne revient pas en arrière).
- L'appel système *adjtimex()* ajuste la variable système *xtime* d'un peu plus ou un peu moins d'1ms à chaque *tick* pour compenser une dérive des dates machine et *ntpd*. Ainsi, le temps évolue de façon uniforme, sans saut brutal vers la micro-seconde suivante.

Exemple avec *gettimeofday()*

```
#include <sys/time.h>
#define REP 4096
struct timeval t1,t2;
main(){
int i;
...
//pour un temps significatif (10 à 100 µs), exécuter 100000 i
//chaque exécution de sadd = 250 i => REP=4096
gettimeofday(&t1,NULL);
for (i=0;i<REP;i++) saddb();
gettimeofday(&t2,NULL);
printf("run time for sadd: %d\n", t2.tv_usec-t1.tv_usec);
}
```

run time for sadd: 225

Dater un événement

- La fonction POSIX *gmtime()* convertit le nombre de secondes écoulées depuis *Epoch* retourné par *gettimeofday()* en une date composite (*aaa/mm/jj/hh/mm/ss*) formée d'*int* (*aaa* représente le nombre d'années depuis 1900. Les mois sont numérotés de 0 à 11).
- Pour obtenir une date de précision inférieure à la seconde, on peut la compléter avec le champ *tv_usec* du résultat de *gettimeofday()*.
- Rappelons que *tv_usec* exprime des micro-secondes. La précision de la date lue dans *xtime* est la nano-seconde (*xtime.tv_nsec*) et est ramenée à la micro-seconde par *gettimeofday()* .
- La fonction *ctime()* convertit un nombre de secondes écoulées depuis *Epoch* en une date ASCII ("Mon Mar 30 09:25:33 2009").

La comptabilité des temps par processus

- L'interruption du compteur APIC de période *HZ* comptabilise le temps de chaque processus.
- Les threads d'un même processus cumulent leurs temps.
- A chaque interruption, un compteur de *tick*++.
- Si le thread interrompu était en mode *user*, le compteur *tms_utime* ++. S'il était en mode *kernel, tms_stime*++.
- Si plusieurs threads travaillent en parallèle pour un même processus sur plusieurs cœurs, *stime* et *utime* sont les temps cumulés de tous les threads, dont la somme peut être supérieure à la durée réelle.
- L'interruption décrémente le compteur de *ticks* alloués par l'ordonnanceur au thread interrompu, provoquant un changement de contexte s'il est nul.

L'appel système *times()*

- Il donne accès aux compteurs de temps d'un processus.
- Il retourne dans une variable de type *struct tms* le temps passé par le processus appelant en mode *user* (*tms_utime*), en mode *kernel* (*tms_stime*) et les temps *user* et *system* des processus fils terminés non zombis (*tms_cutime* et *tms_cstime*).
- Le temps mesuré est un nombre de *ticks*. La constante système *sysconf(_SC_CLK_TCK)* donne le nombre de *ticks* par seconde (*HZ*).
- La précision de la mesure est de l'ordre de la milli-seconde (pour *HZ*=1000).
- Le temps mesuré exclus les temps où le processus est endormi. Il inclus (dans *tms_stime*) les temps des interruptions au profit d'autres processus.

Mesure de temps effectif d'exécution

- En encadrant un calcul par deux appels à *times()*, on mesure le temps effectif de calcul d'une portion de code (temps passé en mode *user*, en mode *kernel*).
- La finesse de la mesure est de l'ordre de la milli-seconde (six ordres de grandeur par rapport au cycle du processeur).
- Soit on mesure un temps réel moyennement fin (micro-seconde) avec *gettimeofday()*, soit on mesure un temps effectif grossier (milli-seconde) avec *times()*.
- Pour *gettimeofday()*, il faut une mesure de 10^4 à 10^5 instructions exécutées. Pour *times()*, il en faut entre 10^7 et 10^9.

Mesure de temps effectif d'exécution

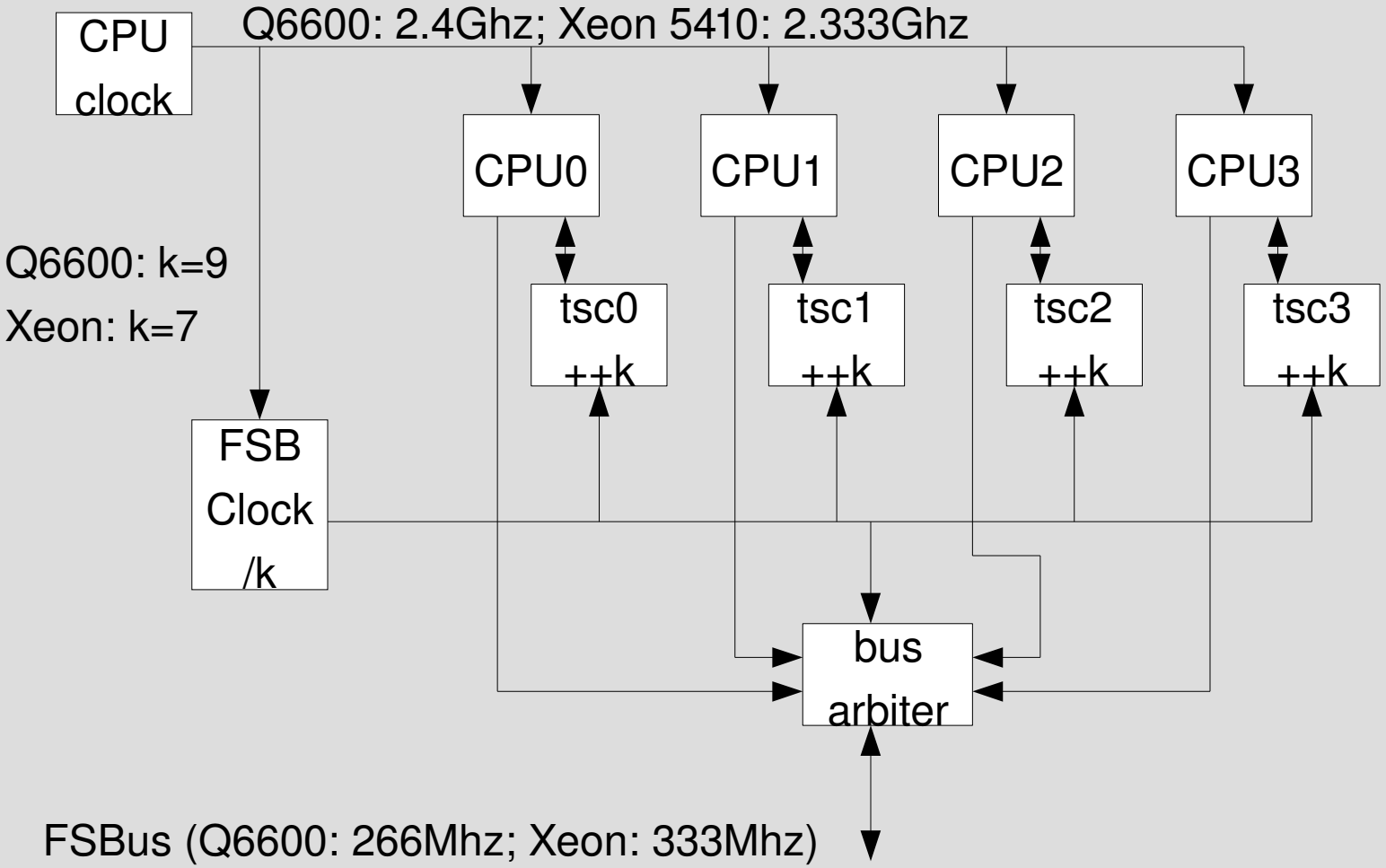
```
#include <sys/times.h>
#define REP (1<<20)
struct tms t1,t2;
main(){
int i;
...
//pour un temps significatif (10 à 100 ms), exécuter 10000000 i
//chaque exécution de sadd = 250 i => REP=2^20
times(&t1);
for (i=0;i<REP;i++) saddb();
times(&t2);
printf("run time for sadd: %d\n", t2.tms_utime-t1.tms_utime);
}
```

run time for sadd: 35

Le *Time Stamp Counter* (TSC)

- TSC est un compteur 64 bits comptant sans cesse depuis le *reset* du processeur. Il compte des cycles bus FSB convertis en cycles CPU tant que l'horloge du bus est active (elle peut-être arrêtée en mode *deep sleep* du processeur, mais continue quand un cœur est en HALT). Cette horloge bat à une période constante.
- Il y a au moins un TSC par cœur, voire un par processeur logique (Core i7?). Les TSC ne sont pas toujours identiques (on peut en désactiver un temporairement).
- Le compteur TSC peut être lu avec l'instruction RDTSC.
- Dans les micro-architectures Nehalem (Core i7), il peut aussi être lu avec l'instruction machine RDTSCP, qui retourne en plus du compteur, un identifiant du cœur lecteur (on peut ainsi constater une migration).

Micro-architecture des *Time Stamp Counters*



RDTSC n'est pas sérialisante

- La mesure de temps doit être effectuée avec précaution pour rester précise car l'instruction RDTSC n'est pas sérialisante.
- Cela a pour conséquence qu'on peut malencontreusement inclure dans la mesure des instructions qui précèdent la portion à mesurer.
- On peut aussi malencontreusement exclure de la mesure des instructions faisant partie de la portion à mesurer.
- Pour éviter ces erreurs de mesure, il faut sérialiser le début et la fin de la mesure.
- On emploie l'instruction CPUID qui est sérialisante.

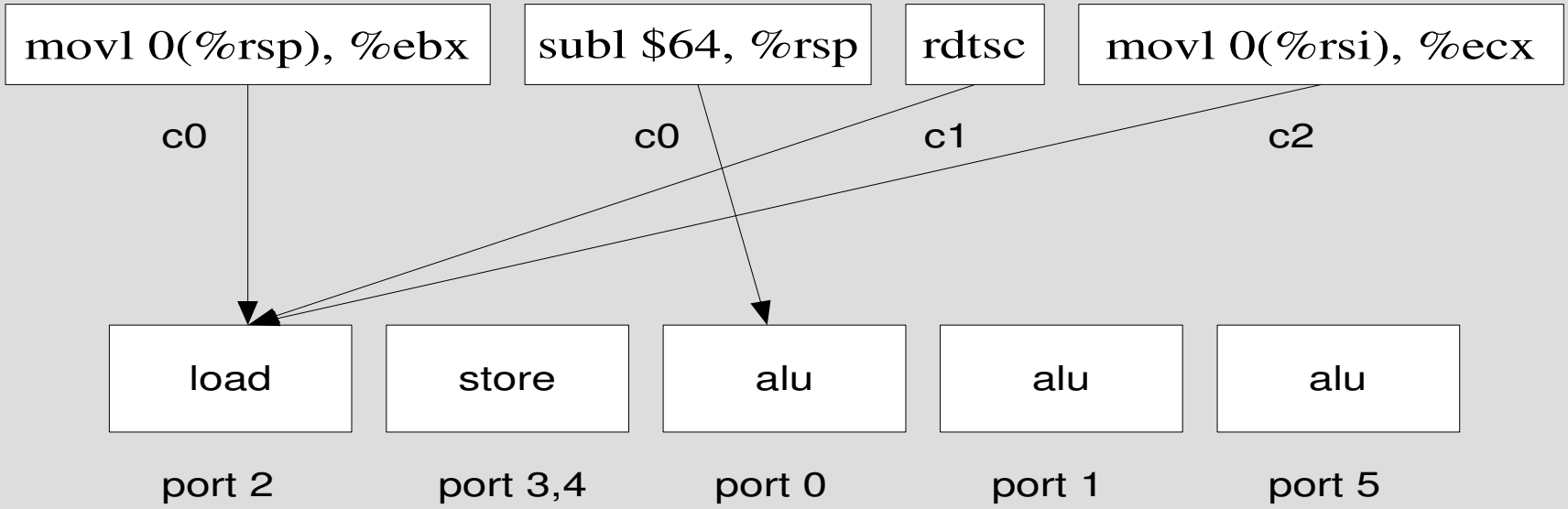
Instructions incluses et exclues de la mesure: un exemple

- Soit le morceau de programme suivant:

```
movl    0(%rsp), %ebx
movl    0(%rsi), %ecx
rdtsc
subq    $64, %rsp
```

- On suppose que la valeur de %rsi n'est pas disponible.
- L'exécution de RDTSC commence avant le second accès mémoire, après le premier (sérialisation de l'accès au port d'entrée à l'unité de chargement) et après la soustraction (port d'entrée dans l'UAL libre).
- RDTSC lit le compteur dans EDX (32b forts) et EAX (32b faibles).

Ordonnancement des instructions dans la mesure: un exemple



CPUID est sérialisante

- Une instruction sérialisante bloque l'extraction de ses suivantes tant que toutes les instructions précédentes ne se sont pas terminées (y compris les accès mémoire).
- Ensuite, l'instruction sérialisante s'exécute avant ses suivantes.
- L'instruction CPUID est sérialisante.
- L'encadrement CPUID RDTSC ... CPUID RDTSC permet de mesurer la longueur de la portion de code à l'intérieur.

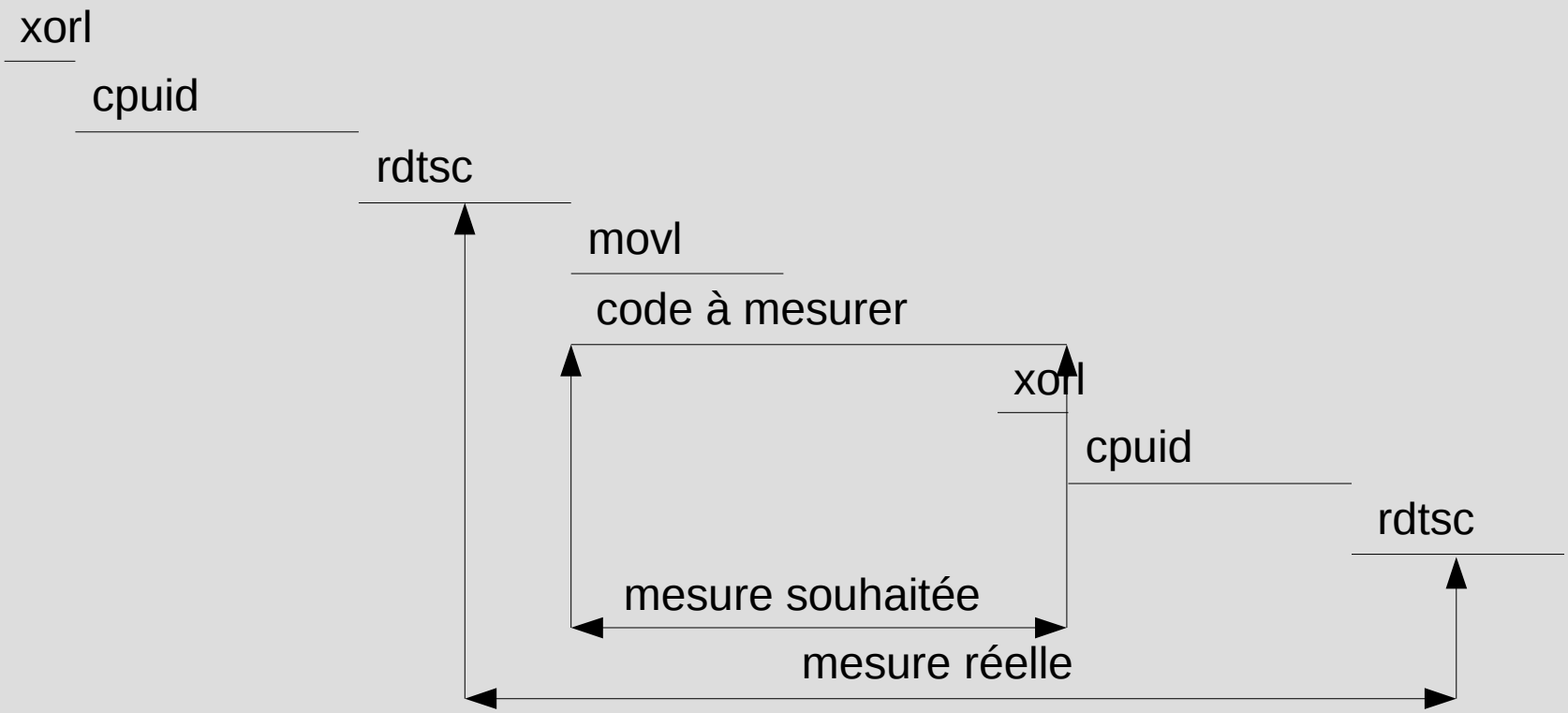
Sérialisation avec CPUID

- Voici un code de lecture de TSC sérialisé:

```
xorl        %eax,%eax
cpuid
rdtsc
movl        %eax, time
```

- RDTSC démarre après CPUID.
- RDTSC effectue sa lecture en gros au milieu de son exécution (traversée du cœur depuis la station d'attente jusqu'au TSC, puis retour de la valeur lue de TSC jusqu'au banc de registres).

Mesure recherchée vs. mesure réelle



Mesure réelle et mesure souhaitée

- La mesure réelle va de la première lecture de TSC (en gros au milieu de RDTSC) jusqu'à la seconde lecture de TSC (milieu du second RDTSC).
- mesure souhaitée = mesure réelle - cpuid - rdtsc.
- De façon plus fine, on remarque que l'exécution du code à mesurer se fait en concurrence avec MOVL et XORL. Cela peut occasionner un léger parasitage de la mesure si la portion de code partage des ressources avec ces instructions (elle est alors retardée par l'attente de la disponibilité des ressources).

Mesure de temps avec RDTSC et CPUID

- Pour mesurer en cycles bus la longueur d'une portion de code, on l'encadre avec [CPUID, RDTSC] et [CPUID, RDTSC]. (utilisables en mode *user*).
- Il convient de répéter la mesure plusieurs fois pour filtrer les anomalies (longueur<0, parasite tel un changement de contexte, nombre minimal de cycles bus).
- La mesure est à un cycle bus près (cela dépend dynamiquement de la synchronisation du début et de la fin de l'exécution du code à mesurer avec les frontières de cycles bus).
- De toute mesure on doit défalquer *cpuid* + *rdtsc* soit 231 sur un Xeon 5410 et 234 sur un Q6600.

RDTSCP est sérialisante

- Sur une micro-architecture Core i7, on emploie RDTSCP.
- Cela simplifie l'encadrement de la mesure (on encadre par [RDTSCP] et [RDTSCP]).
- Les instructions RDTSC et RDTSCP sont utilisables en mode *user*.
- Les instructions RDTSC et RDTSCP prennent plusieurs dizaines de cycles (elles traversent le cœur pour atteindre la partie FSB du composant; elles sont réalisées avec un micro-code de plusieurs micro-opérations). (31 cycles pour Core 2; vérif. de EFLAGS, accès au compteur, rangement dans EAX puis EDX, vérification d'exception)
- L'instruction CPUID prend plusieurs centaines de cycles. (198 cycles pour Core 2).

Fichier *tsccount.h* pour les mesures

```
#define RDTSC(){                                                    \
   asm volatile ("rdtsc" : : : "eax",  "edx");                      \
}
#define SERIALIZE(){                                                \
   asm volatile ("xorl %%eax,%%eax" : : : "eax");                   \
   asm volatile ("cpuid" : : : "eax", "ebx", "ecx", "edx" );        \
}
#define START(T){//initialize T=0 before START(T)                   \
   SERIALIZE()                                                      \
   RDTSC()                                                          \
   asm volatile ("subl %%eax, %[t]" : [t] "=m" (T) : : "eax");      \
}
#define STOP(T){                                                    \
   SERIALIZE()                                                      \
   RDTSC()                                                          \
   asm volatile ("addl %%eax, %[t]" : [t] "=m" (T) : : "eax");      \
}
```

Exemple: mesurer la latence de CPUID

```
#include "tsccount.h"
latence(){
  timecpuid2=0;
  START(timecpuid2);
  SERIALIZE(); SERIALIZE();
  STOP(timecpuid2);
  printf("time for 2 cpuid=%d\n",timecpuid2);
  timecpuid3=0;
  START(timecpuid3);
  SERIALIZE(); SERIALIZE(); SERIALIZE();
  STOP(timecpuid3);
  printf("time for 3 cpuid=%d\n",timecpuid3);
}
```

Résultats de mesures: CPUID sur Q6600 et Xeon 5410

- Sur Xeon 5410, 2 CPUID prennent 56/57 cycles FSB (*timecpuid2* = 623-231/630-231). 3 CPUID prennent 84/85 cycles FSB (*timecpuid3* = 819-231/826-231).
- La latence de CPUID est (*timecpuid3* - *timecpuid2*) soit 27/29 cycles FSB, soit entre 189 et 203 cycles CPU (le manuel dit environ 200 cycles).
- Sur Q6600, 2 CPUID prennent 41/42 cycles FSB (*timecpuid2* = 603-234/612-234). 3 CPUID prennent 62/63 cycles FSB (*timecpuid3* = 792-234/801-234).
- La latence de CPUID est (*timecpuid3* - *timecpuid2*) soit 20/22 cycles FSB, soit entre 180 et 198 cycles CPU (le manuel dit environ 200 cycles).

Exemple: comparer l'addition scalaire à l'addition vectorielle

```
#include tsccount.h
vaddb(){//exécuter au moins deux fois pour chauffer les caches
 vtime=0;
 START(vtime);
 for (k=0; k<16; k++)
  for (i=0;i<8;i++) z.v[i]=_builtin_ia32_paddb(x.v[i],y.v[i]);
 STOP(vtime);
 printf("vector time: %d\n", vtime);
}
saddb(){//exécuter au moins deux fois pour chauffer les caches
 stime=0;
 START(stime);
 for (k=0; k<16; k++)
  for (i=0;i<8;i++)
   for (j=0;j<8;j++) z.s[i][j]=x.s[i][j] + y.s[i][j]);
 STOP(stime);
 printf("scalar time: %d\n", stime);
}
```

Résultats de mesures: Q6600 et Xeon 5410

- Sur Xeon 5410, *saddb()* prend 290/291 cycles FSB (*stime* = 2261-231/2268-231). v*addb()* prend 21/22 cycles FSB (*vtime* = 378-231/385-231).
- Le rapport s*addb()*/*vaddb()* est de 13.
- Sur Q6600, *saddb()* prend 356/357 cycles FSB (*stime* = 3438-234/3447-234). v*addb()* prend 42/43 cycles FSB (*vtime* = 612-234/621-234).
- Le rapport s*addb()*/*vaddb()* est de 8,5.

Usages de TSC

- Le TSC sert à mesurer des temps.
- Il mesure des délais assez courts (inférieurs à 1 *tick*).
- La mesure est à un cycle bus FSB près.
- Il ne mesure pas des cycles CPU (qui sont de durées variables).
- Il ne permet pas de mesurer un CPI.
- Pour mesurer des portions de code très courtes, il est préférable d'utiliser un PMC (Performance Monitoring Counter). (par exemple, la latence d'une instruction)
- Dans le futur, TSC devrait remplacer le HPET pour établir *HZ*. (avec une synchronisation périodique de tous les TSC par le système).

Les compteurs de performance

- Des compteurs sont disséminés dans l'ensemble de la micro-architecture (lancement de micro-opération, opérateur, cache, prédicteur de saut, retrait d'instruction, horloge de cœur). Ce sont les PMC (Performance Monitoring Counters).
- Chaque PMC porte un numéro qui permet de l'adresser (et un nom symbolique référencé dans la documentation).
- Selon les micro-architectures les PMC varient. Certains sont architecturaux (existent à partir d'un certain modèle de processeur et sur tous les successeurs).

Les registres MSR

- Les micro-architectures Core 2 disposent de 3 registres MSR pour observer 3 PMC fixes et 2 registres MSR pour observer 2 PMC au choix (4 registres sur Core i7).
- Les PMC fixes observables sont:
  - CTR0 (MSR 0x309) Instruction Retired
  - CTR1 (MSR 0x30a) UnHalted Core Cycles
  - CTR2 (MSR 0x30b) UnHalted Reference Cycles
- Ces compteurs fixes peuvent être activés et désactivés en écrivant dans un registre MSR de contrôle (MSR 0x38d |= 0x333 pour activer les 3 compteurs).

Compter les cycles CPU

- Pour compter les cycles CPU avec CTR1, on commence par activer le compteur:

```
movl    $0x38d,%ecx
rdmsr
orl     $0x333,%eax
wrmsr
```

- L'instruction RDMSR lit le registre dont le numéro est dans ECX.
- L'instruction WRMSR écrit EAX dans le registre MSR de numéro ECX.
- L'instruction WRMSR est privilégiée.

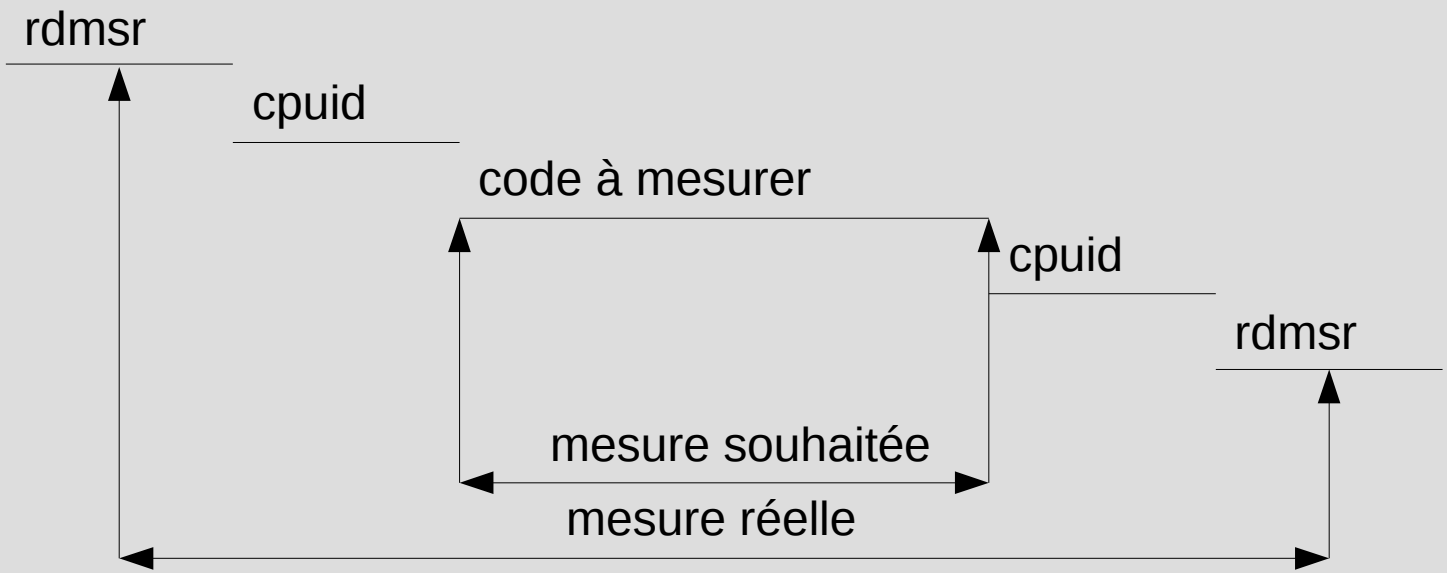
Lire le compteur CTR1

- On lit CTR1 avec:

```
movl    $0x30a,%ecx
rdmsr
```

- L'instant de la mesure est en gros au milieu de l'exécution de RDMSR.
- Pour mesurer, on lit une première fois, on sérialise avec CPUID, on insère le code à mesurer, on sérialise à nouveau avec CPUID et on lit une seconde fois. On effectue une seconde mesure avec un code vide.
- La longueur du code en cycles CPU est la différence entre la première et la seconde mesure.

Mesure avec CTR1



Les modules noyau

- L'instruction WRMSR, indispensable pour activer CTR1, est privilégiée.
- On peut ajouter dynamiquement des modules au noyau (qui s'exécutent en mode noyau et permettent l'exécution d'instructions privilégiées).

/sbin/insmod mon_module.ko

- Pour former un module, il faut compiler son source C et le lier au noyau de façon spéciale.

make -C /lib/modules/`uname -r`/build M=`pwd` modules

- On ajoute le module à un Makefile spécial contenant la ligne suivante.

obj-m += mon_module.o

Contenu d'un module

- Un module se compose d'une fonction *init_module(void)* qui est exécutée à l'insertion du module par */sbin/insmod*.
- Il comprend aussi une fonction *cleanup_module(void)* exécutée à la suppression du module par */sbin/rmmod*.
- Un module n'a pas accès directement aux fonctions d'E/S de la librairie standard. Il faut fournir une fonction *print_string(char *str)* accédant au pilote *tty* de la fenêtre courante. On complète d'une fonction *print_int(int i)* pour convertir un entier et l'afficher avec *print_string*.

Un exemple de module: mesurer CPUID (1)

```
int init_module(void){
  int i,stamp0,stamp1;

  //démarrage des compteurs
  asm volatile("movl $0x38d,%%ecx" : : : "ecx");
  asm volatile("rdmsr" : : : "ecx", "eax", "edx");
  asm volatile("orl $0x333,%%eax" : : : "eax");
  asm volatile("wrmsr" : : : "ecx", "eax", "edx");

  //chauffer les caches en mesurant plusieurs fois
  for (i=0;i<8;i++){
   ...
  }
  ...
}
```

Un exemple de module: mesurer CPUID (2)

```
int init_module(void){
 ...
 //chauffer les caches en mesurant plusieurs fois
 for (i=0;i<8;i++){
  //lire le compteur et sauver en mémoire
  asm volatile("movl $0x30a,%%ecx" : : : "ecx");
  asm volatile("rdmsr" : : : "ecx", "eax", "edx");
  asm volatile("movl %%eax, %[stamp0]" : [stamp0]
            "=m" (stamp0) : : "eax");
  //sérialiser
  asm volatile("xorl %%eax,%%eax" : : : "eax");
  asm volatile("cpuid" : : : "eax", "ebx", "ecx", "edx");
  ...
 }
 ...
}
```

Un exemple de module: mesurer CPUID (3)

```
int init_module(void){
  ...
  //chauffer les caches en mesurant plusieurs fois
  for (i=0;i<8;i++){
    ...
    //code à mesurer
    asm volatile("xorl %%eax,%%eax" : : : "eax");
    asm volatile("cpuid" : : : "eax", "ebx", "ecx", "edx");
    //sérialiser
    asm volatile("xorl %%eax,%%eax" : : : "eax");
    asm volatile("cpuid" : : : "eax", "ebx", "ecx", "edx");
    ...
  }
  ...
}
```

Un exemple de module: mesurer CPUID (4)

```
int init_module(void){
 ...
 //chauffer les caches en mesurant plusieurs fois
  for (i=0;i<8;i++){
   ...
   //lire le compteur et sauver en mémoire
   asm volatile("movl $0x30a,%%ecx" : : : "ecx");
   asm volatile("rdmsr" : : : "ecx", "eax", "edx");
   asm volatile("movl %%eax, %[stamp1]" : [stamp1]
            "=m" (stamp1) : : "eax");
 }
 ...
}
```

Un exemple de module: mesurer CPUID (5)

```
int init_module(void){
  ...
  for (i=0;i<8;i++){
    ...
  }
  //afficher le résultat
  print_string("# core cycles elapsed");
  print_int(stamp1-stamp0);
  return 0;
}
```

La fonction *print_string*

```
static void print_string(char *str){
  struct tty_struct *my_tty;
  my_tty = current->signal->tty;
  if (my_tty != NULL){
     ((my_tty->driver)->ops->write) (my_tty,str,strlen(str));
     ((my_tty->driver)->ops->write) (my_tty, "\015\012", 2);
  }
}
```

Résultat de mesure: CPUID

- Xeon 5410:
  - 0 CPUID = 607 cycles
  - 1 CPUID = 804 cycles
- CPUID = 804 – 607 = 197 cycles
- Q6600:
  - 0 CPUID = 562 cycles
  - 1 CPUID = 744 cycles
- CPUID = 744 – 562 = 182 cycles

Résultat de mesure: RDMSR

- Xeon 5410:
  - 0 RDMSR = 607 cycles
  - 1 RDMSR = 818 cycles
- RDMSR = 818 – 607 = 211 cycles
- Q6600:
  - 0 RDMSR = 562 cycles
  - 1 RDMSR = 762 cycles
- RDMSR = 762 – 562 = 200 cycles

Résultat de mesure: *sadd()*

- Xeon 5410:
  - 0 sadd: 607 cycles
  - 1 sadd: 739 cycles
- sadd: 739 – 607 = 132 cycles
- Q6600:
  - 0 sadd: 562 cycles
  - 1 sadd: 695 cycles
- sadd: 695 – 562 = 133 cycles

Résultat de mesure: *vadd()*

- Xeon 5410:
  - 0 vadd: 607 cycles
  - 1 vadd: 625 cycles
- vadd: 625 – 607 = 18 cycles
- Q6600:
  - 0 vadd: 562 cycles
  - 1 vadd: 579 cycles
- vadd: 578 – 562 = 16 cycles

Exécution pipelinée de *vadd()*

```
load x,mmx
load y,mmy
-
-
paddb mmx,mmy
store mmy,z
-
```

```
load x,mmx
load y,mmy
-
-
paddb mmx,mmy
store mmy,z
-
```

```
load x,mmx
load y,mmy
-
-
paddb mmx,mmy
store mmy,z
-
```

```
load x,mmx
load y,mmy
-
-
paddb mmx,mmy
store mmy,z
-
```

Exécution pipelinée de *vadd()*

- On exécute 8 fois la même séquence: chargement de deux vecteurs de 8 octets, addition parallèle, écriture du vecteur résultat en mémoire.
- Il n'y a qu'un port de lecture en mémoire, donc les lectures sont sérialisées.
- Chaque lecture prend 3 cycles. Chaque écriture prend 2 cycles, Chaque addition prend 1 cycle.
- Il y a une unité d'écriture en mémoire séparée de celle de lecture. On peut donc lire et écrire au même cycle.
- L'exécution de *vadd()* prend 8*2 + 5 = 21 cycles.

Exécution pipelinée de *sadd()*

```
load x,rx
load y,ry
-
-
addb rx,ry
store ry,z
-
```

```
load x,rx
load y,ry
-
-
addb rx,ry
store ry,z
-
```

```
load x,rx
load y,ry
-
-
addb rx,ry
store ry,z
-
```

```
load x,rx
load y,ry
-
-
addb rx,ry
store ry,z
-
```

```
load x,rx
load y,ry
-
-
addb rx,ry
store ry,z
-
```

Exécution pipelinée de *sadd()*

- On exécute 8 fois la même séquence: chargement de deux octets, addition, écriture de l'octet résultat en mémoire.
- Ceci est exécuté 8 fois (8 tours de boucle).
- Le contrôle de boucle s'effectue en parallèle avec le calcul (i++, cmp et jne utilisent des opérateurs distincts de ceux employés par load, add et store).
- L'exécution de s*add()* prend 64*2 + 5 = 133 cycles.
- La mesure observée (132/18 sur Xeon et 133/16 sur Q6600 au lieu de 133/21 réel) permet de constater le rapport 8 des deux temps d'exécution *sadd()*/*vadd()*.
- Les imprécisions sont de quelques cycles.

Usages de CTR1

- Le compteur CTR1 sert à mesurer des cycles CPU, qui peuvent avoir une durée variable. On ne peut donc pas en déduire de temps.
- Il permet de mesurer un CPI.
- Il peut être utilisé pour mesurer la latence en cycle d'une instruction.
- Son emploi par les modules est assez peu commode (par exemple, il faut intégrer le code à mesurer dans le module).

Conclusion

- Dates, estampilles: *gettimeofday()*. La précision de la date est la micro-seconde.
- Temps réel: *gettimeofday()*. La finesse de la mesure est la micro-seconde.
- Temps *user* et/ou *kernel*: *times()*. La finesse de la mesure est la milli-seconde.
- Temps *user* précis: TSC. La finesse de la mesure est la nano-seconde.
- Cycles *user*, CPI: CTR1. La finesse de la mesure est le cycle CPU, de durée variable.