

Informatique embarquée

Mesure du temps et de la charge CPU

Philippe.Plasson@obspm.fr

Mesure du temps et de la charge CPU

1. Introduction
2. Techniques de mesure du temps et évaluation des temps d'exécution
3. Etablissement d'un budget CPU

Mesure du temps et de la charge CPU

1. Introduction
2. Techniques de mesure du temps et évaluation des temps d'exécution
3. Etablissement d'un budget CPU

Introduction

- Dans le domaine des logiciels embarqués, il est crucial de maîtriser, dès la phase de conception préliminaire, les temps d'exécution des différents traitements implémentés dans les applications.
 - Ressources CPU limitées
 - Contraintes temps-réel (timing et rendez-vous temporels à respecter)
- L'objectif est de pouvoir réaliser des études de dimensionnement matériel / logiciel :
 - Dimensionner correctement les ressources matérielles des cartes processeurs (puissance de traitement et ressources mémoire) qui seront nécessaires pour faire fonctionner le logiciel embarqué en fonction des spécifications du système embarqué dans lequel il s'insère.

Introduction

■ Démarche :

1. Identifier les principaux traitements qui devront être implémentés dans le logiciel embarqué ainsi que leur fréquence d'activation.
2. Prototyper en C ou C++ les principaux algorithmes candidats à une implémentation dans le logiciel embarqué.
3. Réaliser sur simulateur ou avec des cartes processeurs du commerce des mesures de temps d'exécution.
4. Etablir des budgets CPU prenant en compte les temps d'exécution mesurés des algorithmes et les fréquences d'activation.
5. Identifier les sources d'optimisation et les compromis possibles.
6. Itérer avec les architectes électroniques en charge de la conception du matériel.

Mesure du temps et de la charge CPU

1. Introduction
2. Techniques de mesure du temps et évaluation des temps d'exécution
3. Etablissement d'un budget CPU

Techniques de mesure du temps et évaluation des temps d'exécution

- Deux approches peuvent être utilisées :
 1. Utilisation d'un simulateur de la cible embarquée proposant des fonctions non-intrusives de mesure des performances
 2. Mesure sur cible matérielle ou sur simulateur en utilisant les ressources propres du matériel, c'est-à-dire les timers hardware

Mesure du temps à l'aide de fonctions de mesure de performance d'un simulateur

- Certains simulateurs de cibles embarquées offrent des fonctions de mesure de performance inspirées par la commande Linux « **perf** » qui permet de :
 - Compter le nombre d'instructions exécutées
 - Donner des statistiques sur l'utilisation du cache
 - Etc.

Mesure du temps à l'aide de fonctions de mesure de performance d'un simulateur

- Par exemple, le simulateur TSIM de processeurs LEON offre un telle commande :
 - La commande « perf reset » permet de remettre à 0 les statistiques de performance.
 - La commande « perf » permet d'afficher les statistiques courantes de performance :
 - Nombre de cycles processeur
 - Nombre d'instructions
 - Nombre moyens de cycles par instruction (CPI)
 - Nombre d'instructions par seconde (en MIPS)
 - Nombre d'opérations flottantes par seconde (en MFLOPS)
 - Taux de réussite d'accès au cache (Cache hit rate)
 - Taux d'utilisation du bus processeur (Bus AMBA)
 - Temps d'exécution en ms (Simulated time)

Mesure du temps à l'aide de fonctions de mesure de performance d'un simulateur

La commande perf du simulateur TSIM permet d'obtenir les statistiques de performance liées à l'appel de la fonction factorial(1000) (la fonction a été compilée avec l'option -O2).

La fréquence de processeur est 50 MHz,
donc 1 cycle = $1/50.000.000$ s = 20 ns
Temps d'exécution = 9101 cycles = 0.182 ms

```
#include "factorial.h"

uint32_t factorial( uint32_t n)
{
    uint32_t result = 1;
    uint32_t i;
    for (i = 1; i <= n; i++)
    {
        result = result * i;
    }
    return result;
}
```

```
Cycles           :      9101

Instructions      :      4045
Overall CPI      :      2.25

CPU performance (50.0 MHz) : 22.22 MOPS (22.22 MIPS, 0.00 MFLOPS)
Cache hit rate    :    99.9 % (inst: 99.9, data: 80.0)
AHB bandwidth utilisation :    0.0 % (inst: 0.0, data: 0.0)
Simulated time    :    0.18 ms
Processor utilisation : 100.00 %
Real-time performance :    2.60 %
Simulator performance :    0.58 MIPS
Used time (sys + user) :    0.01 s
```

Mesure du temps à l'aide de fonctions de mesure de performance d'un simulateur

Quand la fonction `factorial()` a été compilée avec l'option `-O0`, le temps d'exécution mesuré par la fonction `perf` de `factorial(1000)` est de 24259 cycles soit 0.49 ms (à comparer aux 0.18 ms obtenus avec l'option `-O2`).

```
#include "factorial.h"

uint32_t factorial( uint32_t n)
{
    uint32_t result = 1;
    uint32_t i;
    for (i = 1; i <= n; i++)
    {
        result = result * i;
    }
    return result;
}
```

```
Cycles           :      24259

Instructions      :      14074
Overall CPI       :       1.72

CPU performance (50.0 MHz) : 29.01 MOPS (29.01 MIPS, 0.00 MFLOPS)
Cache hit rate    :    99.9 % (inst: 99.9, data: 100.0)
AHB bandwidth utilisation :    0.8 % (inst: 0.0, data: 0.8)
Simulated time    :    0.49 ms
Processor utilisation : 100.00 %
Real-time performance :    4.41 %
Simulator performance :    1.28 MIPS
Used time (sys + user) :    0.01 s
```

Mesure du temps à l'aide de fonctions de mesure de performance d'un simulateur

Quand la fonction factorial() a été compilée avec l'option -O2 mais que **le cache (données et instructions) a été désactivé**, le temps d'exécution mesuré par la fonction perf de factorial(1000) est de 25266 cycles soit 0.51 ms (à comparer aux 0.18 ms obtenus quand le cache est activé). Le nombre d'instructions est le même mais le nombre de cycles a été multiplié par 2.8 !

```
#include "factorial.h"

uint32_t factorial( uint32_t n)
{
    uint32_t result = 1;
    uint32_t i;
    for (i = 1; i <= n; i++)
    {
        result = result * i;
    }
    return result;
}
```

Cycles : 25266

Instructions : 4045

Overall CPI : 6.25

CPU performance (50.0 MHz) : 8.00 MOPS (8.00 MIPS, 0.00 MFLOPS)

Cache hit rate : 0.0 % (inst: 0.0, data: 0.0)

AHB bandwidth utilisation : 1.6 % (inst: 1.6, data: 0.0)

Simulated time : 0.51 ms

Processor utilisation : 100.00 %

Real-time performance : 5.05 %

Simulator performance : 0.40 MIPS

Used time (sys + user) : 0.01 s

Le cache est
désactivé

Mesure du temps à l'aide de fonctions de mesure de performance d'un simulateur

La commande perf du simulateur TSIM permet d'obtenir les statistiques de performance liées à l'appel de la fonction square().

La fonction a été compilée avec l'option -O2, mais avec l'option -msoft-float (calculs flottants réalisés à l'aide d'une bibliothèque utilisant des calculs sur entiers).

Le nombre de cycles mesuré est 904.

```
float square(float x) {  
    float result = x * x;  
    return result;  
}
```

```
Cycles           :          904  
  
Instructions      :          235  
Overall CPI      :          3.85  
  
CPU performance (50.0 MHz) : 13.00 MOPS (13.00 MIPS, 0.00 MFLOPS)  
Cache hit rate    :    76.9 % (inst: 76.3, data: 79.3)  
AHB bandwidth utilisation :    0.1 % (inst: 0.0, data: 0.0)  
Simulated time    :    0.02 ms  
Processor utilisation : 100.00 %  
Real-time performance :    1.#J %  
Simulator performance :    0.00 MIPS  
Used time (sys + user) :    0.00 s
```

Mesure du temps à l'aide de fonctions de mesure de performance d'un simulateur

La commande perf du simulateur TSIM permet d'obtenir les statistiques de performance liées à l'appel de la fonction square().

La fonction a été compilée avec l'option -O2, mais sans l'option -msoft-float (calculs flottants réalisés à l'aide d'un FPU).

Le nombre de cycles mesuré est 136, à comparer aux 904 cycles quand le FPU n'est pas utilisé. Le compteurs de MFLOPS indique 0.37.

```
float square(float x) {  
    float result = x * x;  
    return result;  
}
```

Cycles : 136

Instructions : 45

Overall CPI : 3.02

CPU performance (50.0 MHz) : 16.54 MOPS (16.18 MIPS, **0.37 MFLOPS**)

Cache hit rate : 82.7 % (inst: 82.6, data: 71.4)

AHB bandwidth utilisation : 0.0 % (inst: 0.0, data: 0.0)

Simulated time : 0.00 ms

Processor utilisation : 100.00 %

Real-time performance : 1.#J %

Simulator performance : 0.00 MIPS

Used time (sys + user) : 0.00 s



Le FPU est utilisé

Mesure des temps d'exécution en utilisant les timers hardware du processeur

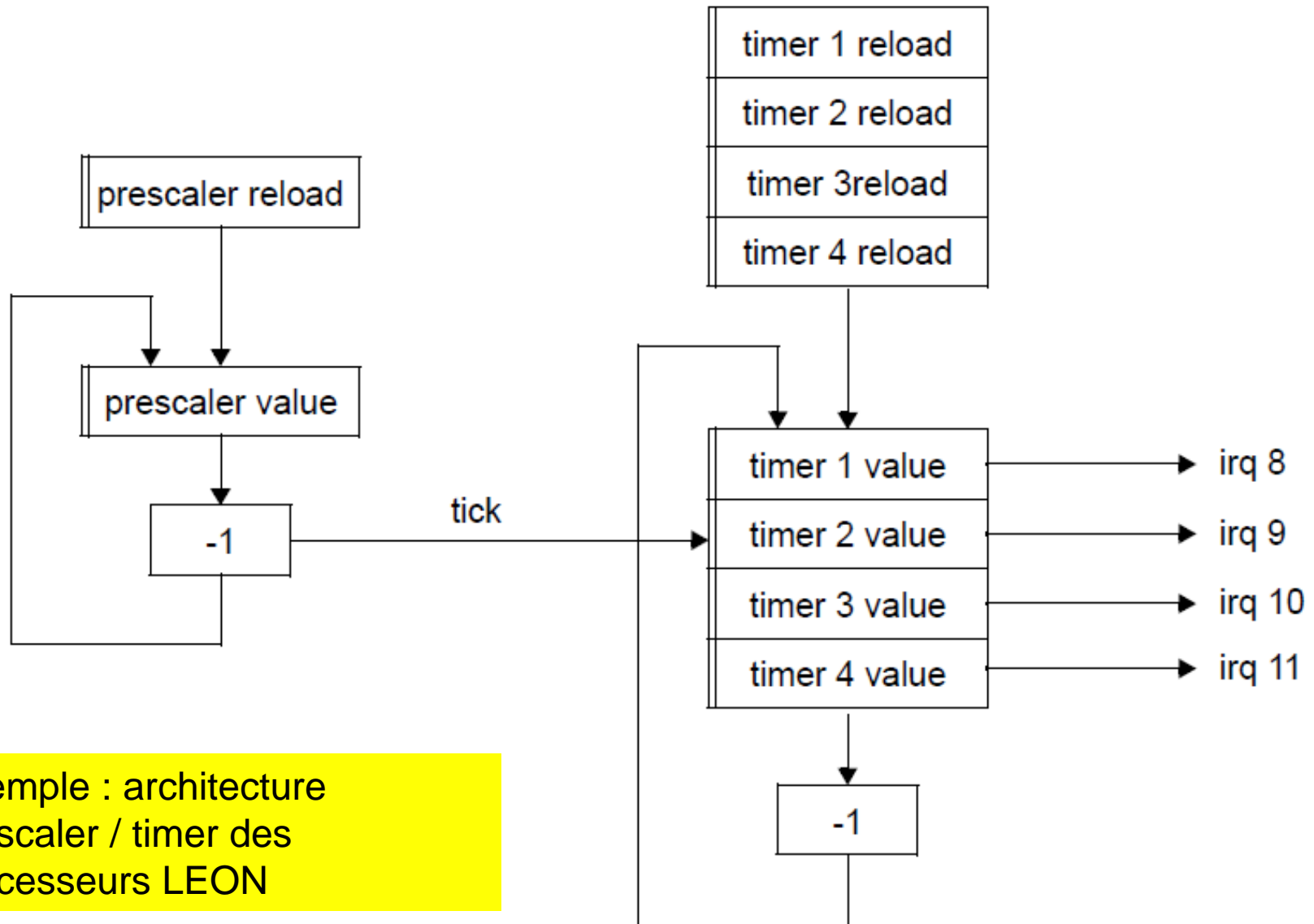
- Les processeurs embarqués intègrent généralement plusieurs timers hardware.
- Un timer est un compteur qui est décrémenté tous les n cycles processeurs et qui va pouvoir déclencher une interruption lorsqu'il atteint 0.
- Un timer peut être initialisé avec n'importe quelle valeur.
- Généralement, un des timers hardware est réservé au système d'exploitation et délivre un tick de base généralement réglé à 10 ms ou 1 ms.
 - Toutes les timers logiciels ou timeouts gérés par l'OS seront des multiples de ce tick de base.
 - La résolution de ce timer réservé au tick système est généralement de 1 μ s.

Mesure des temps d'exécution en utilisant les timers hardware du processeur

■ Les timers sont :

- soit directement connectés à l'horloge du processeur : à chaque cycle processeur, ils sont alors décrémentés de 1
 - Avec un timer 32 bits et une horloge à 50 MHz, la durée maximum d'expiration du timer est de $0.02\mu s \times 2^{32} = 85.9 \text{ s}$
- soit connectés à un prescaler (diviseur de fréquence) qui fournira le signal de décrémentement tous les n cycles processeurs, ce qui permet d'avoir des périodes d'expiration plus longues.
 - Le prescaler est lui connecté directement à l'horloge du processeur (ou à l'horloge du bus AMBA) .

Mesure des temps d'exécution en utilisant les timers hardware du processeur



Exemple : architecture
prescaler / timer des
processeurs LEON

Mesure des temps d'exécution en utilisant les timers hardware du processeur

- Réglage du timer système :
 - On règle généralement le prescaler de telle sorte qu'il délivre un tick chaque μs au timer système (le prescaler expire chaque μs) => le timer système à une résolution de 1 μs
 - On règle généralement le timer système pour qu'il génère un tick de 1ms ou 10ms.

Mesure des temps d'exécution en utilisant les timers hardware du processeur

■ Exercice :

- Fréquence processeur = 25 MHz
- Prescaler 16 bits ; Timer système 32 bits
- On souhaite une résolution de 1 μ s du timer système et un tick système (délivré par le timer système) de 1 ms.
- Quelle valeur de rechargement faut-il mettre dans le prescaler ?
- Quelle valeur de rechargement faut-il mettre dans le timer système ?

Mesure des temps d'exécution en utilisant les timers hardware du processeur

■ Exercice :

- Fréquence processeur = 25 MHz
- Prescaler 16 bits ; Timer système 32 bits
- On souhaite une résolution de 1 μ s du timer système et un tick système (délivré par le timer système) de 10 ms.
- Quelle valeur de rechargement faut-il mettre dans le prescaler ?
- Quelle valeur de rechargement faut-il mettre dans le timer système ?

■ Solution :

- $1 / 25 \text{ MHz} = 0.04 \mu\text{s}$
- $\text{prescaler_load} = 1 \mu\text{s} / 0.04 \mu\text{s} = 25$
- $\text{system_timer_load} = 10 \text{ ms} / 1 \mu\text{s} = 10000$

Mesure des temps d'exécution en utilisant les timers hardware du processeur

- Pour faire des mesures de temps, on peut accéder directement aux timers du processeur.
- Les bibliothèques BSP (Board Support Package) des processeurs, ainsi que les RTOS, proposent généralement des fonctions donnant :
 - le nombre de ticks écoulés depuis le démarrage du processeur
 - le temps écoulé en μ s depuis le démarrage du processeur
- Ex. : fonction `clock()` de la bibliothèque `libleonbare`
- Cette approche de la mesure du temps basée sur l'utilisation des timers peut être utilisée sur simulateur ou sur une cible matérielle.

Mesure des temps d'exécution en utilisant les timers hardware du processeur

- La fonction `get_elapsed_time()` donnée ci-dessous permet de donner le temps écoulé en μ s entre deux appels de la fonction :

```
int32_t get_elapsed_time() {  
    static int32_t last_time = 0;  
    int32_t elapsed_time = 0;  
    int32_t current_time = clock();  
  
    elapsed_time = current_time - last_time;  
    last_time = current_time;  
    return elapsed_time;  
}
```

Automatisation des mesures de performance avec un script GDB

- Une fonction `test()` est créée pour « instrumenter » le code : elle sera utilisée pour positionner de façon générique des points d'arrêt GDB dans le code aux endroits où l'on souhaite réaliser une mesure de temps d'exécution :

```
uint32_t test(const char * test_name, uint32_t test_id, int32_t time)
{
    uint32_t static test_counter = 0;
    return test_counter++;
}
```

Automatisation des mesures de performance avec un script GDB

■ Exemple de code instrumentalisé avec la fonction test() :

```
#define FACTORIAL_ARRAY_SIZE 10
uint32_t factorial_array[FACTORIAL_ARRAY_SIZE] __attribute__((section
(".bss_factorial_array")));
float square_result;
```

```
int main( void ) {
    square_result = 0.0f;
```

```
/* performance statistic initialization */
```

```
test("start",0, get_elapsed_time());
```

Initialisation des mesures

```
/* performance test of store_factorial() function */
```

```
store_factorial(factorial_array,FACTORIAL_ARRAY_SIZE);
```

```
test("store_factorial",1,get_elapsed_time());
```

Point de mesure

```
/* performance test of display_factorial() function */
```

```
display_factorial(factorial_array,FACTORIAL_ARRAY_SIZE);
```

```
test("display_factorial",1,get_elapsed_time());
```

Point de mesure

```
/* performance test of square() function */
```

```
square_result = square(5.67);
```

```
test("square",1, get_elapsed_time());
```

Point de mesure

Automatisation des mesures de performance avec un script GDB

■ Suite du code instrumentalisé

```
/* performance tests of factorial() function */
factorial(10);
test("factorial(10)",1, get_elapsed_time());
factorial(100);
test("factorial(100)",1, get_elapsed_time());
factorial(1000);
test("factorial(1000)",1, get_elapsed_time());
factorial(10000);
test("factorial(10000)",1, get_elapsed_time());

/* end of performance measurements */
test("end",2, clock());
return 0;
}
```

Automatisation des mesures de performance avec un script GDB

■ Script GDB générique pour la mesure des performances

```
set remotetimeout 10000

shell echo set logging file
C:/workspace/factorial/tests/%target%/gdb_result_002.txt >
    C:/workspace/factorial/temp/tmp.txt
source C:/workspace/factorial/temp/tmp.txt
shell del C:/workspace/factorial/temp/tmp.txt

set logging overwrite on
set logging on
set height 0
set print pretty on
set print array on

tar extended-remote localhost:1234

load
mon perf reset
```

Connexion au simulateur

Chargement de l'application

Automatisation des mesures de performance avec un script GDB

■ Suite du script GDB

```
hbreak test
commands
  silent
  printf "\n\n**** test = %s **** time = %d\n", test_name, time

  if (test_id == 0)
    print factorial_array
    print square_result
    mon perf reset
  end
  if (test_id == 1)
    mon perf
    mon perf reset
  end
  if (test_id == 2)
    print factorial_array
    print square_result
  end
end
cont

end
start
cont
detach
```

Affichage du temps écoulé en μ s
calculé à l'aide du timer système et
de la fonction clock()

Affichage de l'état initial des
variables

Affichage des statistiques de
performance fournies par le
simulateur

Affichage de l'état final des
variables

Automatisation des mesures de performance avec un script GDB

- Affichage de l'état initial des variables après le point d'arrêt sur `test("start", 0, get_elapsed_time());`

```
**** test = start **** time = 23049
$1 = {0,
      0,
      0,
      0,
      0,
      0,
      0,
      0,
      0,
      0}
$2 = 0
```

Automatisation des mesures de performance avec un script GDB

- Affichage du temps d'exécution de la fonction `store_factorial()` après le point d'arrêt sur
`test("store_factorial",1,get_elapsed_time());`

```
**** test = store_factorial **** time = 18
```

```
Cycles      :      846
```

```
Instructions :      367
```

```
Overall CPI  :      2.31
```

846 cycles = 16.92 μ s à
comparer aux 18 μ s retournés
par `get_elapsed_time()`

```
CPU performance (50.0 MHz) : 21.69 MOPS (21.69 MIPS, 0.00 MFLOPS)
```

```
Cache hit rate      : 95.4 % (inst: 95.7, data: 60.0)
```

```
AHB bandwidth utilisation : 0.0 % (inst: 0.0, data: 0.0)
```

```
Simulated time      : 0.02 ms
```

```
Processor utilisation : 100.00 %
```

```
Real-time performance : 1.69 %
```

```
Simulator performance : 0.37 MIPS
```

```
Used time (sys + user) : 0.00 s
```

Automatisation des mesures de performance avec un script GDB

■ Affichage du temps d'exécution de la fonction `display_factorial()` après le point d'arrêt sur

```
test("display_factorial",1,get_elapsed_time());
```

```
**** test = display_factorial **** time = 1021

Cycles      :      51062

Instructions :      20206
Overall CPI  :        2.53

CPU performance (50.0 MHz) : 19.79 MOPS (19.79 MIPS, 0.00 MFLOPS)
Cache hit rate      : 92.4 % (inst: 91.9, data: 96.2)
AHB bandwidth utilisation : 2.3 % (inst: 1.4, data: 0.9)
Simulated time      : 1.02 ms
Processor utilisation : 100.00 %
Real-time performance : 0.98 %
Simulator performance : 0.19 MIPS
Used time (sys + user) : 0.10 s
```

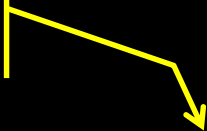
Automatisation des mesures de performance avec un script GDB

- Affichage du temps d'exécution de la fonction square() après le point d'arrêt sur
`test("square",1,get_elapsed_time());`

```
**** test = square **** time = 3
```

```
Cycles      :      136  
Instructions :       45  
Overall CPI :     3.02
```

On voit ici que le
FPU est utilisé



```
CPU performance (50.0 MHz) : 16.54 MOPS (16.18 MIPS, 0.37 MFLOPS)  
Cache hit rate             : 82.7 % (inst: 82.6, data: 71.4)  
AHB bandwidth utilisation  : 0.0 % (inst: 0.0, data: 0.0)  
Simulated time             : 0.00 ms  
Processor utilisation       : 100.00 %  
Real-time performance      : 1.#J %  
Simulator performance      : 0.00 MIPS  
Used time (sys + user)     : 0.00 s
```

Automatisation des mesures de performance avec un script GDB

- Affichage de l'état final des variables après le point d'arrêt sur `test("end", 2, clock());`

```
**** test = end **** time = 26100
$3 = {1,
      1,
      2,
      6,
      24,
      120,
      720,
      5040,
      40320,
      362880}
$4 = 32.1488991
```


Influence des options d'optimisation, du cache et de la présence d'un FPU sur les performances

- Le tableau ci-dessous compare les temps d'exécution en nombre de cycles machine pour différentes fonctions et différentes configurations (FPU / pas de FPU, cache / pas de cache, -O0 / -O2)

	-O0	-O2	-O2 (FPU)	-O3 (FPU)	-O2 (FPU) (NOCACHE)
store_factorial	1985	849	846	848	2226
display_factorial	51728	51261	51062	51692	114933
square	904	802	136	115	277
factorial(10)	499	249	200	205	516
factorial(100)	2659	1024	1000	1002	2766
factorial(1000)	24259	9096	9101	9100	25266
factorial(10000)	240292	90097	90113	90091	250271

On constate :

- ✓ La fonction factorial() est bien de complexité $O(n)$.
- ✓ Le rôle prépondérant du cache.
- ✓ Le rôle prépondérant du FPU (voir ligne « square »).

Mesure du temps et de la charge CPU

1. Introduction
2. Techniques de mesure du temps et évaluation des temps d'exécution
3. Etablissement d'un budget CPU

Etablissement d'un budget CPU

■ Taux d'occupation CPU

- Un taux d'occupation CPU est donné pour une période $T \Rightarrow$ c'est donc toujours une moyenne
- Il est défini comme étant : T_e / T où T_e est le temps pendant lequel le CPU est réellement actif durant la période T , c'est-à-dire exécutant des instructions ne correspondant pas à une boucle « idle » (« idle » = « not working »)
 - $T = T_e + T_s$ où T_s correspond au temps pendant lequel le CPU est soit endormi (certains CPU possède un mode « power down » pendant lequel le CPU est gelé en attente d'une interruption extérieure), soit dans une boucle « idle »

Etablissement d'un budget CPU

■ Remarque :

- Un CPU fonctionnant dans une boucle idle (celle d'un OS par exemple) consomme des cycles machines. On peut ne pas en tenir compte dans l'évaluation de la charge CPU car on considère que ces cycles peuvent être « récupérés » pour d'autres traitements.
- Un CPU en instantané fonctionne toujours à 100% de ses capacités : le taux d'occupation CPU correspond à une moyenne sur une période donnée.

Etablissement d'un budget CPU

- Quand on évalue un taux d'occupation CPU, on doit construire un modèle prenant en compte :
 - Le temps d'exécution unitaire de chacun des traitements implémentés : ces temps d'exécution sont mesurés avec des techniques vues dans le chapitre précédent.
 - La fréquence à laquelle ces traitements sont appelés (période d'activation)

Etablissement d'un budget CPU

- Mais attention, l'évaluation d'une charge CPU n'est jamais quelque chose de triviale.
- Il faut prendre en compte aussi :
 - Les optimisations algorithmiques potentielles
 - Le mécanisme de mémoire cache (cache instructions, cache de données)
 - L'impact de l'architecture temps réel de l'application → importance du prototypage dès les premières phases des projets
 - → Il faut prendre des marges quand on réalise le dimensionnement d'une carte processeur !

Etablissement d'un budget CPU

- En fin de phase A d'un projet (phase de démarrage du projet / études de faisabilité), le taux d'occupation CPU évalué / mesuré ne doit pas dépasser 30-40%
- En fin de phase B d'un projet (phase de conception préliminaire), le taux d'occupation CPU évalué / mesuré ne doit pas dépasser 50-60%.
- Au moment de la livraison du logiciel embarqué, le taux d'occupation CPU ne doit pas dépasser 80%.

Etablissement d'un budget CPU

■ Exemple :

- Soit une application de traitement d'images temps réel devant traiter 10 images par seconde. Les traitements suivants doivent être effectués pour chaque image :

Traitement	Temps d'exécution unitaire en cycles machines
Acquisition de l'image	230000
Correction du bruit	2330000
Suppression des hot spots	1952000
Compression	820000
Transmission de l'image compressée	430000
Total	5762000

Etablissement d'un budget CPU

- Q1 : Sachant que le processeur réalisant le traitement fonctionne à 75 MHz, quel est le taux d'occupation CPU ?
- Q2 : Quel serait le taux d'occupation si le processeur était cadencé à 50 MHz ?
- Q3 : A quelle cadence devrait fonctionner le processeur si l'on souhaitait traiter 50 images par seconde tout en ayant une charge CPU maximum de 50% ?

Etablissement d'un budget CPU

■ Q1 :

- On traite 10 images par seconde => tous les traitements doivent être réalisés en 0.1 s
- Taux d'occupation CPU = $(5762000 * 1 / 75000000) / 0.1 = 0.07682 / 0.1 = 0.76 = 76\%$

■ Q2 :

- Taux d'occupation CPU = $(5762000 * 1 / 50000000) / 0.1 = 0.115 / 0.1 = 1.15 = 115\%$
- On peut en conclure que la fréquence processeur n'est pas assez élevée.

Etablissement d'un budget CPU

■ Q3 :

- On traite 50 images par seconde => tous les traitements doivent être réalisés en 0.02 s
- Charge CPU = 50% = 0.5 = $(5762000 * 1 / \text{Freq_Proc}) / 0.02$
- $\text{Freq_Proc} = 5762000 / (0.5 * 0.02) = 576\,200\,000 = 576\text{ MHz}$