INF3610 - Systèmes embarqués

Hiver 2024

Laboratoire 1 – Partie 3 : Partitionnement de l'application de routage sur 2 cœurs en mode AMP

Groupe 01 – Équipe 30

2221053 – Thomas Rouleau

2043696 – Étienne Hourdebaigt

Soumis à : M. Guy Bois

2024-03-24

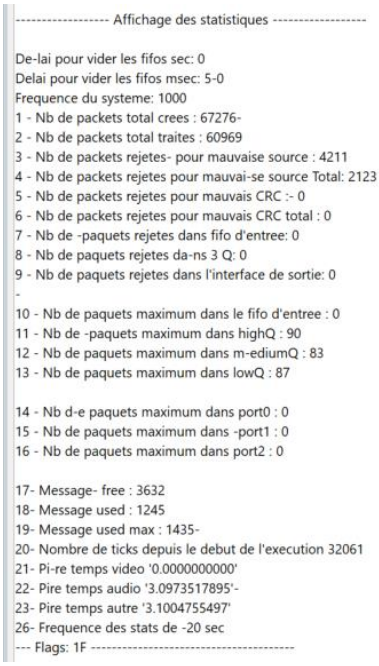
5. Questions sur le TP

Question 1) Lors des modifications à *TaskComputing*, en créant un packet avec MemGet, vous avez recopié le timestamp envoyé par *TaskGenerate*.

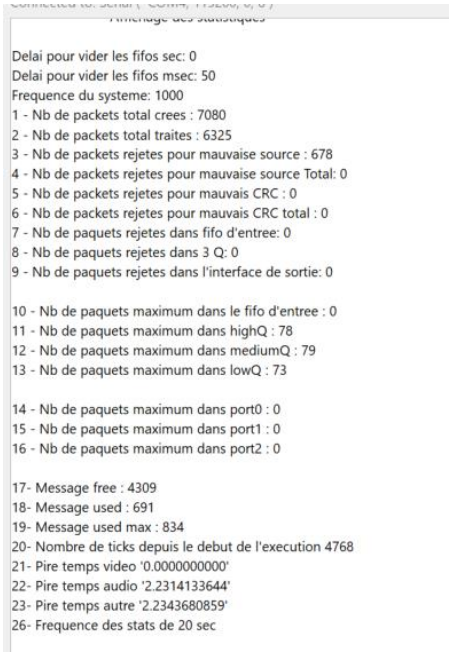
- ***a)*** *Sachant que l'horloge du timestamp à 325 MHz (période de 1/325 MHz) est le même horloge pour les 2 cores, que représente le timestamp d'un paquet quand il arrive dans les fifos de OutputPort. Donnez le print des statistiques après 1 minute lorsque delai_pour_vider_les_fifos_sec = 0 et delai_pour_vider_les_fifos_msec = 50 et sans attente active dans TaskComputing. Êtes-vous surpris de ces chiffres. Expliquez.*

Après vérification avec le professeur, il s'avère que le timestamp obtenu avec CPU_TS_Get64() diffère entre les deux cœurs. Toutefois, il est possible de remédier à ce problème en enregistrant le timestamp du cœur 1 au moment du startupTask dans la mémoire partagée, afin de l'utiliser comme référence pour calculer un décalage à soustraire au timestamp du cœur 0, également au moment de son exécution de sa startupTask. Cette approche permet de simuler un synchronisme des horloges de timestamp des deux cœurs.

Le timestamp d'un paquet lorsqu'il atteint les files de OutputPort représente le temps d'un échange entre les 2 cœurs combiné au temps de traitement des FIFOs. Il reflète donc la latence du traitement d'un paquet depuis sa génération jusqu'à son traitement final. Les valeurs obtenues dans nos résultats diffèrent de celles fournies par le professeur (21- Pire temps vidéo '2.4879696369', 23- Pire temps audio '2.4910106659', 25- Pire temps autre '2.4796819687'), et peuvent varier en fonction du délai entre l'activation des deux cœurs.



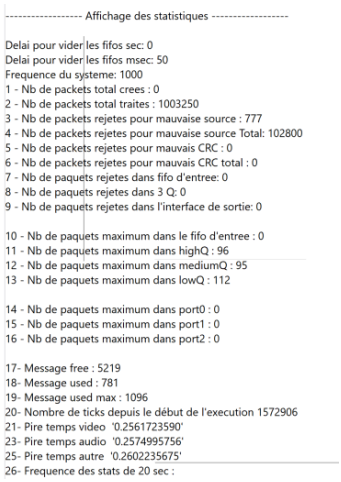
```
---------- Affichage des statistiques ----------

De-lai pour vider les fifos sec: 0
Delai pour vider les fifos msec: 5-0
Frequence du systeme: 1000
1 - Nb de packets total crees : 67276-
2 - Nb de packets total traites : 60969
3 - Nb de packets rejetes- pour mauvaise source : 4211
4 - Nb de packets rejetes pour mauvai-se source Total: 2123
5 - Nb de packets rejetes pour mauvais CRC :- 0
6 - Nb de packets rejetes pour mauvais CRC total : 0
7 - Nb de -paquets rejetes dans fifo d'entree: 0
8 - Nb de paquets rejetes da-ns 3 Q: 0
9 - Nb de paquets rejetes dans l'interface de sortie: 0
-
10 - Nb de paquets maximum dans le fifo d'entree : 0
11 - Nb de -paquets maximum dans highQ : 90
12 - Nb de paquets maximum dans m-ediumQ : 83
13 - Nb de paquets maximum dans lowQ : 87

14 - Nb d-e paquets maximum dans port0 : 0
15 - Nb de paquets maximum dans -port1 : 0
16 - Nb de paquets maximum dans port2 : 0

17- Message- free : 3632
18- Message used : 1245
19- Message used max : 1435-
20- Nombre de ticks depuis le debut de l'execution 32061
21- Pi-re temps video '0.0000000000'
22- Pire temps audio '3.0973517895'-
23- Pire temps autre '3.1004755497'
26- Frequence des stats de -20 sec
--- Flags: 1F ----------------------------------------
```



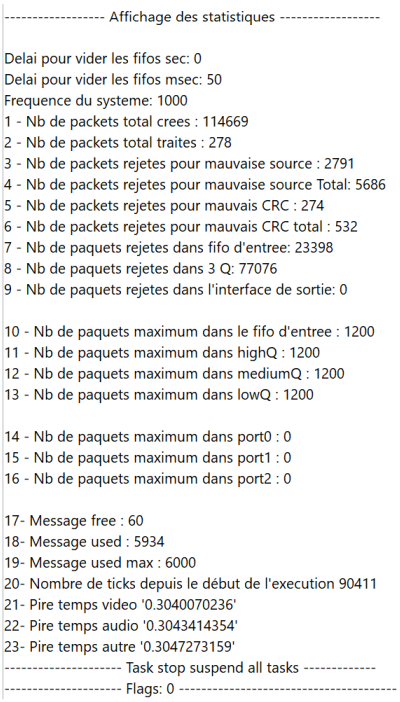
Les pires temps observés pour tous les types de paquets sont nettement plus élevés par rapport à ceux obtenus lorsque toutes les tâches, y compris taskGenerate, étaient exécutées sur le même cœur (voir question 1.b). Étant donné que le temps de traitement des files FIFO n'a probablement pas changé entre ces deux types d'implémentations, cette différence de temps provient vraisemblablement du transfert d'informations d'un cœur à l'autre.

Le temps nécessaire à l'écriture et à la lecture dans la mémoire partagée DDR est plus long que celui requis pour l'écriture et la lecture dans la mémoire cache, qui n'est pas partagée entre les deux cœurs. De plus, une partie du pire temps est probablement due au délai entre les activations de taskGenerate et de taskComputing. L'activation des cœurs avec la commande "resume" dans la fenêtre de débogage de Xilinx SDK introduit un délai humain, et ce temps représentera toujours le pire scénario.

b) *Plutôt que de recopié le timestamp, vous auriez aussi pu le remettre au temps courant (i.e. à packet->timestamp = CPU_TS_Get64()). Soit le résultat suivant que j'ai obtenu en générant 100,000 paquets avec delai_pour_vider_les_fifos_sec = 0 et delai_pour_vider_les_fifos_msec = 50 et une attente active de 1 ms dans TaskComputing. Est-ce un meilleur résultat que ce que vous avez obtenu à la section 5.3.3 du de la partie 1 du lab 1? Expliquez ce résultat.*



Voici un rappel de nos résultats à la section 5.3.3 du de la partie 1 du lab 1 avec *delai_pour_vider_les_fifos_sec* = 0 et *delai_pour_vider_les_fifos_msec* = 50 :



Les pires temps obtenus sont inférieurs à ceux relevés dans la section 5.3.3 de la partie 1 du laboratoire 1. Cette disparité s'explique par le déplacement de TaskGenerate vers un autre cœur que celui du routeur. Ce changement permet à la génération et à la consommation de paquets de ne pas rivaliser pour les mêmes ressources, leur permettant ainsi d'opérer en parallèle sans contrainte. Dans l'ancienne

configuration, il était possible que TaskGenerate ait la priorité sur TaskComputing avant l'enregistrement du timestamp, ce qui aurait pu augmenter le délai.

Question 2) Quels sont les avantages d'avoir amené TaskGenerate sur le core 1. Aussi, aurait-on pu faire rouler TaskGenerate sur un simple baremetal (standalone dans SDK) plutôt que uC/OS-III? Expliquez.

Il semble que la latence entre le traitement de chaque paquet soit plus élevée en raison des délais d'écriture et de lecture dans la DDR, mais que le taux de traitement global soit amélioré grâce à une meilleure répartition des ressources, obtenue par le transfert de taskGenerate() sur le core 1. En attribuant les tâches à différents cœurs, nous favorisons leur parallélisation, ce qui réduit le temps de traitement par rapport à une exécution sur un seul cœur, améliorant ainsi les performances.

Nous pensons qu'il n'est pas réalisable de faire fonctionner TaskGenerate sur un simple système bare-metal. TaskGenerate utilise des mécanismes permettant l'écriture et la lecture de la mémoire DDR, ce qui représente un avantage des systèmes RTOS tels que uC/OS-III par rapport au bare-metal, où la gestion des ressources est effectuée directement dans le code de l'application. Le code utilise des fonctionnalités spécifiques à un RTOS, telles que les mutex et l'ordonnancement des tâches. Cependant, il serait possible de le mettre en œuvre en bare-metal moyennant des modifications significatives à son implémentation. Bien qu'il s'agisse d'une application monotâche répétitive, ce qui la rendrait propice à une implémentation bare-metal, nous devrions supprimer toutes les fonctions spécifiques aux RTOS, telles que la génération de tâches et l'utilisation des mutex, ou les réimplémenter d'une manière adaptée au contexte bare-metal.