

- PROJET RTEL4I -



**Conception détaillée des moyens de test du portage
PREEMPT-RT sur processeur NIOS2 (L2.5-b)**



SOMMAIRE

1. Cartes cibles.....	5
1.1. Carte Altera Stratix 1S10.....	5
1.2. Carte Altera Cyclone III 3C25.....	6
2. Programmes utilisés.....	7
2.1. Ping flooding.....	7
2.2. Script godd.....	7
2.3. Programme stress.....	7
2.4. Script gos.....	8
2.5. Programme hackbench.....	8
2.6. Script goh.....	8
2.7. Script goperf.....	8
2.8. Programme cyclictst.....	8
2.9. Programme nanosleep.....	8
2.10. Script make_hist.sh.....	9
2.11. Script golatency.....	9
2.12. Script gohist.....	9
3. Protocoles de test.....	9
4. Résultats.....	10
4.1. Mesures avec goperf.....	10
4.2. Mesures avec godd.....	12
4.3. Mesures avec combinaisons.....	16
4.4. Mesures avec stress.....	17
4.5. Mesures à l'oscilloscope.....	20
5. Références.....	25
6. Annexe 1 : shell script godd.....	26
7. Annexe 2 : shell script gos.....	26
8. Annexe 3 : shell script goh.....	26
9. Annexe 4 : shell script goperf.....	26
10. Annexe 5 : fichier source nanosleep de génération d'un signal périodique.....	27
11. Annexe 6 : Pilote de périphérique pour accéder aux leds de la carte cible Stratix 1S10.....	29
12. Annexe 7 : shell script make_hist.sh.....	32
13. Annexe 8 : shell script golatency.....	34
14. Annexe 9 : shell script gohist.....	35

Fiche synthétique

Matériel	
Processeur cible	NIOS II avec MMU
Fondeur	Altera
Outils de synthèse	Quartus II version 9.1 et supérieur
Carte cible	Altera Stratix 1S10, Altera Cyclone III 3C25
Logiciel	
Distribution Linux	Fedora 13
Version Linux pour NIOS II	Linux 2.6.33
Version compilateur croisé pour NIOS II	4.1.2
Version patch pour NIOS II	preempt-rt-2.6.33.6-rt27-nios2-1.0-00.patch
Version patch PREEMPT-RT pour Linux	2.6.33.6-rt27

Gestion du document :

Date	Version	Modifications	Relecture	Commentaires
05/09/10	1.0	PK	PK	Création fichier

1. CARTES CIBLES

2 cartes cibles ont été utilisées pour les mesures de performances.

1.1. Carte Altera Stratix 1S10

La première carte choisie est une carte d'évaluation moyenne gamme d'Altera : carte Stratix 1S10.

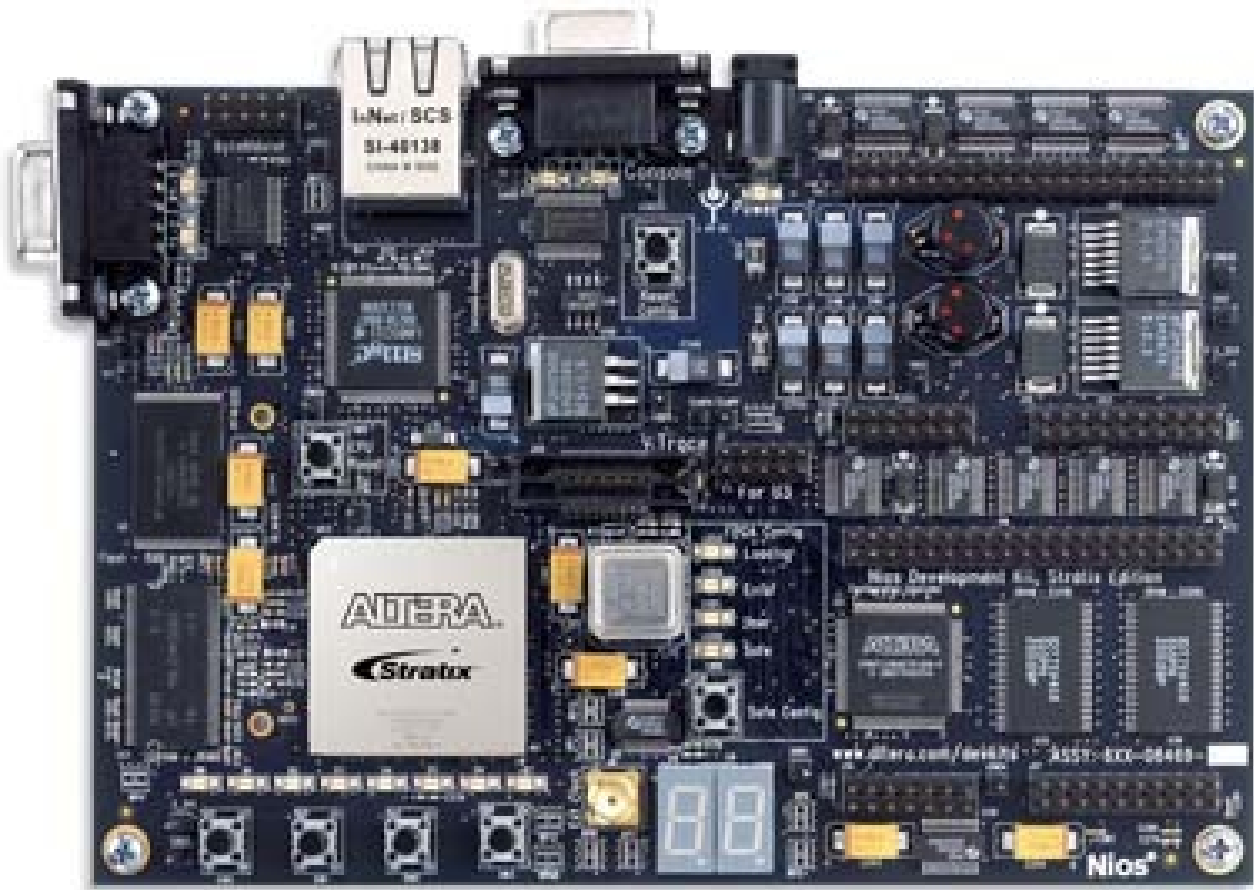


Figure 1 : Carte cible Altera Stratix 1S10

La carte Stratix 1S10 possède les caractéristiques suivantes :

- Circuit FPGA Stratix II EP2S30F672C5.
- 1 Mo de SRAM 16 bits.
- 16 Mo de SDRAM 32 bits.
- 16 Mo de mémoire Flash.
- 1 support CompactFlash type I.
- 1 interface Ethernet 10/100 Mb/s.
- 2 ports série (RS-232 DB9).
- 2 ports d'extension pour carte fille maison.
- 1 connecteur JTAG.
- 4 boutons poussoirs.
- 8 leds utilisateurs.
- 2 afficheurs 7 segments.

- 1 afficheur LCD 2x16 caractères.

1.2. Carte Altera Cyclone III 3C25

La deuxième carte choisie est une carte d'évaluation moyenne plus gamme d'Altera : carte Cyclone III 3C25.



Figure 2 : Carte cible Altera Cyclone III 3C25

La carte Stratix 1S10 possède les caractéristiques suivantes :

- Circuit FPGA Cyclone III 3C25F324 .
- 16 Mo de mémoire Flash.
- 32 Mo de mémoire DDR SDRAM à 133 MHz, 16 bits.
- 1 Mo de mémoire SRAM.
- Interface SD/MMC.
- Interface SPI à 20 MHz.
- Interface Ethernet MAC 10/100/1000 Base T.
- Interface JTAG UART.
- 1 port série (RS-232 DB9).
- Ecran LCD 800 × 480.
- 4 boutons poussoirs.
- 2 leds.

2. PROGRAMMES UTILISÉS

Les principaux programmes utilisés ont été choisis pour stresser la cible afin de :

- Générer des interruptions : *ping flooding*.
- Générer des E/S : *dd*, *stress*, *hackbench*.
- Générer de la charge CPU : *stress*, *hackbench*.

Les principaux programmes utilisés pour mesurer les temps de latence sont :

- Le programme *cyclictest*.
- L'oscilloscope. On l'utilise conjointement avec un programme comme *nanosleep* qui génère un signal périodique sur des E/S parallèles de la carte cible.

Les principaux programmes utilisés pour générer des courbes de mesures sont :

- Le dump de l'écran de l'oscilloscope.
- Le shell script *gohist* pour construire un histogramme établi à partir d'un fichier généré par *cyclictest*.
- Le shell script *golatency* pour construire un histogramme établi à partir d'un fichier généré par *cyclictest*.

2.1. Ping flooding

Il suffit depuis le PC hôte de lancer la commande :

```
# ping -f IP_cible
```

Cette commande permet de générer beaucoup d'interruptions du processeur via l'interface Ethernet de la carte cible.

2.2. Script godd

Le shell script *godd* donné en annexe 1 permet de générer des E/S vers fichier via la commande *dd*.

2.3. Programme stress

Le programme *stress* [1] permet de stresser la carte cible sur différents aspects :

- E/S : option *-i*
- Charge CPU : option *-c*.
- Mémoire : option *-m*.
- E/S vers fichier : option *-d*.

Exemple d'usage : 2 processus de charge CPU, 2 processus d'E/S, 2 processus de consommation mémoire (2 Mo), 2 processus d'E/S vers fichier (2 Mo) :

```
# stress -c 2 -i 2 -m 2 --vm-bytes 2MB -d 2 --hdd-bytes 2MB
```

2.4. Script gos

Le shell script `gos` donné en annexe 2 permet de stresser continuellement la carte cible avec `stress`.

2.5. Programme hackbench

Le programme `hackbench` [2] permet de stresser la carte cible en générant des groupes de 20 threads communiquant entre eux.

Exemple d'usage : 1 groupe de 20 threads communiquant pendant un temps limité :

```
# hackbench 1
```

2.6. Script goh

Le shell script `goh` donné en annexe 3 permet de stresser continuellement la carte cible avec `hackbench`.

2.7. Script goperf

Le shell script `goperf` donné en annexe 4 permet de stresser continuellement la carte cible avec `hackbench`. Suivant le protocole suivant :

- Lancement de `cyclictest` avec un timer périodique de 10000 μ s et génération d'un fichier de mesure.
- Attente de 5 secondes.
- Lancement de `hackbench`.
- Attente de 5 secondes.
- Arrêt de `cyclictest`.

2.8. Programme cyclictest

Le programme `cyclictest` [3] permet de mesurer le temps de latence sur un timer logiciel périodique. C'est un programme complet qui permet de créer des fichiers de mesure pour la construction d'histogramme ou de temps de latence.

Exemple d'usage : mesure du temps de latence pour un timer périodique de 10000 μ s avec la priorité Temps Réel la plus forte de 99 :

```
# cyclictest -n -p 99 -i 1000
```

Exemple d'usage : construction d'un fichier de mesure du temps de latence pour un timer périodique de 10000 μ s avec la priorité Temps Réel la plus forte de 99 :

```
# cyclictest -n -p 99 -i 1000 -v > cyclicdata.log
```

2.9. Programme nanosleep

Le programme `nanosleep` dont le code source est donné en annexe 5 permet de générer un signal électrique périodique sur une des sorties de la carte Stratix 1S10. Il est utilisé conjointement avec le pilote de périphérique `led.c` dont le code source est donné en annexe 6 pour accéder aux ressources physiques.

Exemple d'usage : génération d'un signal périodique de 10 ms avec un processus de priorité Temps Réel de priorité la plus forte 99 :

```
$ nanosleep 10000 99
```

2.10. Script `make_hist.sh`

Le shell script `make_hist.sh` donné en annexe 7 permet de trier les données du fichier de mesure généré avec `cyclictest` en vue de construire ensuite un graphique de temps de latence

2.11. Script `golatency`

Le shell script `golatency` donné en annexe 8 permet de construire un graphique de temps de latence à partir des données générées par `make_hist.sh`.

2.12. Script `gohist`

Le shell script `gohist` donné en annexe 9 permet de construire un histogramme du temps de latence à partir des données générées par `make_hist.sh`.

3. PROTOCOLES DE TEST

Les tests ont été effectués sur la carte cible Cyclone III 3C25 sauf pour les mesures de temps de latence à l'oscilloscope faites sur la carte cible Stratix 1S10 qui permet d'accéder facilement aux E/S.

Le temps CPU est alloué à 100 % aux threads Temps Réel [4] :

```
# echo -1 > /proc/sys/kernel/sched_rt_runtime_us
```

Ce point est crucial car Linux possède un mécanisme de partage du temps CPU sur une période de temps (1 seconde par défaut) entre les threads Temps Réel et les threads ordinaires. Par défaut, sur 1 seconde, 95 % du temps est dévolu aux threads Temps Réel, les 5 % restants sont pour les threads ordinaires. En effectuant le réglage précédent, tout le temps CPU est affecté aux threads Temps Réel si besoin bien sûr. C'est bien ce que l'on veut avec un système Temps Réel !

Différentes mesures de temps de latence ont été effectués avec les programmes de charge précédents, seuls ou combinés.

4. RÉSULTATS

4.1. Mesures avec goperf

Nous obtenons les courbes suivantes du temps de latence en fonction du temps (numéro d'échantillon divisé par 100 donne le temps en seconde) :

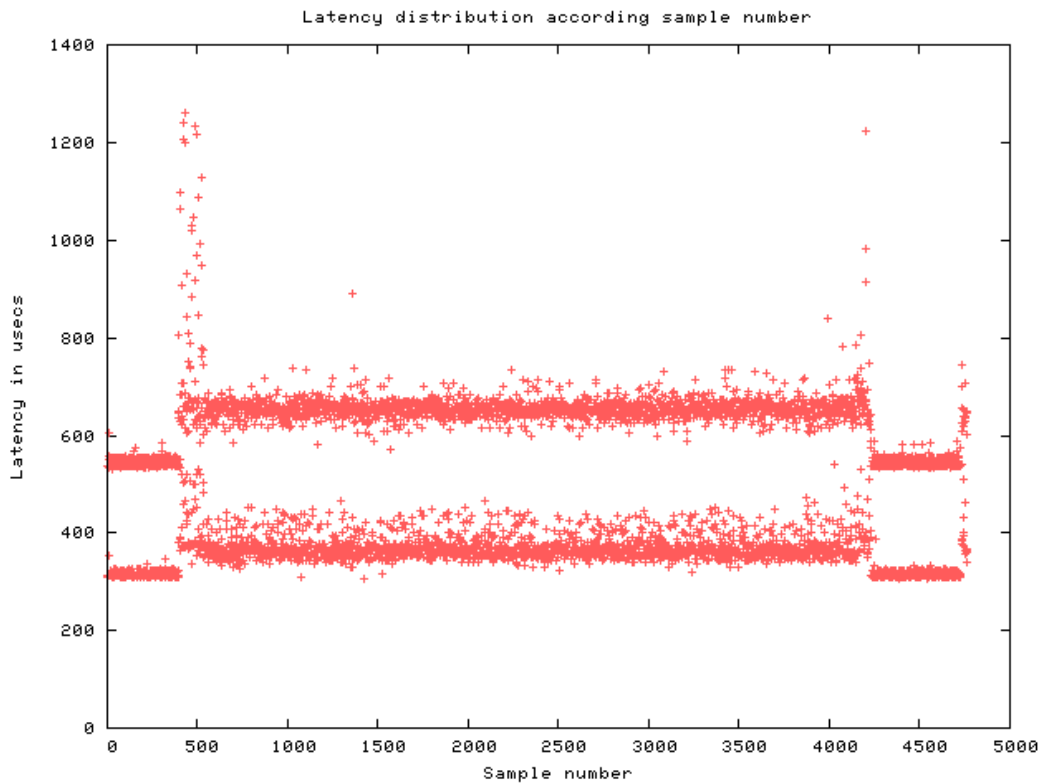
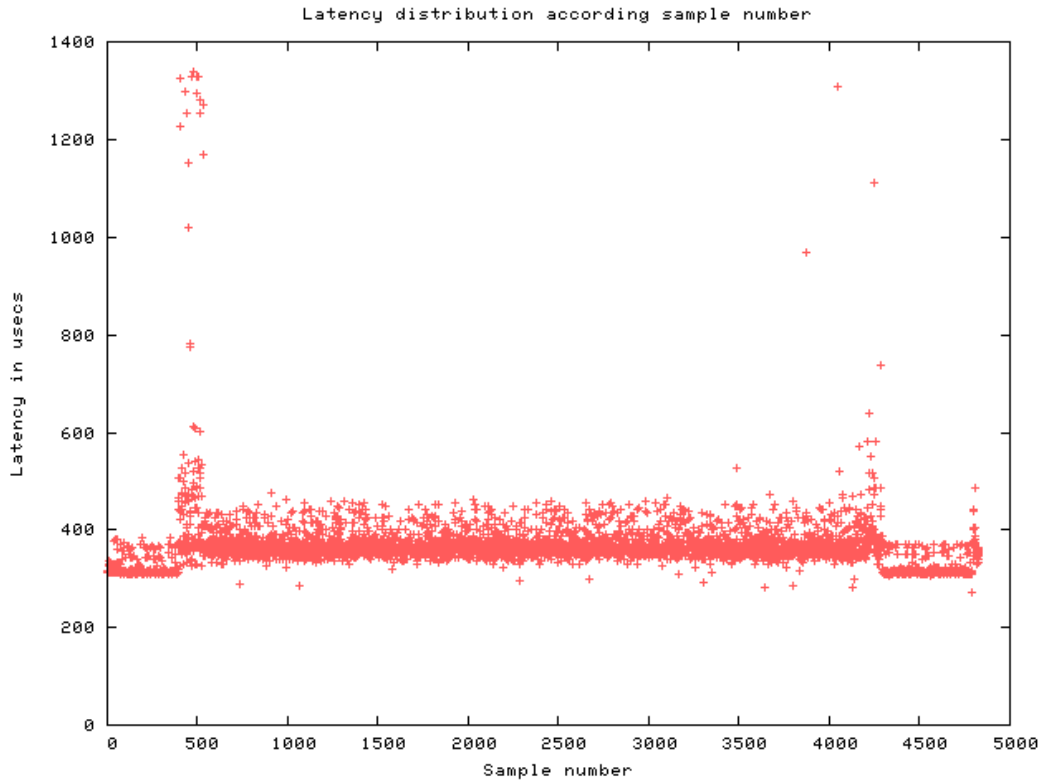


Figure 3 : Mesure du temps de latence au cours du temps avec `goperf`

On retrouve les paliers de repos de 5 secondes et la charge avec `hackbench`. On a un temps de latence autour de 400 μ s au repos, puis de 700 μ s en charge avec des pointes maximales à 1300 μ s au lancement et à la terminaison de `hackbench`.

On retrouve sur différentes mesures les 2 « barres verticales » dont l'effet est grossi par l'échelle des temps de latence.

Nous obtenons les courbes suivantes pour les histogrammes correspondant aux 2 mesures précédentes :

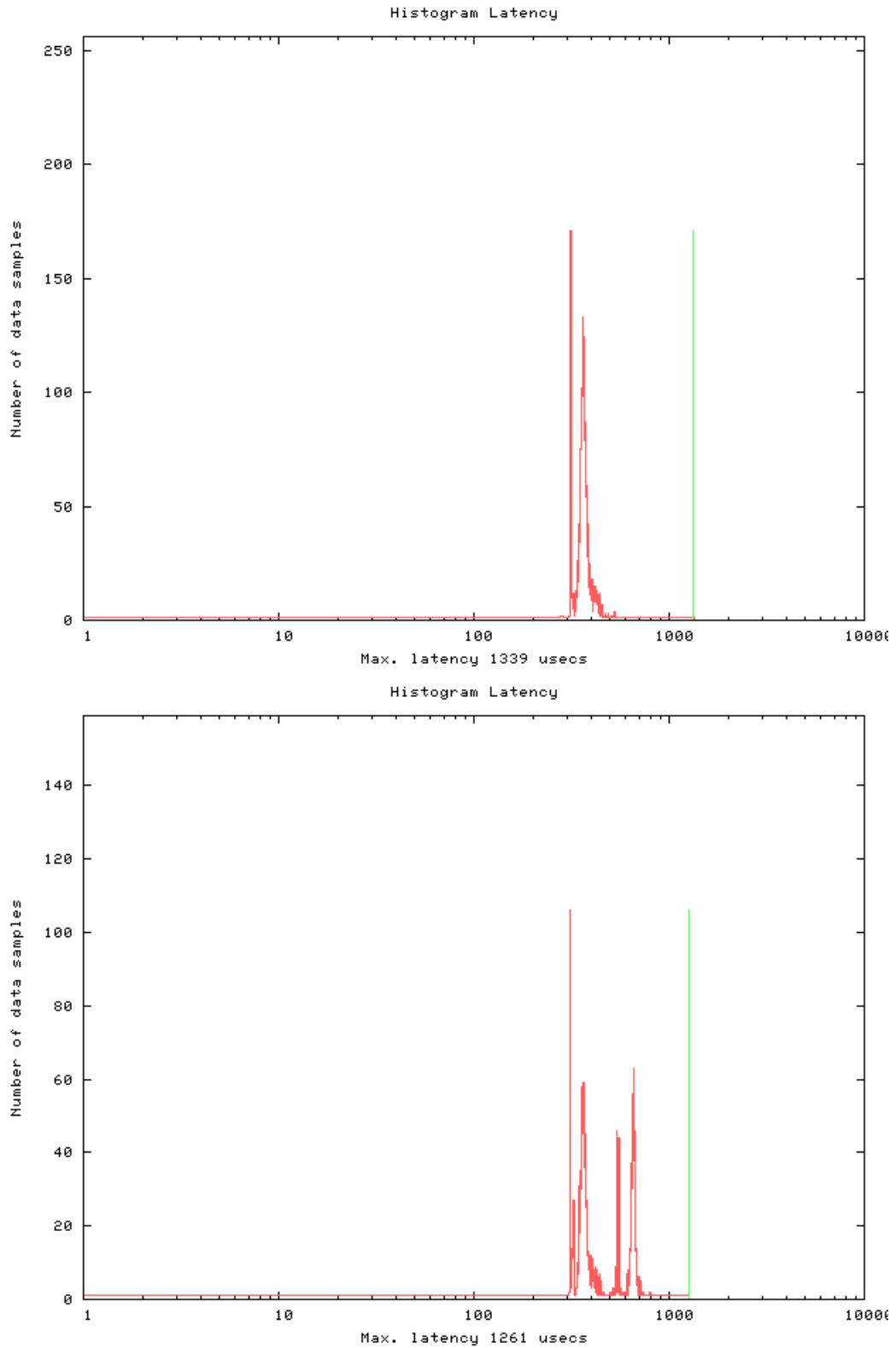


Figure 4 : Histogramme du temps de latence avec goperf

4.2. Mesures avec godd

Nous obtenons les courbes suivantes du temps de latence en fonction du temps (numéro d'échantillon divisé par 100 donne le temps en seconde) :

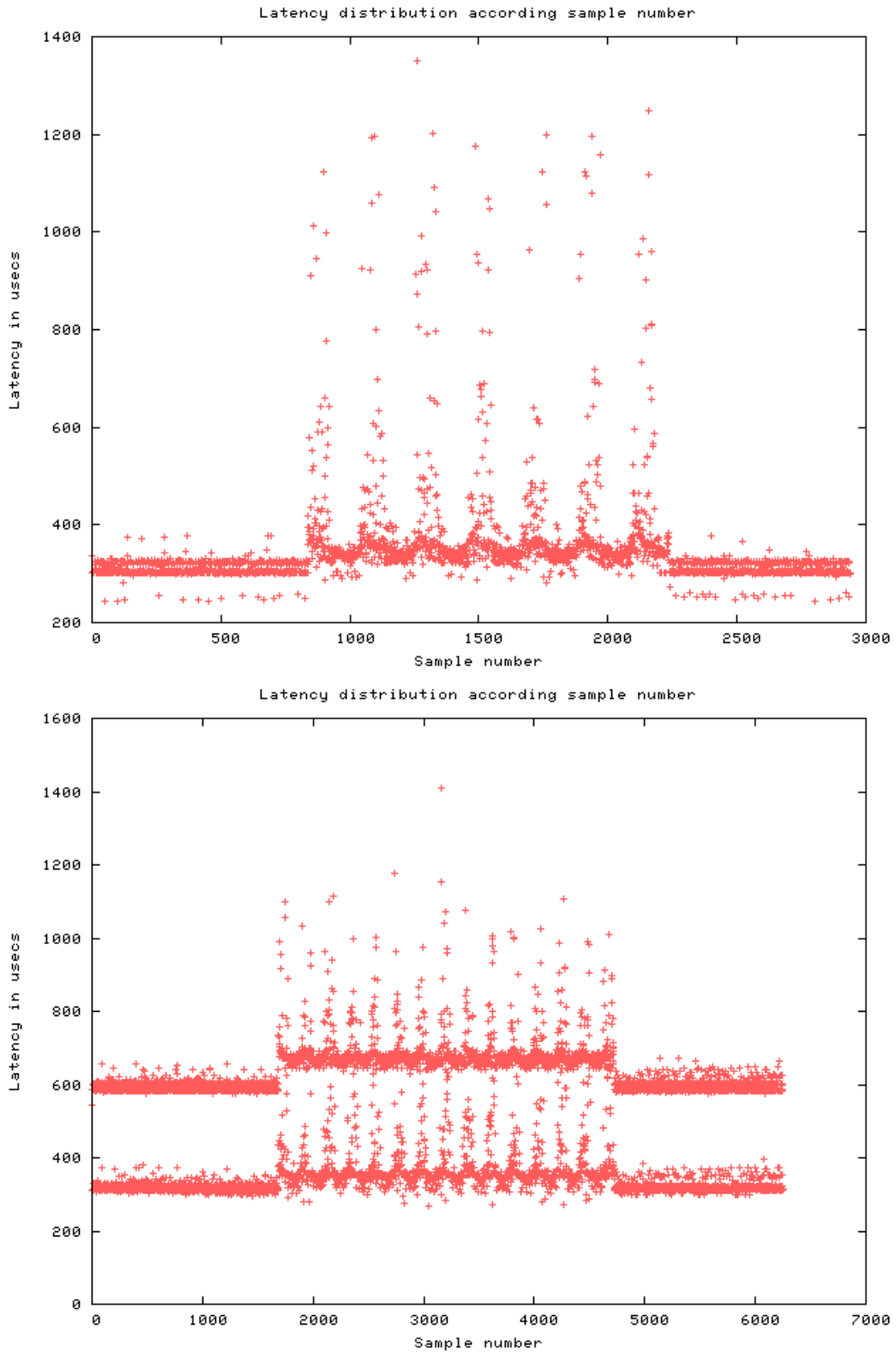


Figure 5 : Mesure du temps de latence au cours du temps avec godd

A chaque lancement d'un cycle d'écriture, on retrouve une salve de points hors épure avec une valeur maximale du temps de latence autour de 1400 μ s. Le temps de latence reste ensuite confiné autour de 700 μ s.

Nous obtenons les courbes suivantes pour les histogrammes correspondant aux 2 mesures précédentes :

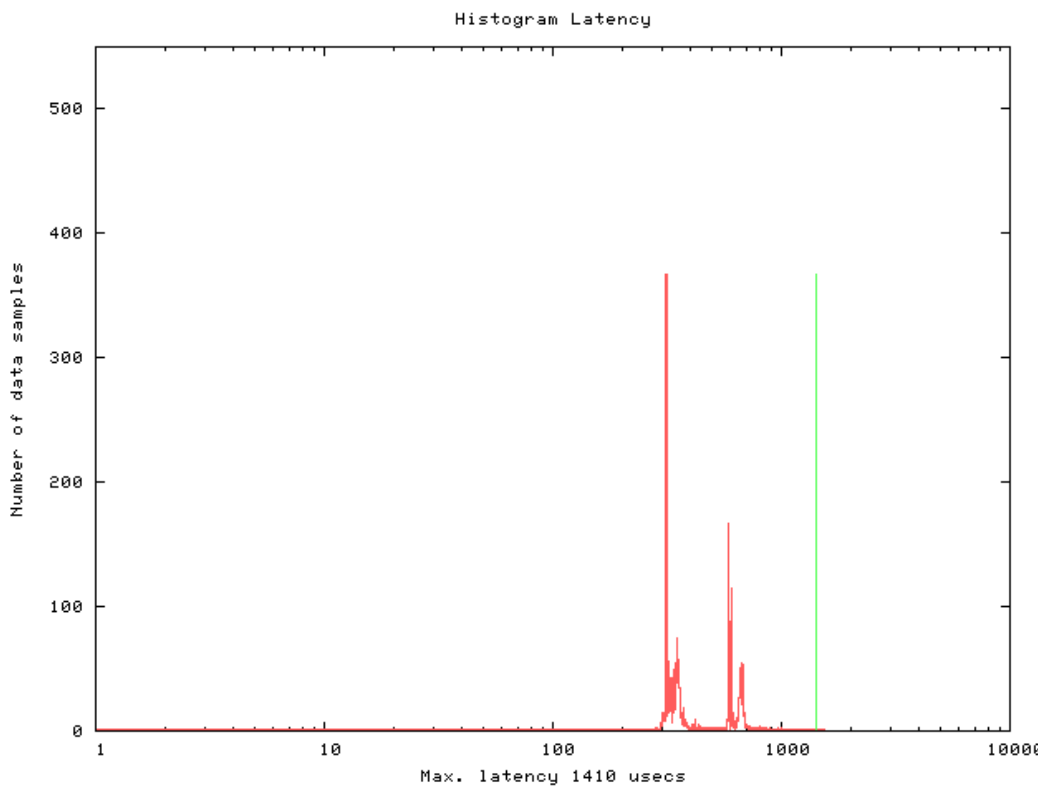
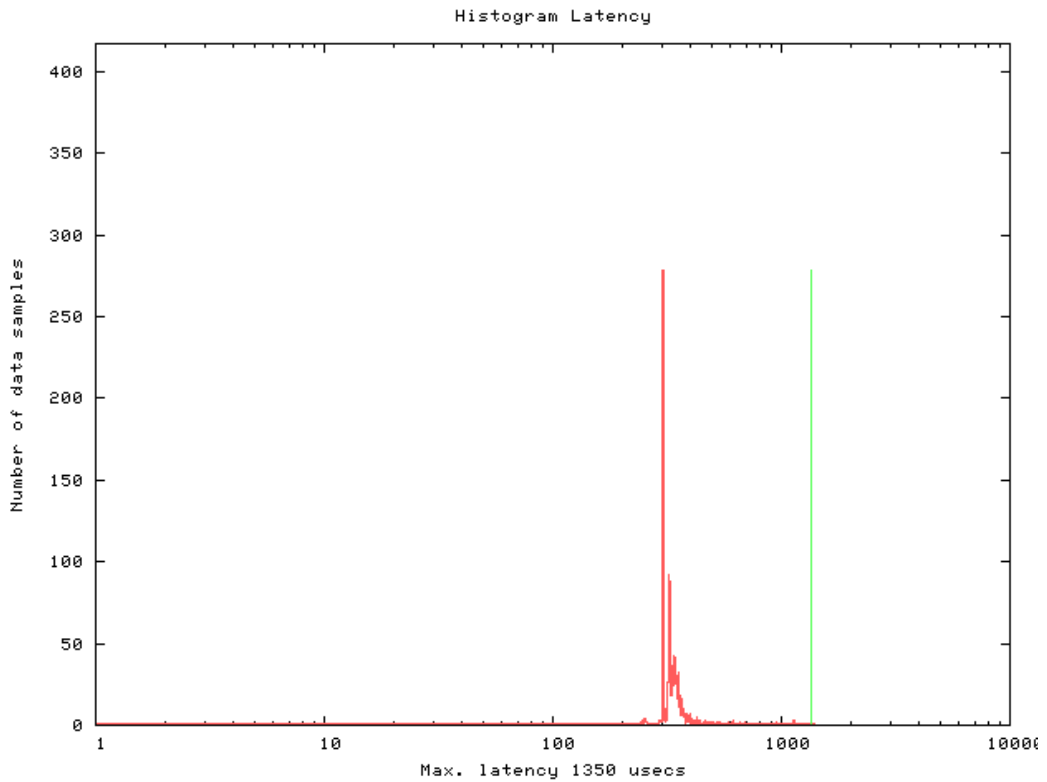


Figure 6 : Histogramme du temps de latence avec godd

4.3. Mesures avec combinaisons

Nous avons fait des mesures avec la combinaison de gohh, goh suivant le protocole suivant :

- Lancement de `cyclictest` avec un timer périodique de 10000 μ s et génération d'un fichier de mesure.
- Attente de quelques dizaines de secondes.
- Lancement de `godd`.
- Attente de 5 secondes.
- Lancement de 10 fois `hackbench`.
- Attente de quelques dizaines de secondes.
- Arrêt de `godd`.
- Arrêt de `cyclictest`.

Nous obtenons les courbes suivantes du temps de latence en fonction du temps (numéro d'échantillon divisé par 100 donne le temps en seconde) :

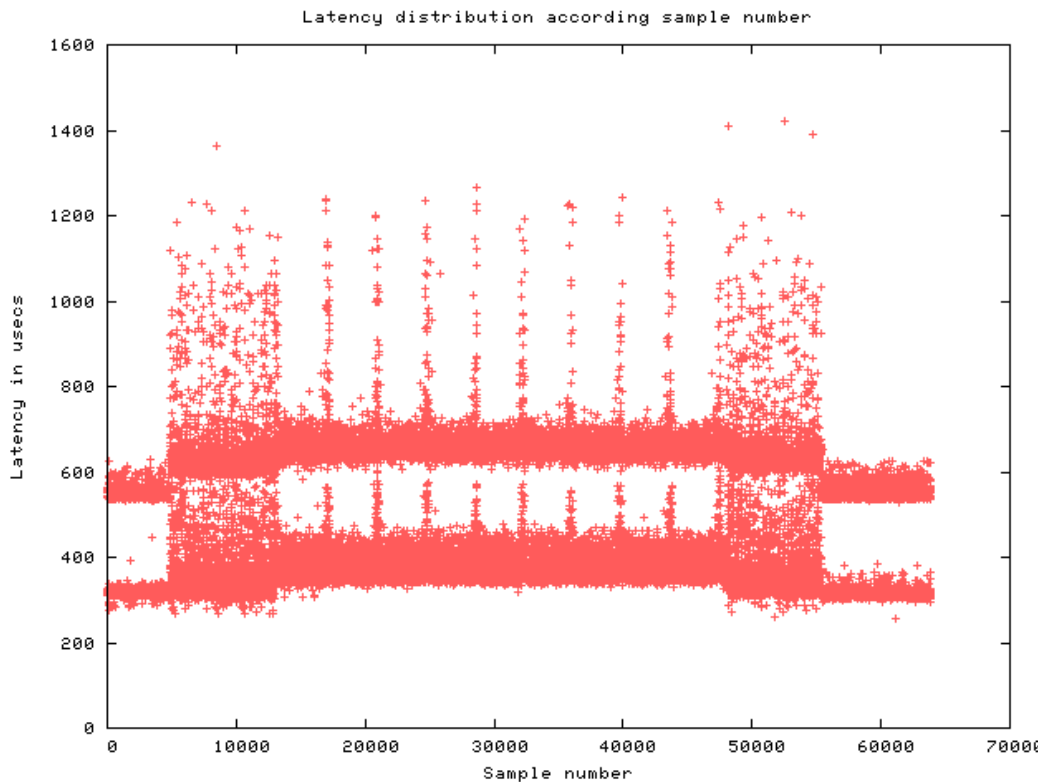


Figure 7 : Mesure du temps de latence au cours du temps avec combinaisons

Nous retrouvons les différentes phases temporelles du test avec les points hors épure à la création/destruction de threads.

Nous obtenons la courbe suivante pour l'histogramme correspondant à la mesure précédente :

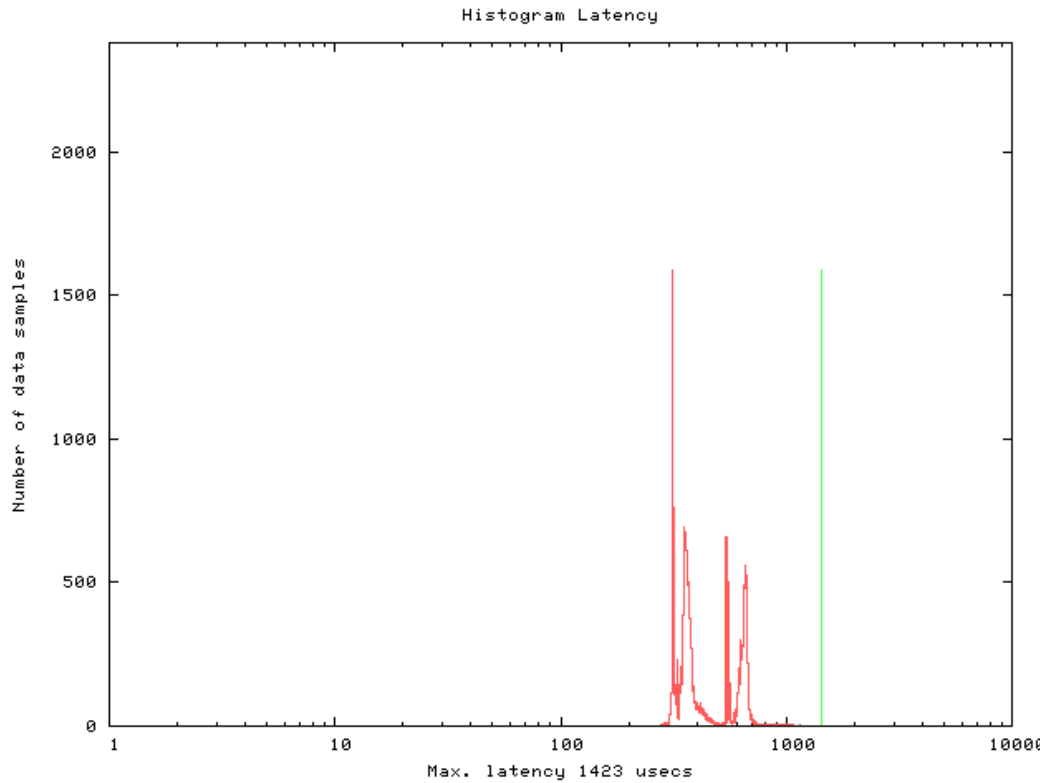


Figure 8 : Histogramme du temps de latence avec combinaisons

4.4. Mesures avec stress

Nous avons fait des mesures avec `stress` suivant le protocole suivant :

- Lancement de `cyclictest` avec un timer périodique de 10000 μ s et génération d'un fichier de mesure.
- Attente de quelques dizaines de secondes.
- Lancement de `stress -c 20 -i 20`.
- Attente de quelques dizaines de secondes.
- Arrêt de `stress`.
- Arrêt de `cyclictest`.

Nous obtenons la courbe suivante du temps de latence en fonction du temps (numéro d'échantillon divisé par 100 donne le temps en seconde) :

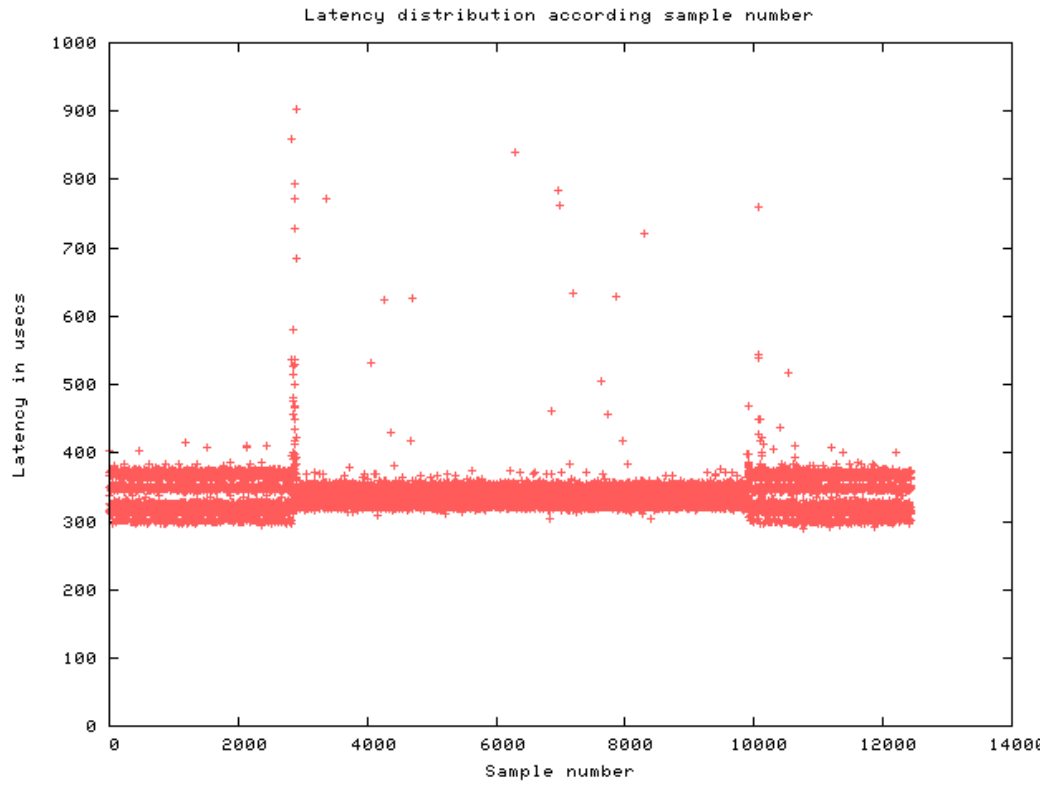


Figure 9 : Mesure du temps de latence au cours du temps avec stress

Si l'on fait une autre mesure spécifique durant l'exécution de stress, nous obtenons la courbe suivante du temps de latence en fonction du temps (numéro d'échantillon divisé par 100 donne le temps en seconde) :

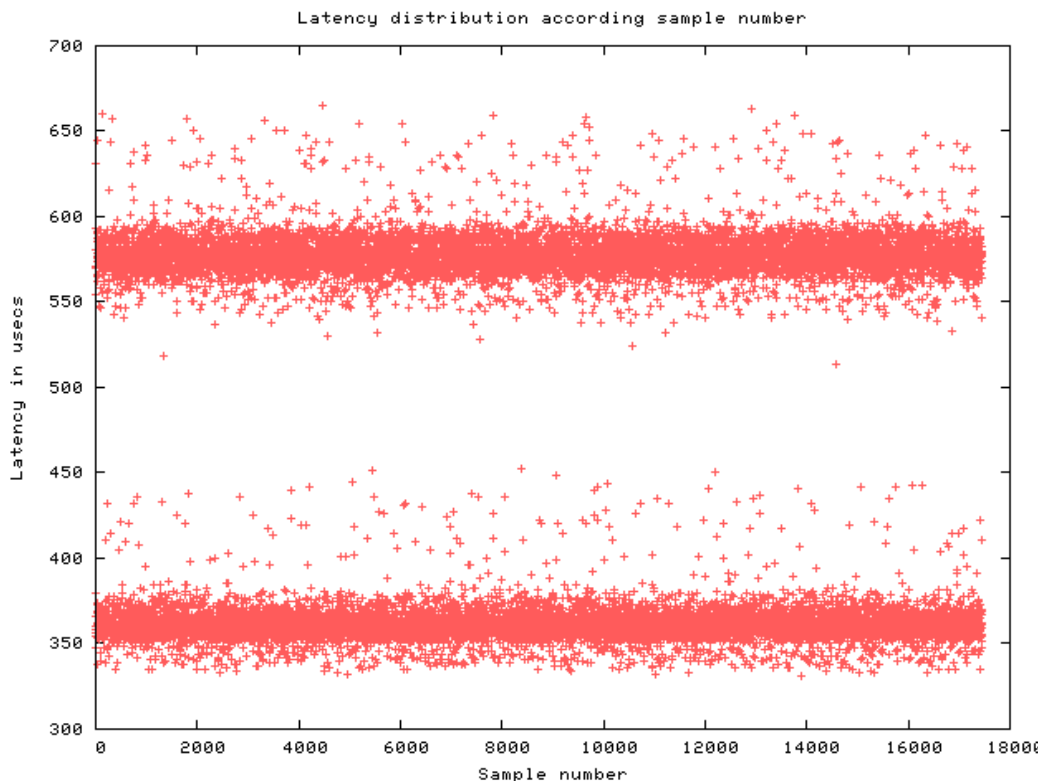


Figure 10 : Mesure du temps de latence au cours du temps avec stress

Le temps de latence reste bien confiné autour de 700 μ s.

Nous avons aussi fait des mesures sur plus de 24 heures avec cyclicttest et en stressant le système avec stress comme suit :

```
# stress -c 60 -i 60
```

Nous avons alors fait varier la valeur de la période de cyclicttest (10000 μ s, 2000 μ s, 1000 μ s) :

```
root:/> cyclicttest -n -p 99 -i 10000
policy: fifo: loadavg: 121.96 121.64 121.48 62/153 772
T: 0 ( 772) P:99 I:10000 C:1055332 Min: 266 Act: 356 Avg: 401 Max: 624
```

```
root:/> cyclicttest -n -p 99 -i 2000
policy: fifo: loadavg: 121.16 121.10 121.13 62/151 772
T: 0 ( 772) P:99 I:2000 C:21751276 Min: 255 Act: 570 Avg: 449 Max: 812
```

```
root:/> cyclicttest -n -p 99 -i 1000
policy: fifo: loadavg: 121.18 121.11 121.07 62/151 787
T: 0 ( 787) P:99 I:1000 C:6584472 Min: 92 Act: 297 Avg: 383 Max: 982
```

Le temps de latence maximum reste confine de 624 μ s pour une période de 10000 μ s à 982 μ s pour une période de 1000 μ s. Ce dernier cas génère beaucoup d'interruptions du timer de *clock event*.

4.5. Mesures à l'oscilloscope

Pour utiliser le test du *ping flooding*, nous avons utilisé la carte Stratix 1S10 qui a une interface Ethernet supportée par Linux.

Différents tests ont été effectués en utilisant les programmes *nanosleep*, *stress* et le *ping flooding*. Il s'agit de mesurer le temps de latence à l'oscilloscope.

Comme avant, le temps CPU est alloué à 100 % aux threads Temps Réel :

```
# echo -1 > /proc/sys/kernel/sched_rt_runtime_us
```

Pour générer le signal périodique mesuré à l'oscilloscope, on utilise le programme *nanosleep*, pour une période de 10000 μ s et un thread Temps Réel de priorité 99 :

```
# nanosleep 10000 99
```

Au repos, nous obtenons la figure suivante :

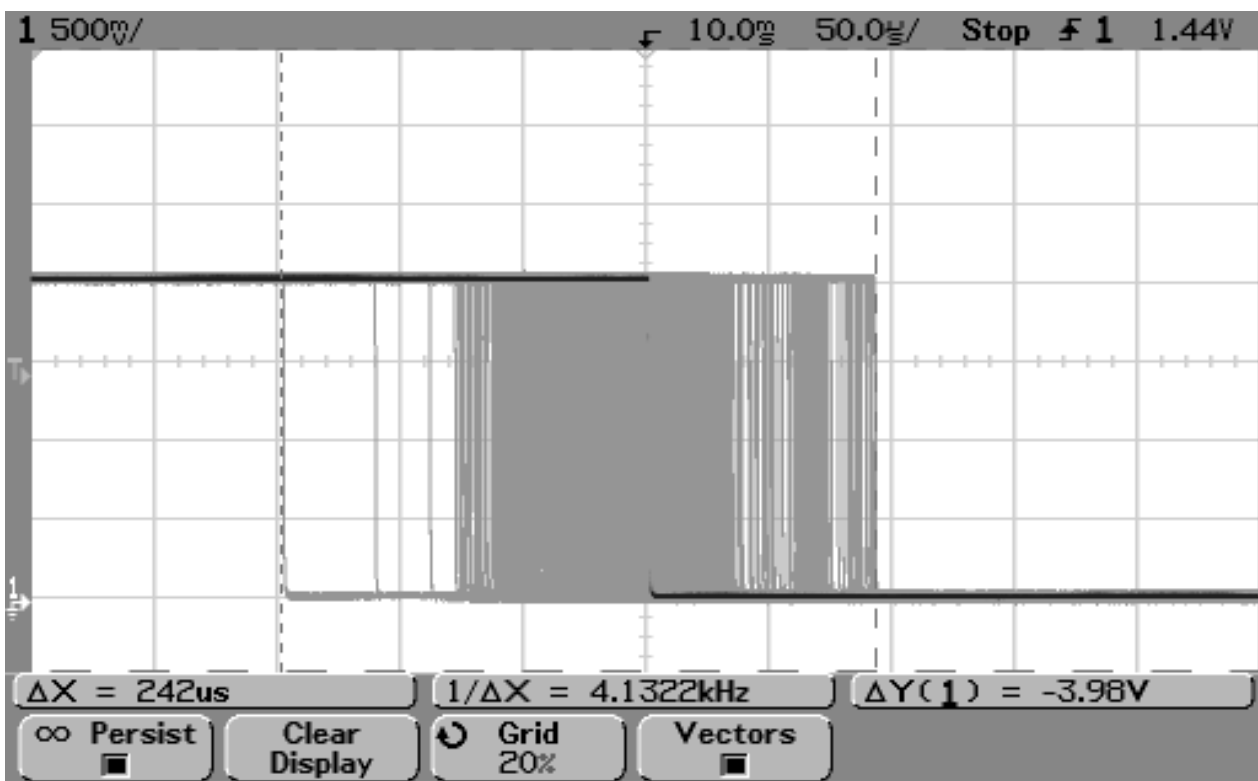


Figure 11 : Temps de latence au repos

Le temps de latence mesuré est de 242 μ s.

La carte cible est ensuite chargée par le *ping flooding*. Nous obtenons la figure suivante :

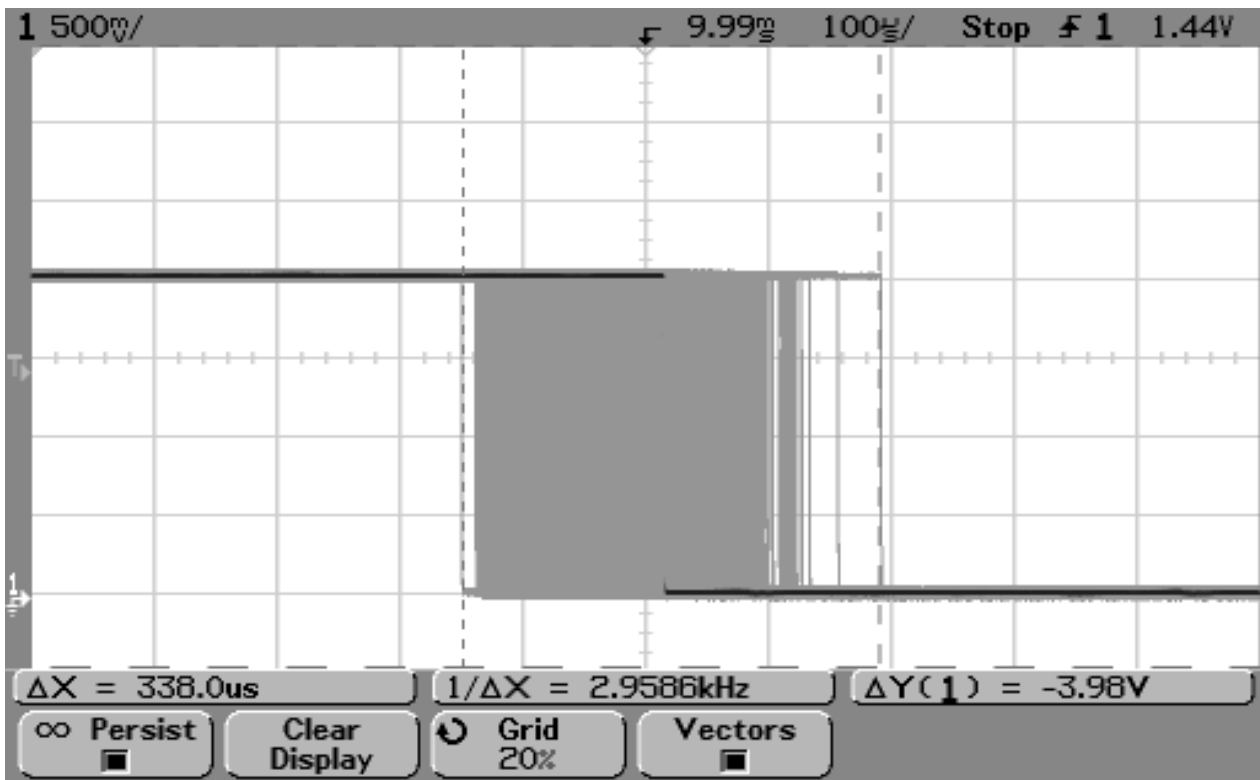


Figure 12 : Temps de latence en charge par *ping flooding*

Le temps de latence mesuré est de 338 μs .

La carte est ensuite chargée par le programme stress uniquement :

```
# stress -c 10 -i 10
```

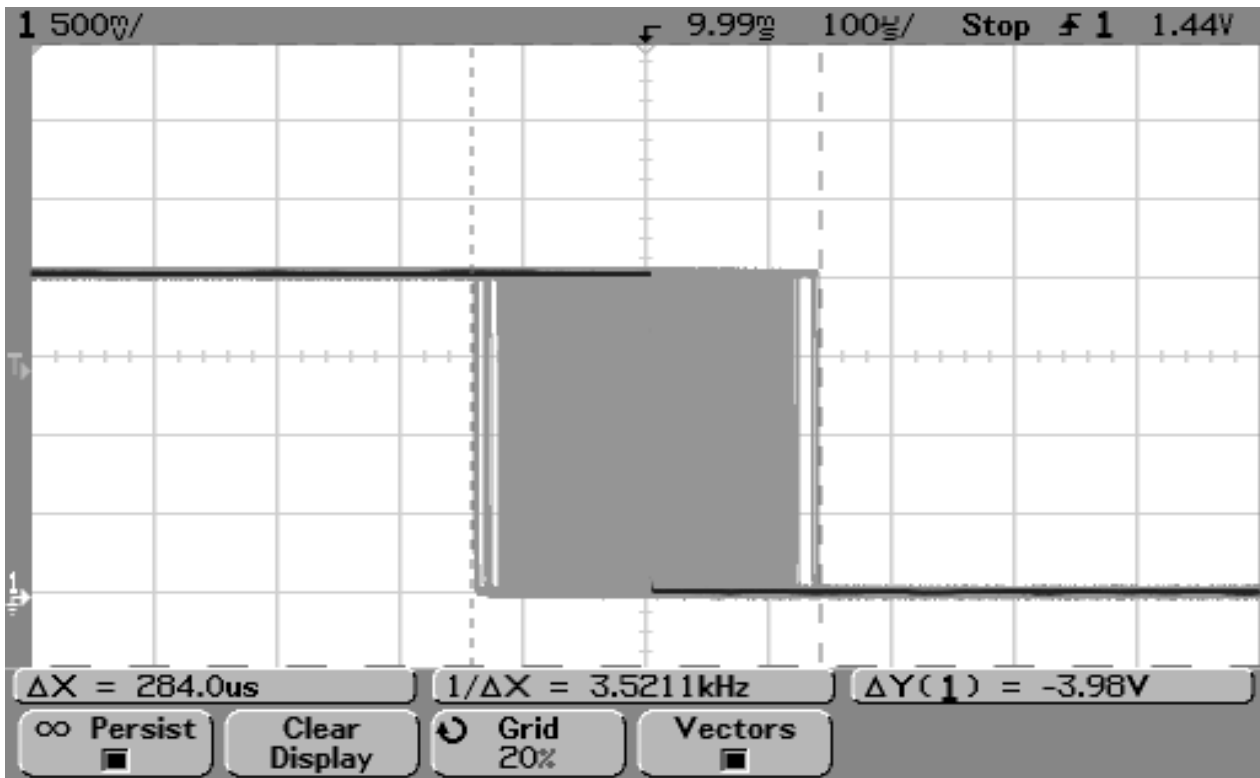


Figure 13 : Temps de latence en charge par stress

Le temps de latence mesuré est de 284 μs.

La carte est ensuite chargée par le programme `stress` et avec le *ping flooding* :

```
# stress -c 10 -i 10
```

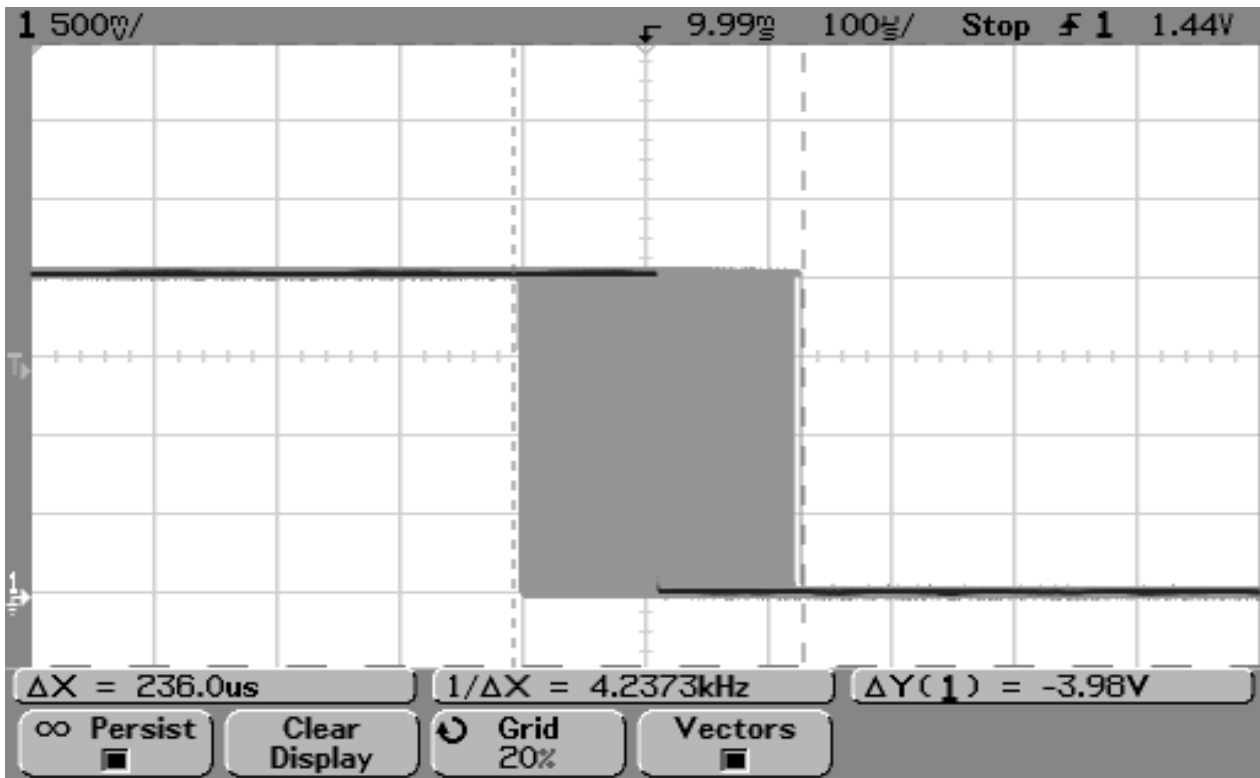


Figure 14 : Temps de latence en charge par `stress` plus *ping flooding*

Le temps de latence mesuré est du même ordre que précédemment, soit de 236 µs.

Enfin, la carte est chargée par le shell script godd :

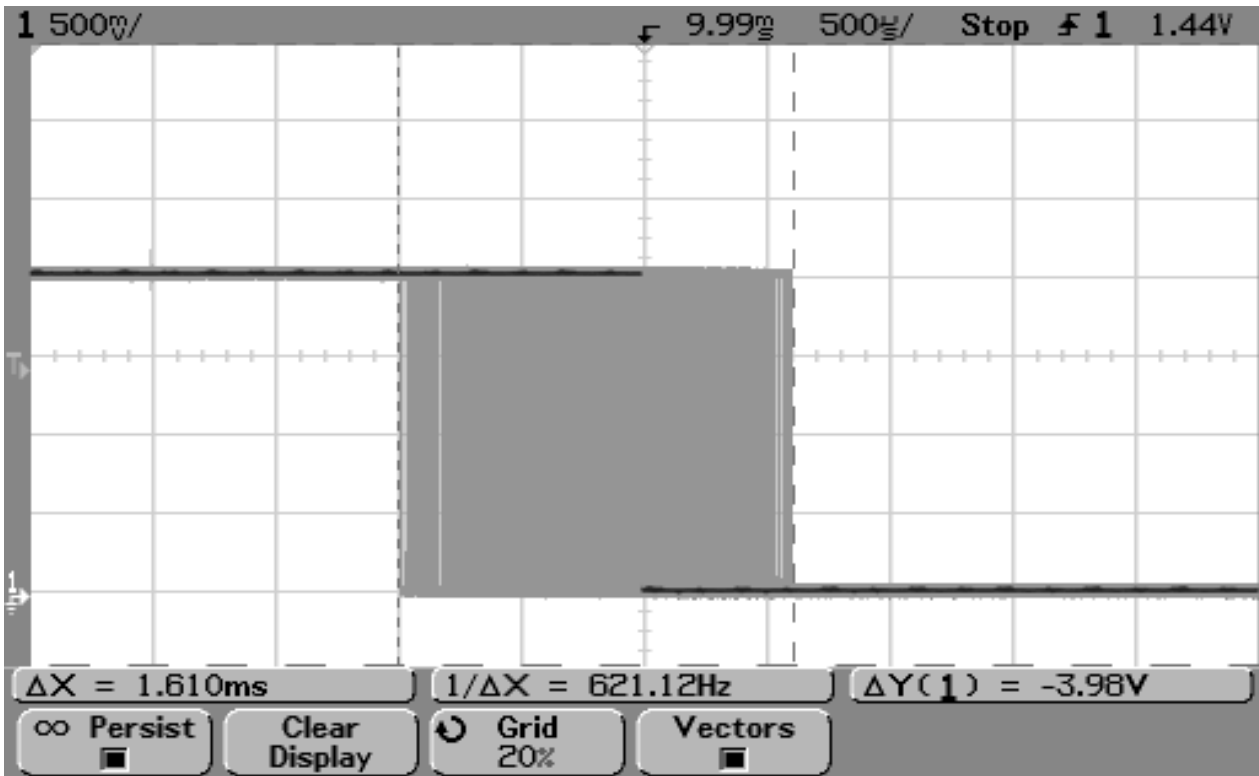


Figure 15 : Temps de latence en charge par godd

Le temps de latence mesuré est de 1610 μs .

5. RÉFÉRENCES

- [1] Outil `stress` : <http://weather.ou.edu/~apw/projects/stress/>
- [2] Outil `hackbench` : <http://devresources.linuxfoundation.org/craiger/hackbench/>
- [3] Outil `cyclictest` : <https://rt.wiki.kernel.org/index.php/Cyclictest>
- [4] Real-Time group scheduling : <http://www.kernel.org/doc/Documentation/scheduler/sched-rt-group.txt>
- [5] Interrupts considered harmful. Peter Chubb et Yang Song.
<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.156.9914&rep=rep1&type=pdf>
- [6] Real-Time Failure. Frank Rowand. http://elinux.org/images/b/be/Real_time_linux_failure.pdf
- [7] Threaded IRQs on Linux PREEMPT-RT. Luís Henriques. <http://www.artist-embedded.org/docs/Events/2009/OSP/OSP09-Henriques.pdf>
- [8] Adventures in Real-Time Performance Tuning. Frank Rowand.
http://elinux.org/images/9/99/Mips_real_time.pdf
- [9] Open Source industrial software: more hype or a new, better way? Industrial Ethernet Book.
<https://www.osadl.org/fileadmin/dam/press/Industrial-Ethernet-Book-2009-05.pdf>
- [10] Investigating latency effects of the Linux real-time Preemption Patches (PREEMPT RT) on AMD's GEODE LX Platform. Kushal Koolwal.
<http://lwn.net/images/conf/rtlws11/papers/proc/p19.pdf>

6. ANNEXE 1 : SHELL SCRIPT GODD

```
while [ 1 ]
do
    dd if=/dev/zero of=toto bs=1M count=2
    rm toto
done
```

7. ANNEXE 2 : SHELL SCRIPT GOS

```
stress -c 2 -i 2
```

8. ANNEXE 3 : SHELL SCRIPT GOH

```
while [ 1 ]
do
    hackbench 1
done
```

9. ANNEXE 4 : SHELL SCRIPT GOPERF

```
#!/bin/sh

cyclicttest -n -p 99 -i 10000 -v > 1.log &
echo "sleep 5..."
sleep 5

echo "hackbench 1..."
hackbench 1

echo "sleep 5..."
sleep 5
killall cyclicttest
```

10. ANNEXE 5 : FICHER SOURCE NANOSLEEP DE GÉNÉRATION D'UN SIGNAL PÉRIODIQUE

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <signal.h>
#include <sys/mman.h>
#include <time.h>
#include <sched.h>
#include <string.h>

#define MY_PRIORITY (90)
#define MAX_SAFE_STACK (8*1024)
#define NSEC_PER_SEC (1000000000) /* The number of nsecs per sec. */

void stack_prefault(void) {
    unsigned char dummy[MAX_SAFE_STACK];

    memset(&dummy, 0, MAX_SAFE_STACK);
}

static inline void tsnorm(struct timespec *ts)
{
    while (ts->tv_nsec >= NSEC_PER_SEC) {
        ts->tv_nsec -= NSEC_PER_SEC;
        ts->tv_sec++;
    }
}

int main(int argc, char **argv) {
    int fd;
    char value;
    int period;
    int priority;
    struct sched_param param;
    struct timespec t;

    if(argc != 3) {
        printf("%% nanosleep period_in_us priority\n");
        exit(-1);
    }

    period = atoi(argv[1]);
    priority = atoi(argv[2]);
    value = 0;

    printf("nanosleep: period=%d us, priority=%d\n", period, priority);

    /* Declare ourself as a real time task */
    param.sched_priority = priority;
```

```
if(sched_setscheduler(0, SCHED_FIFO, &param) == -1) {
    perror("sched_setscheduler failed");
    exit(-1);
}

/* Lock memory */
if(mlockall(MCL_CURRENT|MCL_FUTURE) == -1) {
    perror("mlockall failed");
    exit(-2);
}

/* Pre-fault our stack */
stack_prefault();

fd=open("/dev/leds", O_RDWR | O_SYNC);
if(fd < 0) {
    printf("Failed to open /dev/leds\n");
    exit(-1);
}

clock_gettime(CLOCK_MONOTONIC ,&t);
/* start after one second */
t.tv_sec++;

while(1) {

/* wait untill next shot */
    clock_nanosleep(0, TIMER_ABSTIME, &t, NULL);

    write(fd, &value, 1);
    value = ~value;

/* calculate next shot in ns */
    t.tv_nsec += (period *1000);
    tsnorm(&t);
}

close(fd);
exit(0);
}
```

Usage :

Génération d'un signal périodique de 50 ms avec un processus de priorité Temps Réel PREEMPT-RT de priorité la plus forte :

```
$ nanosleep 50000 99
```

11. ANNEXE 6 : PILOTE DE PÉRIPHÉRIQUE POUR ACCÉDER AUX LEDS DE LA CARTE CIBLE STRATIX 1S10

Pour observer le signal électrique, un oscilloscope est branché sur une des 8 leds de la carte.

```
#include <linux/kernel.h>    /* We're doing kernel work */
#include <linux/module.h>    /* Specifically, a module */
#include <linux/fs.h>
#include <asm/uaccess.h>    /* for get_user and put_user */

MODULE_AUTHOR("Patrice Kadionik");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Module for led access");
MODULE_SUPPORTED_DEVICE("none");

#include <asm/custom_fpga.h>
#define LED_BASE LED_PIO_BASE // 0x810880

#define MAJOR_NUM            100
#define DEVICE_FILE_NAME    "leds"
#define DEVICE_NAME         "leds"

#undef DEBUG
#undef DEBUG2

static int Device_Open = 0;
static volatile char *ptr;
static char buf[1];

static int device_open(struct inode *inode, struct file *file)
{
#ifdef DEBUG
    printk(KERN_ALERT "device_open(%p)\n", file);
#endif

    if (Device_Open)
        return -EBUSY;

    Device_Open++;

    try_module_get(THIS_MODULE);
    return 0;
}

static int device_release(struct inode *inode, struct file *file)
{
#ifdef DEBUG
    printk(KERN_ALERT "device_release(%p,%p)\n", inode, file);
#endif

    Device_Open--;

    module_put(THIS_MODULE);
}
```

```
        return 0;
    }

static ssize_t device_write(struct file *file, const char __user * buffer,
size_t count, loff_t * offset)
{
#ifdef DEBUG
    printk(KERN_ALERT "device_write(%p,%s,%d)", file, buffer, count);
#endif
    if (copy_from_user(buf, buffer, count))
        return -EFAULT;

#ifdef DEBUG2
    printk(KERN_ALERT "data=%x", buf[0]);
#endif

    local_irq_disable();
    *ptr = buf[0];
    local_irq_enable();

    return count;
}

struct file_operations Fops = {
    .read = NULL,
    .write = device_write,
    .ioctl = NULL,
    .open = device_open,
    .release = device_release,    /* a.k.a. close */
};

static int device_init(void)
{
    int ret_val;

    ptr = (char *)ioremap_nocache(LED_BASE, 1);

    ret_val = register_chrdev(MAJOR_NUM, DEVICE_NAME, &Fops);

    if (ret_val < 0) {
        printk(KERN_ALERT "%s failed with %d\n",
            "Sorry, registering the character device ", ret_val);
        return ret_val;
    }

    printk(KERN_ALERT "Leds registration success.The major device number is
%d.\n", MAJOR_NUM);
    printk(KERN_ALERT "mknod /dev/%s c %d 0\n", DEVICE_FILE_NAME, MAJOR_NUM);
    printk(KERN_ALERT "Leds remapped in VM at %x\n", ptr);

    return 0;
}

static void device_exit(void)
{
    unregister_chrdev(MAJOR_NUM, DEVICE_NAME);
}

module_init(device_init);
```

```
module_exit(device_exit);
```

Usage :

```
# mknod /dev/leds c 100 0
```

12. ANNEXE 7 : SHELL SCRIPT MAKE_HIST.SH

```
#!/bin/bash

#
# Copyright (C) 2006,2007 Luotao Fu, Pengutronix (lfu@pengutronix.de)
#
# parse the output file of cyclicttest in debug mode in plottable histogramm
# files, single files are automatically created on detecting new threads.
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License version 2 as
# published by the Free Software Foundation;
#
# Software distributed under the License is distributed on an "AS
# IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or
# implied. See the License for the specific language governing
# rights and limitations under the License.

set -e

args=$(getopt i: $*)

if [ $? -ne 0 ] || [ $# -eq 0 ];then
    echo "Usage: $(basename $0) -i inputfile"
    exit 1
fi
for i in $args; do
    case "$i" in
        -i) shift;logfile_name=$(basename $1);;
        esac
    done

logfile_name_prefix=$(echo $logfile_name | cut -d . -f 1)

step_dist=2
thread_sum=0
step_counter=0

IFS_BUF=$IFS
IFS="$IFS:"

echo -n "splitting single thread logs"

while read thread_nr loop_count m_result ;do
    echo "${loop_count} ${m_result}" >> ${logfile_name_prefix}_plot_thread${
thread_nr} ".log"
    #determine thread summary and search for min, max
    if [ ${loop_count} -eq 0 ];then
#       if [ ! $thread_nr ] || [ ! $loop_count ] || [ ! $m_result ];then
#           echo "parsing failed, check your input file"
#           exit 1
#       fi
        (( thread_sum++ ))
        if [ -e ${logfile_name_prefix}_plot_thread${thread_nr} ".log" ] ; then
            rm ${logfile_name_prefix}_plot_thread${thread_nr} ".log"
        fi
    elif [ $loop_count -eq 1 ];then
```

```
    max[thread_nr]=${m_result}
    min[thread_nr]=${m_result}
else
    if [ ${m_result} -gt ${max[thread_nr]} ];then
        max[thread_nr]=${m_result}
    elif [ ${m_result} -lt ${min[thread_nr]} ];then
        min[thread_nr]=${m_result}
    fi
fi

#now try to collect histogram data
if [ $loop_count -ne 0 ];then
    hist_index=$(( ${m_result}/${step_dist} ))
    tmp_name=hist_data_${thread_nr}[$hist_index]
    tmp_val=${!tmp_name}
    (( tmp_val++ ))
    eval ${tmp_name}=${tmp_val}
fi

#we'd give the user some lifesign every 5000 lines
if [ $( ${loop_count} / 5000 ) -ge $step_counter ]; then
    echo -n "."
    (( step_counter++ ))
fi
done < $1
echo done

# PK
# Histogramme fait avec autre script
#
exit

echo -n "making histogram files"

for ((thr_co=0; thr_co < thread_sum - 1; thr_co++)); do
    echo -n "."
    hist_array=hist_data_${thr_co}[@]
    hist_index=0;
    if [ -e ${logfile_name_prefix}_hist_thread${thr_co}.log ]; then
        rm ${logfile_name_prefix}_hist_thread${thr_co}.log"
    fi
    #hist_index_max=$(( ${max[$thr_co]}/${step_dist} ))
    hist_index_max=$(( ${max[$thr_co]}/${step_dist} ))

    for ((i=0; i < ${hist_index_max} ; i++)); do
        var_pnt=hist_data_${thr_co}[$i]
        index_value=$(( ${i} * ${step_dist} + ${min[$thr_co]} ))
        if [ ! -z ${!var_pnt} ]; then
            echo "$index_value ${!var_pnt}" >> $
{logfile_name_prefix}_hist_thread${thr_co}.log"
        fi
    done
done

echo done

IFS=$IFS_BUF
```

13. ANNEXE 8 : SHELL SCRIPT GOLATENCY

```
#!/bin/bash

defaultcyclicdata=cyclicdata

cyclicdata=$1
if test -z "$cyclicdata"
then
    cyclicdata=$defaultcyclicdata
fi

plot=latency.pbm

echo "Creating latency plots (may take a while)..."
#./make_hist.sh -i ${cyclicdata}.log 1>/dev/null 2>/dev/null
./make_hist.sh -i ${cyclicdata}.log

latency=${cyclicdata}_plot_thread0.log

echo -e "set title \"Latency distribution according sample number\"\n\
set terminal pbm color\n\
set xlabel \"Sample number\"\n\
#set logscale x\n\
#set logscale y\n\
set xrange [1:*\n\
#set xrange [1:150000]\n\
set ylabel \"Latency in usecs\"\n\
set output \"$plot\"\n\
plot \"$latency\" notitle " | gnuplot -persist

#display $plot
#gimp $plot
```

14. ANNEXE 9 : SHELL SCRIPT GOHIST

```
#!/bin/bash

defaultcyclicdata=cyclicdata.log

cyclicdata=$1
if test -z "$cyclicdata"
then
    cyclicdata=$defaultcyclicdata
fi

latencydata=latencydata
maxlatencydata=maxlatencydata
plot=histo.pbm

#echo $cyclicdata
echo "Creating histo plots (may take a while)..."

cat $cyclicdata | cut -d: -f3 | sort -n | uniq -c | sed 's/^[ ]*//g' >
latencydata
MAXIMUM_COUNT=`cat $latencydata | awk '{print $1}' | sort -n -u | tail -1`
MAXIMUM=`tail -1 $latencydata | awk '{print $2}'`
echo MAXIMUM=$MAXIMUM

if [ -z "$(head -1 $latencydata | awk '{print $2}')" ]
then
    sed --in-place 'ld' $latencydata
fi

YRANGE=$2
if test -z "$YRANGE"
then
    YRANGE=$MAXIMUM_COUNT+$MAXIMUM_COUNT/2
fi

echo -e $YRANGE\\t$MAXIMUM >$maxlatencydata

echo -e "set title \"Histogram Latency\"\\n\\
set terminal pbm color\\n\\
set xlabel \"Max. latency $MAXIMUM usecs\"\\n\\
set logscale x\\n\\
set xrange [1:*]\\n\\
set yrange [0:$YRANGE]\\n\\
set ylabel \"Number of data samples\"\\n\\
set output \"$plot\"\\n\\
plot \"$latencydata\" using 2:1 notitle with histeps, \"$maxlatencydata\" using
2:1 notitle with boxes" | gnuplot -persist

#display $plot
#gimp $plot
```