

- PROJET RTEL4I -



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



## SOMMAIRE

<b>1. Carte cible.....</b>	<b>4</b>
<b>2. Programmes utilisés.....</b>	<b>5</b>
2.1. Ping flooding.....	5
2.2. Script godd.....	5
2.3. Programme stress.....	5
2.4. Script gos.....	5
2.5. Programme hackbench.....	6
2.6. Script goh.....	6
2.7. Script goperf.....	6
2.8. Programme cyclictest.....	6
2.9. Script make_hist.sh.....	6
2.10. Script golatency.....	7
2.11. Script gohist.....	7
<b>3. Protocoles de test.....</b>	<b>7</b>
<b>4. Résultats.....</b>	<b>8</b>
4.1. Mesures avec goperf.....	8
4.2. Mesures avec godd.....	10
4.3. Mesures avec stress.....	12
<b>5. Références.....</b>	<b>14</b>
<b>6. Annexe 1 : shell script godd.....</b>	<b>15</b>
<b>7. Annexe 2 : shell script gos.....</b>	<b>15</b>
<b>8. Annexe 3 : shell script goh.....</b>	<b>15</b>
<b>9. Annexe 4 : shell script goperf.....</b>	<b>15</b>
<b>10. Annexe 5 : shell script make_hist.sh.....</b>	<b>16</b>
<b>11. Annexe 6 : shell script golatency.....</b>	<b>18</b>
<b>12. Annexe 7 : shell script gohist.....</b>	<b>19</b>

## Fiche synthétique

<b>Matériel</b>	
Processeur cible	MicroBlaze avec MMU
Fondeur	Xilinx
Outils de synthèse	ISE version 11.4 et supérieur
Carte cible	Xilinx ML403
<b>Logiciel</b>	
Distribution Linux	Fedora 13
Version Linux pour MicroBlaze	Linux 2.6.31
Version compilateur croisé pour MicroBlaze	
Version patch pour MicroBlaze	preempt-rt-2.6.31.12-rt21-microblaze-1.0-00.patch
Version patch PREEMPT-RT pour Linux	2.6.31.12-rt21

### Gestion du document :

Date	Version	Modifications	Relecture	Commentaires
06/09/10	1.0	PK	PK	Création fichier

## 1. CARTE CIBLE

La carte choisie est une carte d'évaluation moyenne gamme de Xilinx : carte Virtex-4 ML403.

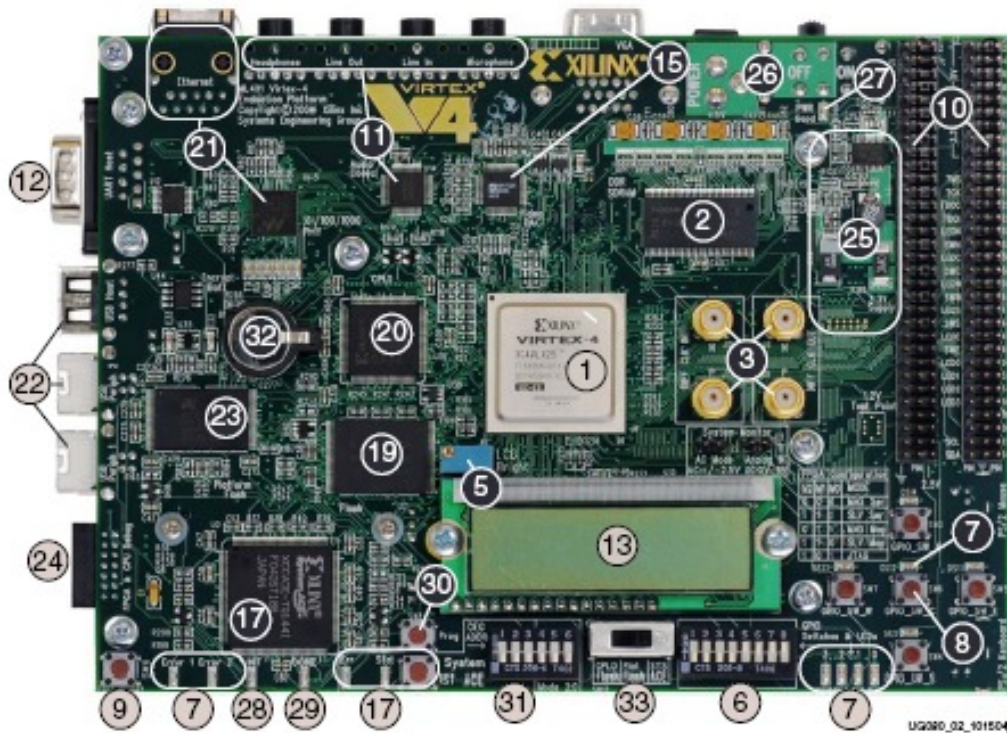


Figure 1 : Carte cible Xilinx Virtex-4 ML403

La carte ML403 possède les caractéristiques suivantes :

- Circuit FPGA XC4VFX12-FF668-10.
- 8 Mo de SRAM 32 bits.
- 64 Mo de SDRAM DDR32 bits.
- 8 Mo de mémoire Flash.
- Oscillateur à 100 MHz.
- 1 support CompactFlash type I par interface System ACE.
- 1 interface Ethernet 10/100/1000 Mb/s.
- 1 port série (RS-232 DB9).
- 1 port d'extension pour carte fille maison.
- 1 connecteur JTAG.
- Boutons poussoirs.
- Leds utilisateurs.
- 1 afficheur LCD 2x16 caractères.
- Bus I2C.
- 4Kb d'EEPROM SPI.
- Bus USB.
- Sortie audio AC97.
- Connecteurs PS2 clavier et souris.
- Sortie VGA.

## 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*.

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

- 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  $\mu$ 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  $\mu$ s avec la priorité Temps Réel la plus forte de 99 :  
`# cyclictest -n -p 99 -i 10000`

Exemple d'usage : construction d'un fichier de mesure du temps de latence pour un timer périodique de 10000  $\mu$ s avec la priorité Temps Réel la plus forte de 99 :  
`# cyclictest -n -p 99 -i 10000 -v > cyclicdata.log`

## 2.9. Script `make_hist.sh`

Le shell script `make_hist.sh` donné en annexe 5 permet de trier les données du fichier de mesure généré avec `cyclictest` en vue de construire ensuite un graphique de temps de latence

## **2.10. Script golatency**

Le shell script `golatency` donné en annexe 6 permet de construire un graphique de temps de latence à partir des données générées par `make_hist.sh`.

## **2.11. Script gohist**

Le shell script `gohist` donné en annexe 7 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 ML403.

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

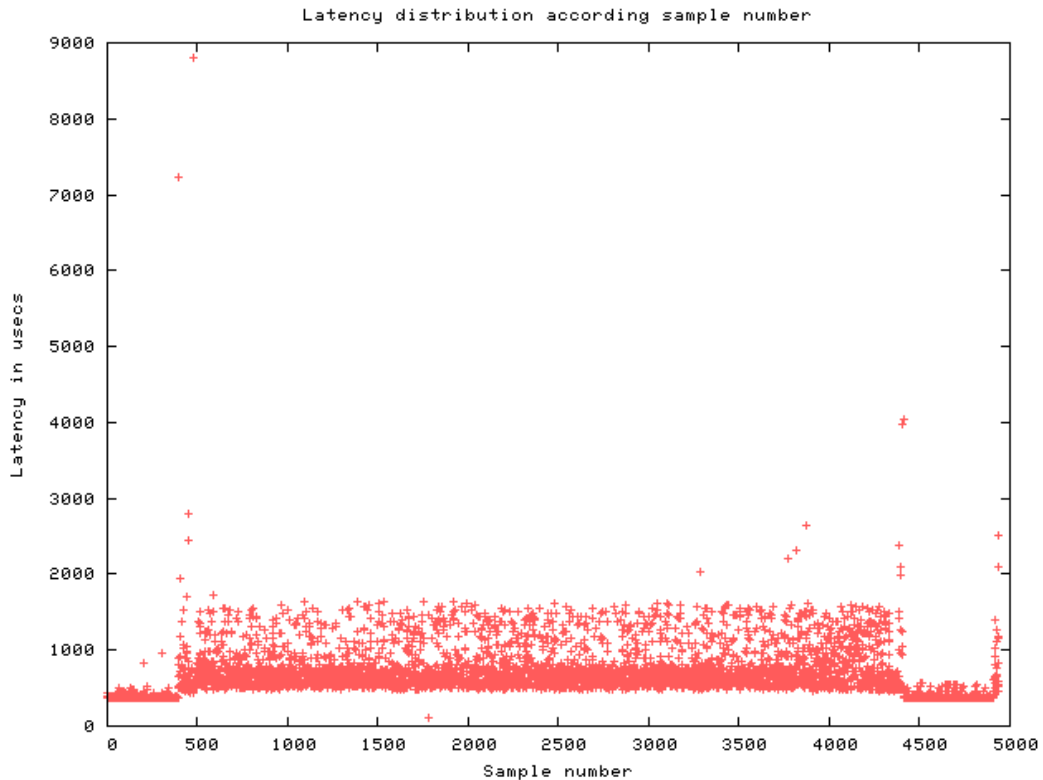


Figure 2 : Mesure du temps de latence au cours du temps avec goperf

On retrouve les paliers de repos de 5 secondes et la charge avec `hackbench`. On a un temps de latence autour de 400  $\mu$ s au repos, puis de 1500  $\mu$ s en charge avec des pointes maximales à 8000  $\mu$ s au lancement et à la terminaison de `hackbench`.

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

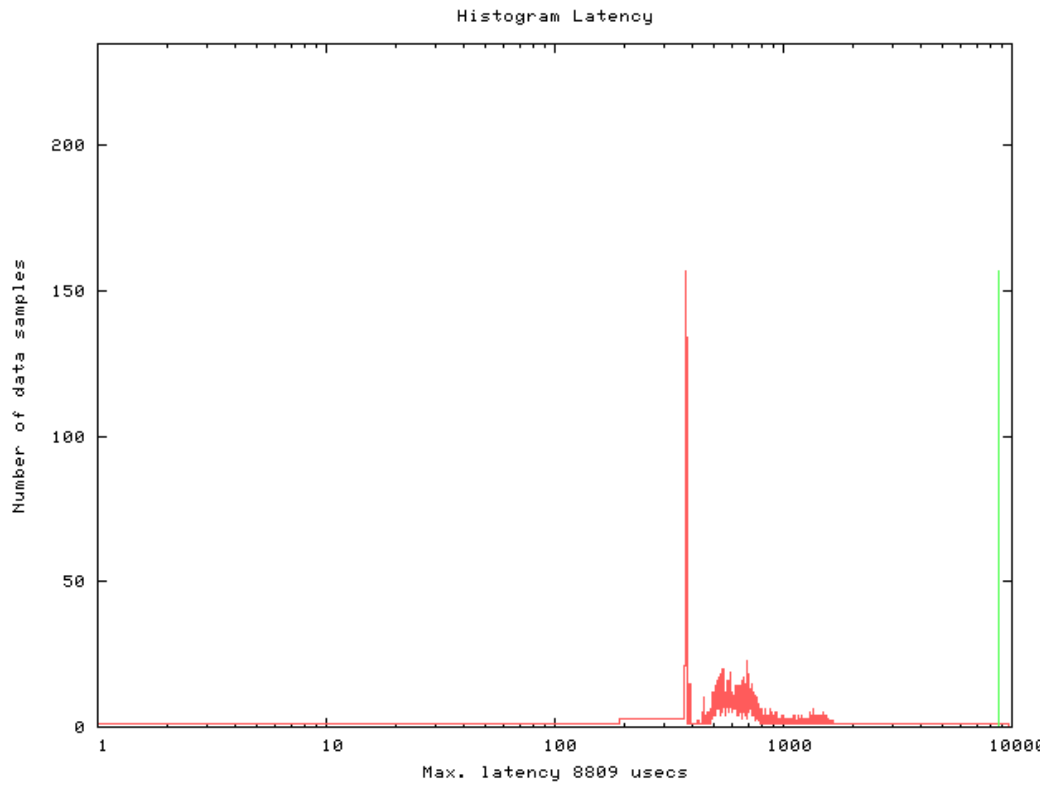


Figure 3 : Histogramme du temps de latence avec goperf

## 4.2. Mesures avec godd

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

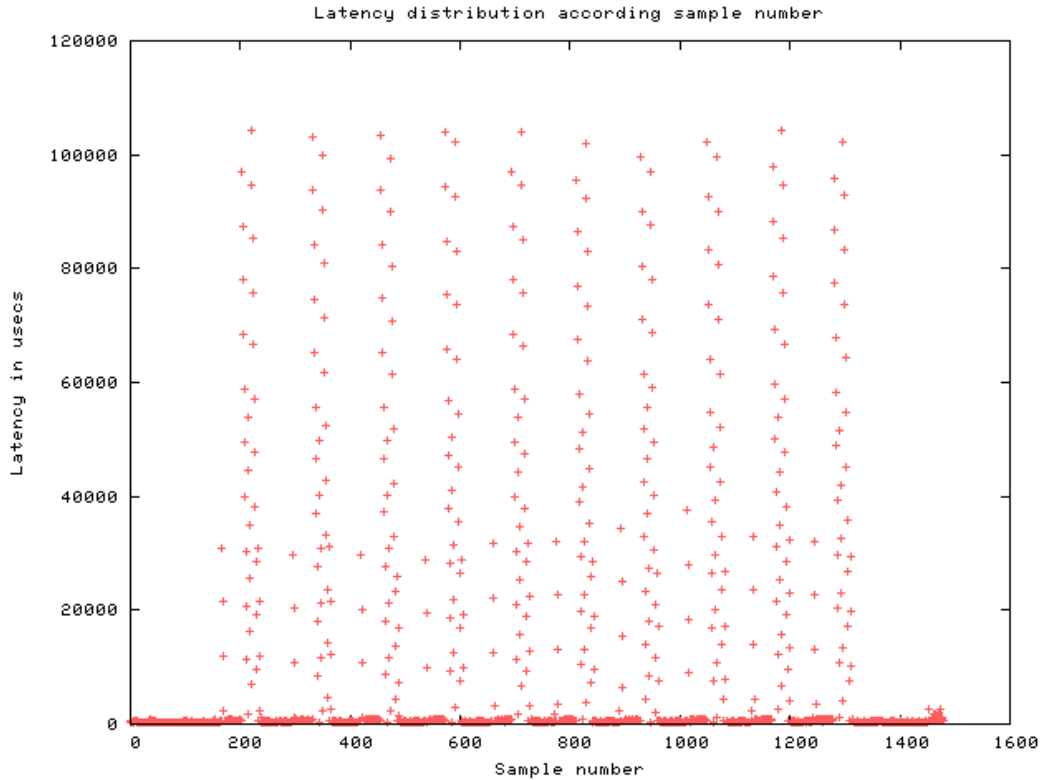


Figure 4 : Mesure du temps de latence au cours du temps avec godd

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 100000  $\mu$ s (100 ms) ! Le temps de latence reste ensuite confiné.

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

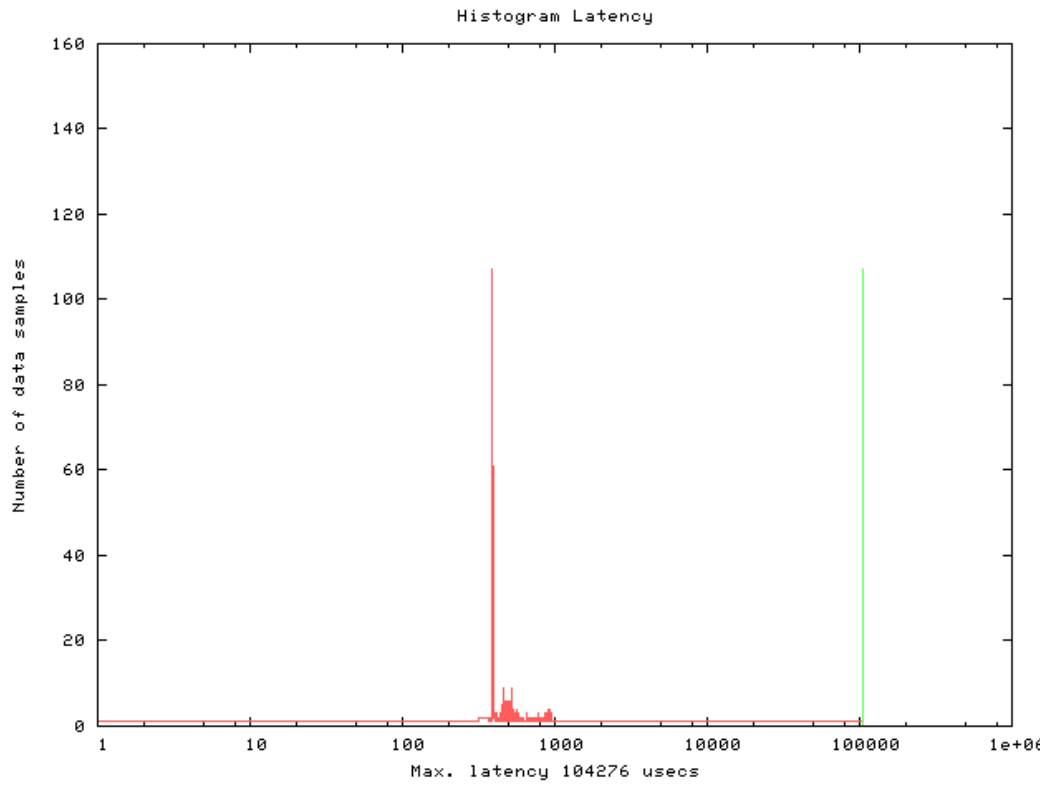


Figure 5 : Histogramme du temps de latence avec godd

### 4.3. Mesures avec stress

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

- Lancement de `cyclictest` avec un timer périodique de 10000  $\mu$ 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) :

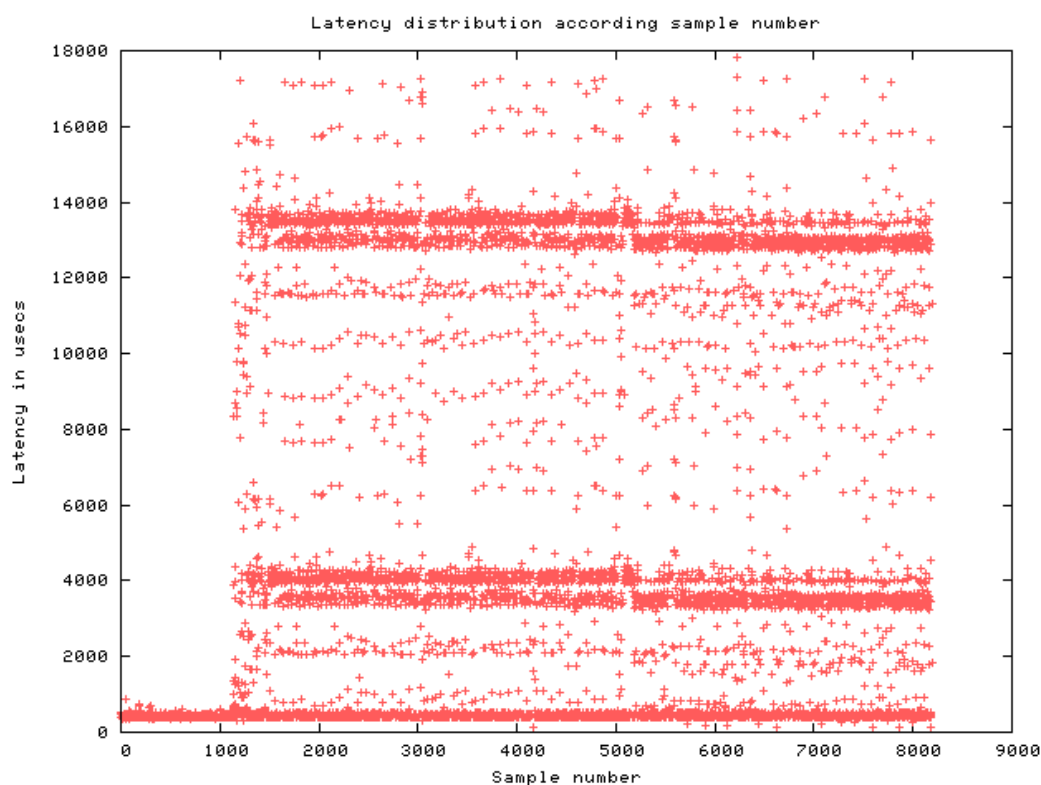


Figure 6 : Mesure du temps de latence au cours du temps avec `stress`

Le temps de latence reste a des pointes jusqu'à 17000  $\mu$ s.

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

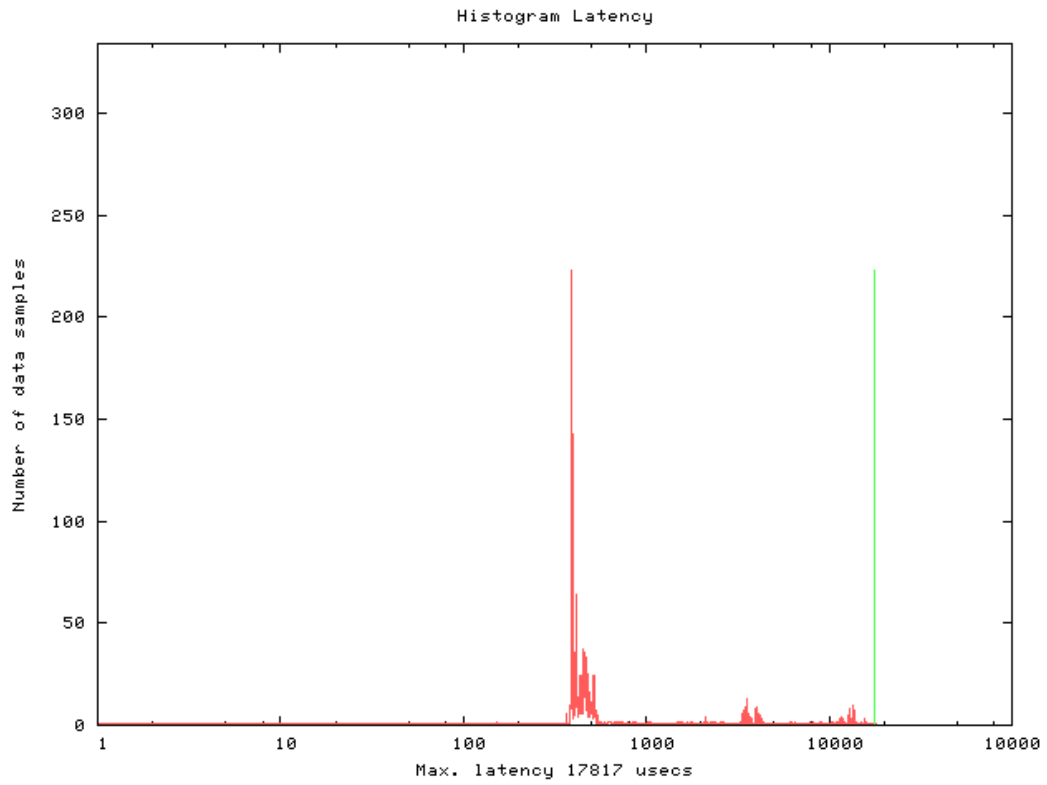


Figure 7 : Mesure du temps de latence au cours du temps avec stress

## 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](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](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 : 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
```

## 11. ANNEXE 6 : 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
```

## 12. ANNEXE 7 : 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
```