Raspberry Pi et temps réel*

Christophe BLAESS

La notion de temps réel est source de nombreuses controverses dans le monde du développement informatique. Je commencerai donc par qualifier les différentes classes de systèmes temps réel que l'on considère habituellement. Nous pourrons alors voir quelles sont les différentes solutions utilisables pour le Raspberry Pi et les niveaux de qualité que nous pouvons en attendre.

Concept de temps réel

La notion de temps réel fait référence à la possibilité pour un système de répondre à des stimuli extérieurs en prenant en compte précisément l'écoulement du temps, ceci indépendamment du flux d'instructions traité par le processeur. J'ai l'habitude de présenter ce concept en disant qu'un système est soumis à des contraintes temps réel si, lorsqu'il répond à un événement extérieur, l'instant auquel il parvient à fournir sa réponse est à prendre en considération dans la validité de celle-ci.

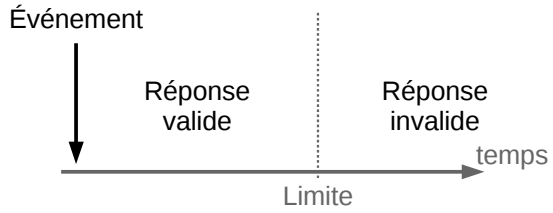

Fig. 1 – Notion de contrainte temps réel

Les polémiques habituelles sur le temps réel tiennent essentiellement à la précision de la limite et aux tolérances qu'on accorde lors de sa prise en compte. Face à la multitude et la diversité de systèmes prétendant répondre aux critères du temps réel, j'ai choisi de les caractériser en définissant quatre catégories là où généralement on n'en prend que deux en considération (*hard realtime* et *soft realtime*).

Temps réel absolu

Dans un système temps réel **absolu**, les temps de réponse aux stimuli externes sont parfaitement connus et stables. Lorsqu'un événement se produit, le système saura y apporter une réponse dans un temps toujours identique, parfaitement **prédictible**, avec des fluctuations infimes (liées aux variations thermiques des composants par exemple) de l'ordre de la nanoseconde ou dizaine de nanosecondes.

Il s'agit du temps réel que peuvent proposer des systèmes entièrement électroniques sans aspects logiciels. Toute la logique est câblée et les temps de réponse sont des durées de propagation de signaux et de commutations électriques au sein des composants.

Inutile d'espérer atteindre ce niveau de prédictibilité dans les temps de réponse du Raspberry Pi, ni d'ailleurs dans ceux d'autres cartes à microprocesseur.

Les seuls systèmes programmables et modifiables susceptibles de répondre à ce type de contrainte temps réels sont les FPGA, CPLD, PAL, etc. qui implémentent des arrangements configurables de composants logiques.

Temps réel strict certifiable

Avec un système apte à répondre à des contraintes de temps réel **strict** (*hard realtime*) **certifiables** (on parle également de temps réel **dur**), de légères fluctuations (de l'ordre de la centaine de nanosecondes ou de la microseconde) pourront intervenir dans les temps de réponses aux événements extérieurs. Ces fluctuations seront très faibles par rapport aux durées considérées, et elles seront **bornées**. S'il n'est pas possible de prédire exactement un temps de réponse absolu, on peut néanmoins savoir qu'elle sera sa valeur maximale. On peut, à l'étude du code applicatif connaître la durée maximale d'exécution d'une portion

* Cet article est paru dans Gnu/Linux Magazine Hors Série numéro 75 « *Raspberry Pi Avancé* » en novembre 2014, disponible sur `https://boutique.ed-diamond.com/anciens-numeros/789-gnulinux-magazine-hs-75.html`

de code, quoiqu'il se passe en dehors du système. C'est cet aspect qui permet à ce type d'application d'être employée dans des environnements où des certifications de sécurité sont réclamées (avionique, ferroviaire, automobile, médical, etc.)

Les systèmes répondant aux contraintes du temps réel strict certifiable sont généralement conçus en employant des microcontrôleurs. Ces systèmes sont mono-tâches ou ne comportent qu'un nombre limité (et figé dès la conception) de tâches.

La possibilité de réaliser des systèmes temps réel strict en employant des microprocesseurs modernes est sujet à controverse. De nombreux sous-systèmes de ces microprocesseurs limitent la prévisibilité des temps de réponse : MMU, caches mémoire, pipelines d'instructions, protocoles de synchronisation de caches dans les systèmes multicœurs, réduction de la consommation d'énergie, etc.

Il existe quelques systèmes d'exploitation visant ce type de performances. Pour la plupart ils fonctionnent sur des microcontrôleurs et ne fournissent qu'un minimum de fonctionnalités, essentiellement liées au multitâche. Ces systèmes (dont le nom tourne souvent autour de **RTOS** *Real-Time Operating System*) sont généralement propriétaires. On notera que la notion de système d'exploitation est limitée ici à son strict minimum, l'essentiel de la gestion des ressources (mémoire, périphériques, communications) étant statique, définie à l'initialisation et figée pour la suite.

Dans les systèmes temps réel strict certifiables capables de tirer parti des fonctionnalités d'un microprocesseur, nous pouvons citer **RTEMS** (*Real Time Executive for Multiprocessor Systems*). Ce dernier a été développé initialement pour l'armée américaine (ce qui explique que le 'M' de son acronyme ait représenté successivement les mots '*Missile*' puis '*Military*') et est distribué aujourd'hui librement sous une licence dérivée de la GPL.

On peut faire fonctionner RTEMS sur un Raspberry Pi même si cette plate-forme n'est pas complètement supportée. On se reportera pour en savoir plus à l'article de Pierre Ficheux dans ce même numéro.

Temps réel strict non-certifiable

Les supports temps réel **stricts non-certifiables** sont des implémentations, dans de véritables systèmes d'exploitation, de mécanismes d'ordonnancement visant à **approcher** au mieux le temps réel strict.

Les systèmes considérés étant généralement multitâches (avec des tâches indépendantes les unes des autres), voire multi-utilisateurs, des mécanismes d'isolation mémoire (MMU) et de communication entre tâches sont proposés par le système.

La plupart du temps il n'y a pas de vraies limites garanties pour les temps de réponse. De nombreux systèmes fonctionnent avec des notions de priorités entre tâches et garantissent qu'un traitement de priorité élevée ne sera pas interrompu par une tâche de priorité moindre.

Le noyau temps réel du système d'exploitation est prévu pour être le plus déterministe possible dans ses traitements. Les gestionnaires d'interruptions utilisés par le système sont les plus brefs possibles.

Ce niveau de temps réel peut être obtenu sur un Raspberry Pi en utilisant **Xenomai** comme nous le verrons plus loin. Le *patch* **`PREEMPT_RT`** dont nous parlerons également permet d'approcher ce type de performances.

Sur une plate-forme de type PC, on peut utiliser le projet **RTAI** (*Real-Time Application Interface*) qui ne supporte pas (encore) le Raspberry Pi.

Temps réel souple

Le temps réel **souple** (*soft realtime*) est une organisation sous forme de priorités entre les tâches applicatives. À chaque invocation de l'ordonnanceur, celui-ci choisit la tâche applicative dont la **priorité** est la plus élevée parmi toutes les tâches prêtes. Elle ne pourra être préemptée (interrompue) que pour laisser la place à une tâche de priorité plus élevée qui vient de se réveiller. Ce type d'ordonnancement est appelé « *Fifo* ». Avec l'ordonnancement nommé « *Round Robin* », on peut en outre avoir une rotation entre les tâches de même niveau de priorité.

Ce qui limite les performances du temps réel souple, ce sont les traitements – parfois longs comme ceux des protocoles réseaux, des systèmes de fichiers etc. – réalisés directement par le système au détriment des tâches applicatives. Sur un système Linux avec un noyau standard sans extension (celui que l'on nomme le noyau *vanilla*), un « *ping* » en provenance d'une machine distante sera immédiatement traité par le kernel (dans un gestionnaire d'interruption puis une *tasklet*) même s'il a interrompu une tâche considérée comme très prioritaire, retardant celle-ci d'autant.

Récapitulatif

En résumé, les quatre catégories que j'ai décrites ci-dessus sont caractérisées dans le tableau suivant.

Type de temps réel	**Temps de réponse**	**Exemple avec Raspberry Pi**
Absolu	Fixés.	*Non*
Strict certifiable	Fluctuants mais bornés.	*RTEMS*
Strict non-certifiable	Non garantis mais indépendants de toute activité moins prioritaire du système.	Xenomai Linux **`PREEMPT_RT`**
Souple	Non garantis mais indépendants des activités moins prioritaires en espace utilisateur.	Linux *vanilla*

Implémentations pour Raspberry Pi

Pierre Ficheux ayant traité du portage de RTEMS sur Raspberry Pi, je vais me consacrer aux trois autres solutions : le noyau Linux *vanilla*, le kernel modifié par le *patch* **`PREEMPT_RT`** et le système Xenomai. Je les traiterai par qualités croissantes du temps réel proposé.

Temps réel souple avec Linux

Le noyau *vanilla* permet de modifier la priorité et le mode d'ordonnancement d'une tâche. Ceci grâce à l'appel-système **`sched_setscheduler()`**, mais plus simplement avec la commande **`chrt`** depuis le shell.

L'appel **`chrt -f 90 ./commande`** permet de lancer la commande sous un ordonnancement *Fifo* (la tâche ne peut être préemptée que par une autre tâche de priorité strictement supérieure) avec la priorité 90 (sur une échelle allant de 1 à 99). De même **`chrt -r 50 ./commande`** lance la commande avec un ordonnancement *Round-Robin* (après un certain temps d'exécution, la tâche peut être préemptée par une autre de même priorité) et la priorité 50.

Précisons qu'il existe depuis quelques mois une autre catégorie d'ordonnancement temps réel : **EDF** (*Earliest Deadline First*) où l'on réserve une partie du temps CPU disponible pour une exécution périodique de la tâche. Cet ordonnancement est implémenté dans le noyau Linux depuis sa version 3.14 mais pas encore supporté par les utilitaires de la Raspbian. La qualité de son comportement temps réel est sensiblement identique à celle de *Fifo* et *Round-Robin*.

Lorsqu'une tâche temps réel s'exécute sans discontinuer pendant plusieurs centaines de millisecondes, le noyau est susceptible de l'interrompre quelques instants pour laisser s'exécuter des tâches non temps réel. Ceci afin d'éviter qu'une tâche ne boucle indéfiniment en bloquant tout le processeur. Ce comportement est un peu surprenant, et peut semble indésirable dans certains cas. Pour désactiver cette option du noyau, il faut exécuter (par exemple dans un script de démarrage du système) :

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

Temps réel classique avec PREEMPT_RT

Le *patch* **`PREEMPT_RT`** regroupe un ensemble de modifications que l'on peut apporter sur un noyau Linux *vanilla* pour améliorer son comportement temps réel. Par exemple il comprend un mode de fonctionnement « *Fully Preemptible* », où une tâche s'exécutant dans l'espace utilisateur est susceptible si elle se réveille suite à l'arrivée d'une interruption de préempter immédiatement une tâche moins prioritaire même si cette dernière exécutait du code dans l'espace noyau. L'amélioration consiste ici à réduire très sensiblement le délai de préemption par rapport à un noyau standard.

Une autre modification visible consiste à « *threader* » les gestionnaires d'interruption, autrement dit à les exécuter dans des tâches (avec les privilèges du kernel) dont la priorité n'est plus infinie, mais au contraire modifiable par l'administrateur avec la commande **`chrt`**. Les *threaded interrupts* sont visibles dans les résultats de la commande **`ps aux`** sous forme **`[36/irq-timer]`** indiquant le numéro et le nom de l'interruption (que l'on peut retrouver dans **`/proc/interrupts`**). Leur priorité par défaut est 50 sous un ordonnancement *Fifo*. Une tâche de priorité 51 sera donc plus prioritaire que les traitements d'interruption.

Le patch **`PREEMPT_RT`** évolue régulièrement. Comme il s'applique sur les sources du noyau *vanilla*, il est

maintenu et disponible pour la plupart des versions stables du kernel.

Approche du temps réel strict

Avec le projet **Xenomai**, on atteint les performances du temps réel strict, tout en continuant de bénéficier des services d'un système d'exploitation très riche. Xenomai s'appuie sur une couche nommée **`ipipe`** (*interrupt pipeline*) qui capture les interruptions provenant du matériel avant Linux. Les interruptions sont transmises tout d'abord à un premier « domaine d'exécution » composé d'un petit ordonnanceur simple nommé *nucleus*, et qui pourra activer les tâches temps réel de Xenomai. Lorsqu'il ne reste plus aucune tâche Xenomai active, **`ipipe`** envoie les interruptions reçues au second domaine d'exécution : le noyau Linux.

Xenomai est constitué d'un *patch* que l'on applique sur les sources du noyau *vanilla*, et de bibliothèques permettant de programmer des tâches temps réel. En outre des outils de mesure sont fournis pour juger des performances du système.

Installations

Nous allons tester sur un Raspberry Pi successivement le noyau *vanilla*, le patch **`PREEMPT_RT`** et Xenomai. Nous partons d'une distribution Raspbian, sur laquelle nous installerons successivement les noyaux modifiés compilés sur un PC.

Sélection d'une version

Pour pouvoir comparer les performances des trois systèmes, il est préférable de partir d'un noyau Linux standard de même niveau pour les trois expériences. Avec **`PREEMPT_RT`** et Xenomai ce noyau sera modifié par un *patch* que nous lui appliquerons. Il faut donc vérifier quelles sont les versions de Linux supportées par les *patches* de ces deux systèmes. **`PREEMPT_RT`** est disponible pour pratiquement toutes les versions stables de Linux, mais le *patch* **`ipipe`** pour Xenomai est plus rare. C'est donc lui que nous allons vérifier en premier.

Téléchargeons la dernière version disponible au moment de la rédaction de ces lignes :

```
[~]$ git clone http://git.xenomai.org/xenomai-2.6.git
Clonage dans 'xenomai-2.6'...
```

Les *patches* sont groupés dans un répertoire qui dépend de l'architecture. En outre il faudra ajouter ceux spécifiques pour le Raspberry Pi

```
[~]$ ls xenomai-2.6/ksrc/arch/arm/patches/
beaglebone  ipipe-core-3.4.6-arm-4.patch  ipipe-core-3.5.7-arm-6.patch
ipipe-core-3.8.13-arm-3.patch  mxc  raspberry  README  zynq
[~]$ ls xenomai-2.6/ksrc/arch/arm/patches/raspberry/
ipipe-core-3.8.13-raspberry-post-2.patch  ipipe-core-3.8.13-raspberry-pre-2.patch
```

Le support pour Raspberry Pi est donc proposé à partir de *patches* pour le noyau **3.8.13**. C'est donc la version que nous sélectionnons.

Les compilations des kernels seront réalisées sur un PC par *cross-compilation*, mais on pourrait – avec beaucoup de patience – faire le travail directement sur le Raspberry Pi avec une grosse carte SD ou un disque externe (il faut compter environ 2Go libres pour compiler le noyau Linux).

Téléchargement

Téléchargeons un premier exemplaire des sources de Linux avec le support Raspberry Pi. Cette opération dure environ une demi-heure :

```
[~]$ git clone http://github.com/raspberrypi/linux rpi-kernel-vanilla
Clonage dans 'rpi-kernel-vanilla'...
 [...]
Checking out files: 100% (44957/44957), done.
```

Le répertoire **`rpi-kernel-vanilla`** contient l'ensemble des sources avec l'historique *Git*. Sélectionnons la

bonne version du noyau :

```
[~]$ cd  rpi-kernel-vanilla/
[rpi-kernel-vanilla]$ git  checkout  rpi-3.8.y
Checking out files: 100% (23274/23274), done.
La branche rpi-3.8.y est paramétrée pour suivre la branche distante rpi-3.8.y depuis origin.
Basculement sur la nouvelle branche 'rpi-3.8.y'
```

Nous sommes sur la branche stable 3.8, vérifions la version exacte :

```
[rpi-kernel-vanilla]$ head  -3  Makefile
VERSION = 3
PATCHLEVEL = 8
SUBLEVEL = 13
```

3.8.13, c'est parfait ! Dupliquons ce répertoire en deux autres exemplaires, afin de pouvoir appliquer les *patches* séparément sur les sources originales :

```
[rpi-kernel-vanilla]$ cd ..
[~]$ cp  -R  rpi-kernel-vanilla/   rpi-kernel-preempt
[~]$ cp  -R  rpi-kernel-vanilla/   rpi-kernel-xenomai
```

Compilation et installation du noyau vanilla

Pour le noyau standard *vanilla*, aucune modification n'est à apporter, nous allons simplement sélectionner une configuration pour le Raspberry Pi.

```
[~]$ cd  rpi-kernel-vanilla/
[rpi-kernel-vanilla]$ make  ARCH=arm   bcmrpi_defconfig
 [...]
# configuration written to .config
```

Si vous le souhaitez, vous pouvez examiner la configuration avec :

```
[rpi-kernel-vanilla]$ make   ARCH=arm   menuconfig
```

Nous lançons à présent la compilation. J'ai installé sur mon PC de travail une chaîne de compilation croisée obtenue avec *Buildroot*, mais on peut en utiliser n'importe quelle autre, pourvue qu'elle fonctionne pour le Raspberry Pi.

Sur ma machine la chaîne de cross-compilation est installée ainsi :

```
[rpi-kernel-vanilla]$ ls  /usr/cross/rpi/usr/bin/arm-buildroot-linux-gnueabi-*
/usr/cross/rpi/usr/bin/arm-buildroot-linux-gnueabi-addr2line
/usr/cross/rpi/usr/bin/arm-buildroot-linux-gnueabi-ar
/usr/cross/rpi/usr/bin/arm-buildroot-linux-gnueabi-as
[…]
/usr/cross/rpi/usr/bin/arm-buildroot-linux-gnueabi-gcc
[...]
```

J'ajoute le chemin d'accès à cette chaîne de compilation dans le PATH :

```
[rpi-kernel-vanilla]$ PATH="$PATH:/usr/cross/rpi/usr/bin/"
```

J'indique dans la variable **`CROSS_COMPILE`** le préfixe à ajouter avant les noms standards des outils de compilation (attention à ne pas oublier le tiret final) :

```
[rpi-kernel-vanilla]$ export  CROSS_COMPILE=arm-buildroot-linux-gnueabi-
```

Puis je lance la compilation du noyau :

```
[rpi-kernel-vanilla]$ make   ARCH=arm
```

```
scripts/kconfig/conf --silentoldconfig Kconfig
  WRAP    arch/arm/include/generated/asm/auxvec.h
[...]
  LD      arch/arm/boot/compressed/vmlinux
  OBJCOPY arch/arm/boot/zImage
  Kernel: arch/arm/boot/zImage is ready
```

Le noyau est compilé, ainsi que ses modules. Regroupons ces derniers dans une arborescence facile à copier vers le Raspberry Pi :

```
[rpi-kernel-vanilla]$ mkdir target
[rpi-kernel-vanilla]$ make ARCH=arm INSTALL_MOD_PATH=target/ modules_install
  INSTALL arch/arm/mach-bcm2708/dmaer_master.ko
  INSTALL arch/arm/oprofile/oprofile.ko
[...]
  INSTALL target//lib/firmware/yam/9600.bin
  DEPMOD  3.8.13+
```

Je crée une petite archive regroupant tous les modules afin de les transférer aisément vers la cible :

```
[rpi-kernel-vanilla]$ cd target/lib/modules
[modules]$ tar -cf 3.8.13.tar 3.8.13+/
[modules]$ cd ../../..
```

Pour installer le nouveau noyau et ses modules sur le Raspberry Pi, je les copie par le réseau via SSH. Mon Raspberry Pi se trouve à l'adresse **`192.168.3.123`**, il faudra évidemment adapter la commande à votre cas.

```
[rpi-kernel-vanilla]$ scp arch/arm/boot/zImage root@192.168.3.123:/boot/kernel-vanilla.img
root@192.168.3.123's password:
zImage                    100% 2885KB   2.8MB/s   00:0
[rpi-kernel-vanilla]$ scp target/lib/modules/3.8.13.tar root@192.168.3.123:/lib/modules/
root@192.168.3.123's password:
3.8.13.tar                100%   30MB   5.0MB/s   00:06
```

Je vais décompresser l'archive qui contient les modules du noyau en me connectant en SSH sur le Raspberry Pi.

```
[rpi-kernel-vanilla]$ ssh root@192.168.3.123
root@192.168.3.123's password:
root@raspberrypi:~# cd /lib/modules/
root@raspberrypi:/lib/modules# tar xf 3.8.13.tar
```

Le noyau est copié sur la partition de *boot* du Raspberry Pi mais il faut le renommer pour qu'il soit pris en considération par le *bootloader*. Je conserve par précaution l'ancienne version du noyau.

```
root@raspberrypi:/lib/modules# cd /boot/
root@raspberrypi:/boot# ls
LICENSE.oracle  bootcode.bin  cmdline.txt  config.txt  fixup.dat
fixup_cd.dat  fixup_x.dat  issue.txt  kernel-vanilla.img
kernel.img  start.elf  start_cd.elf  start_x.elf
root@raspberrypi:/boot# cp kernel.img kernel-backup.img
root@raspberrypi:/boot# cp kernel-vanilla.img kernel.img
```

Je peux à présent tester le noyau fraîchement compilé :

```
root@raspberrypi:/boot# reboot
```

Après re-démarrage et connexion sur le Raspberry Pi, je vérifie le numéro de version du noyau :

```
root@raspberrypi:~# uname -a
Linux raspberrypi 3.8.13+ #1 PREEMPT Mon Sep 1 10:44:49 CEST 2014 armv6l GNU/Linux
```

Attention aux confusions : le mot-clé **`PREEMPT`** affiché par **`uname -a`** indique que le noyau a été compilé en mode préemptible basique. Ceci n'est pas équivalent au patch **PREEMPT_RT** qui améliore très nettement cette préemptibilité.

Compilation et installation d'un noyau PREEMPT_RT

Avant de nous livrer aux expériences sur les performances du temps réel, nous pouvons commencer par installer les deux autres environnements, en commençant par l'application du patch PREEMPT_RT. Téléchargeons-le :

```
[~]$ cd rpi-kernel-preempt/
[rpi-kernel-preempt]$ wget https://www.kernel.org/pub/linux/kernel/projects/rt/3.8/older/patch-3.8.13-rt16.patch.xz
[rpi-kernel-preempt]$ xz -d patch-3.8.13-rt16.patch.xz
[rpi-kernel-preempt]$ ls patch*
patch-3.8.13-rt16.patch
```

On peut, bien sûr, choisir la version avec un navigateur à partir de l'adresse **https://www.kernel.org/pub/linux/kernel/projects/rt/**.

On applique le patch pour modifier les sources du noyau :

```
[rpi-kernel-preempt]$ patch -p1 < patch-3.8.13-rt16.patch
patching file Documentation/hwlat_detector.txt
patching file Documentation/kernel-parameters.txt
[…]
patching file drivers/misc/Kconfig
patching file drivers/misc/Makefile
Hunk #1 FAILED at 49.
1 out of 1 hunk FAILED
1 out of 1 hunk FAILED -- saving rejects to file drivers/misc/Makefile.rej
patching file drivers/misc/hwlat_detector.c
[…]
patching file scripts/mkcompile_h
[rpi-kernel-preempt]$
```

Nous voyons une petite erreur dans l'application du *patch* sur un fichier **`Makefile`**. Ceci n'a pas d'importance dans notre cas, mais on pourrait éventuellement intervenir manuellement pour faire la modification notifiée dans le fichier **`.rej`** (l'ajout du fichier **`hwlat_detector`** en l'occurrence).

En pratique, je conseille d'appeler la commande **`patch`** une première fois avec l'option **`--dry-run`** pour faire un passage à vide en vérifiant les éventuelles erreurs puis de la rappeler sans l'option pour faire effectivement les modifications.

Après sélection d'une configuration par défaut pour Raspberry Pi, nous faisons une petite modification :

```
[rpi-kernel-preempt]$ make ARCH=arm bcmrpi_defconfig
[...]
[rpi-kernel-preempt]$ make ARCH=arm menuconfig
```

Dans le menu « *Kernel features* », pour l'option « *Preemption model* » sélectionner la configuration « *Fully Preemptible Kernel (RT)* ».

Le reste de la compilation est identique au noyau *vanilla*. Une fois celle-ci terminée et le « **`make modules_install`** » réalisé, pour le transfert vers la cible les noms de fichiers sont un peu modifiés :

```
[rpi-kernel-preempt]$ cd target/lib/modules/
[modules]$ ls
3.8.13-rt16+
[modules]$ tar -cf 3.8.13-rt16.tar 3.8.13-rt16+/
[modules]$ cd ../../..
[rpi-kernel-preempt]$ scp target/lib/modules/3.8.13-rt16.tar root@192.168.3.123:/lib/modules/
root@192.168.3.123's password:
3.8.13-rt16.tar                                   100%   30MB   4.3MB/s   00:07
[rpi-kernel-preempt]$ scp arch/arm/boot/zImage root@192.168.3.123:/boot/kernel-rt.img
root@192.168.3.123's password:
zImage                                            100% 2873KB   2.8MB/s   00:01
[rpi-kernel-preempt]$ ssh root@192.168.3.123
root@192.168.3.123's password:
root@raspberrypi:~# cd /lib/modules/
root@raspberrypi:/lib/modules# tar xf 3.8.13-rt16.tar
root@raspberrypi:/lib/modules# cd /boot/
root@raspberrypi:/boot# cp kernel-rt.img kernel.img
```

Pour que le système *boote* correctement sur un noyau **`PREEMPT_RT`**, il est nécessaire d'éditer sur le Raspberry Pi le fichier **`/boot/cmdline.txt`** pour y ajouter l'option (par exemple en début de ligne) « **`sdhci-bcm2708.enable_llm=0`** »

Après redémarrage du Raspberry Pi, nous voyons l'option « **`RT`** » dans la commande **`uname -a`**.

```
root@raspberrypi:~# uname -a
Linux raspberrypi 3.8.13-rt16+ #2 PREEMPT RT Mon Sep 1 15:04:57 CEST 2014 armv6l GNU/Linux
```

Compilation et installation d'un système Xenomai

Xenomai repose sur deux choses :

- un noyau Linux standard sur lequel on applique un patch pour intégrer *ipipe*,

- des bibliothèques pour la compilation (et éventuellement l'exécution) des tâches.

Nous devons appliquer trois *patches* sur le noyau (deux spécifiques au Raspberry Pi, et un générique pour l'architecture ARM). Puis nous allons nous aider du script **`prepare-kernel.sh`** fourni avec Xenomai pour préparer la configuration.

```
[~]$ cd rpi-kernel-xenomai/
[rpi-kernel-xenomai]$ patch -p1 < ../xenomai-2.6/ksrc/arch/arm/patches/raspberry/ipipe-core-3.8.13-raspberry-pre-2.patch
patching file kernel/trace/ftrace.c
[rpi-kernel-xenomai]$ patch -p1 < ../xenomai-2.6/ksrc/arch/arm/patches/ipipe-core-3.8.13-arm-3.patch
[...]
[rpi-kernel-xenomai]$ patch -p1 < ../xenomai-2.6/ksrc/arch/arm/patches/raspberry/ipipe-core-3.8.13-raspberry-post-2.patch
[rpi-kernel-xenomai]$ ../xenomai-2.6/scripts/prepare-kernel.sh --linux=. --ipipe=../xenomai-2.6/ksrc/arch/arm/patches/ipipe-core-3.8.13-arm-3.patch --arch=arm
```

```
[rpi-kernel-xenomai]$ make ARCH=arm bcmrpi_defconfig
[rpi-kernel-xenomai]$ make ARCH=arm menuconfig
```

Dans l'interface de configuration du noyau, deux options devront être modifiées :

- Dans le menu « *Kernel Features* », désactiver l'option « *Enable -fstack-protector buffer overflow detection* ».

- Dans le menu « *Kernel hacking* », désactiver l'option « *KGDB: kernel debugger --->* ».

La compilation et l'installation se feront comme précédemment. Après avoir redémarré sur le nouveau noyau, nous voyons deux nouvelles entrées dans **`/proc`** :

```
root@raspberrypi:~# uname -a
Linux raspberrypi 3.8.13-ipipe+ #1 PREEMPT Tue Sep 2 00:58:06 CEST 2014 armv6l GNU/Linux
root@raspberrypi:~# ls /proc/ipipe/
Linux version Xenomai
root@raspberrypi:~# ls /proc/xenomai/
acct faults interfaces latency rtdm schedclasses timebases timerstat
apc heap irq registry sched stat timer version
```

Ceci nous fournit la partie « kernel » de Xenomai, mais il faut également disposer de la partie applicative (bibliothèques pour compiler et exécuter nos programmes, outils de tests, etc.). Pour compiler tout ceci, nous devons retourner dans le répertoire des sources de Xenomai. La variable d'environnement **`PATH`** doit à nouveau être configurée pour contenir le répertoire du *cross-compiler*.

```
[~]$ cd xenomai-2.6
[xenomai-2.6]$ ./configure --host=arm-buildroot-linux-gnueabi CFLAGS='-march=armv6' LDFLAGS='-march=armv6' --enable-shared=no
 […]
[xenomai-2.6]$ make
 […]
[xenomai-2.6.$ make DESTDIR=$(pwd)/target install
 […]
```

On peut remarquer que j'ai ajouté l'option **`--enable-shared=no`** qui construira les outils de tests que nous emploierons avec une édition statique des liens. Ceci nous évite d'avoir à installer les bibliothèques dynamiques sur la cible.

Ainsi, je ne vais transférer sur le Raspberry que les répertoires contenant les exécutables :

```
[xenomai-2.6]$ cd target/
[target]$ tar cf xenomai-user.tar usr/xenomai/bin/ usr/xenomai/sbin/
[target]$ scp xenomai-user.tar root@192.168.3.123:/
[target]$ ssh root@192.168.3.123
root@192.168.3.123's password:
root@raspberrypi:~# cd /
root@raspberrypi:/# tar xf xenomai-user.tar
```

Outils de mesure

Il existe de nombreux outils pour mesurer la qualité d'un système temps réel. Certains s'intéressent aux temps de commutation entre tâches, d'autres à la précision des timers, à la latence des interruptions, aux temps de prise d'un mutex, etc. Dans le cadre de cet article, j'ai choisi d'utiliser un outil bien connu et dont les résultats sont assez représentatifs du comportement temps réel global d'un système : **`cyclictest`**.

Initialement écrit par Thomas Gleixner, ce programme est maintenant intégré dans la suite **`rt-tests`** maintenue par Clark Williams. Il en existe également une version incorporée dans les outils de tests de Xenomai, ce qui nous permettra d'avoir une base uniforme de mesure entre nos systèmes.

Le principe de **`cyclictest`** est de vérifier la précision des déclenchements de tâches périodiques. Il programme une tâche qui doit être réveillée toutes les millisecondes. Lors de son activation elle vérifie l'heure système et compare le temps écoulé depuis le dernier réveil et la période prévue. Après un nombre

conséquent de déclenchements, on a un bon aperçu du comportement temps réel du système pour ce qui concerne un traitement périodique.

Compilons **`cyclictest`** sur le Raspberry Pi :

```
root@raspberrypi:~# git clone git://git.kernel.org/pub/scm/linux/kernel/git/clrkwllms/rt-tests.git
Cloning into 'rt-tests'...
 [...]
root@raspberrypi:~# cd rt-tests/
root@raspberrypi:~/rt-tests# make
 [...]
root@raspberrypi:~/rt-tests# ls -l cyclictest
-rwxr-xr-x 1 root root 50683 Sep  6 02:27 cyclictest
root@raspberrypi:~/rt-tests#
```

Il y a plusieurs options de **`cyclictest`** qui nous intéressent :

`--duration`	Durée du test, avec un suffixe **`m`** (minutes), **`h`** (heures), voire **`d`** (jours).
`--quiet`	Ne rien afficher pendant l'exécution, seulement un compte-rendu à la fin (utile si on redirige le résultat dans un fichier.
`--mlockall`	S'assurer avant le démarrage du test que tout le code exécutable est chargé en mémoire physique et ne la quittera pas (indispensable en temps réel).
`--latency=0`	Écrire **`0`** dans **`/dev/cpu_dma_latency`** pour empêcher le processeur de s'endormir dans des sommeils plus profonds que *C0*.
`--policy=fifo`	Choisir un ordonnancement **`other`** (non temps réel) **`fifo`** ou **`rr`** (*round robin*) pour l'exécution du test. Les ordonnancements **`fifo`** et **`rr`** réclament les droits *root*.
`--priority=99`	Fixer une priorité temps réel de **`99`** (la plus élevée) pour les ordonnancements **`fifo`** ou **`rr`**. Pour l'ordonnancement **`other`**, seule la priorité **`0`** est possible.
`--nanosleep`	Par défaut **`cyclictest`** utilise des timers basés sur les signaux Unix avec **`setitimer()`**. Avec cette option il emploiera plutôt une boucle autour de **`clock_nanosleep()`**.
`--system`	Combinée avec la précédente, cette option réclame d'utiliser l'appel système **`nanosleep()`** plutôt que **`clock_nanosleep()`** qui n'est pas toujours disponible.
`--histogram=10000`	Afficher à la fin de l'exécution un histogramme des latences mesurées jusqu'à 10000 microsecondes. Ceci nous sera très utile pour comparer les comportements.

Bien sûr, pour réaliser une série complète de mesures, j'utilise un script, qui permettent de tester successivement :

- les ordonnancements **`other`**, **`fifo`**, **`rr`**,

- et pour chacun d'eux les fonctionnements avec **`setitimer()`**, **`nanosleep()`** et **`clock_nanosleep()`**,

Le script est exécuté (il dure quinze heures) successivement sur un noyau *vanilla* puis sur un noyau **`PREEMPT_RT`**. Il est disponible sur mon dépôt Github :

```
root@raspberrypi:~# git clone https://github.com/cpb-/Article-RPi-temps-reel.git
 [...]
root@raspberrypi:~# cd Article-Rpi-temps-reel/
root@raspberrypi: ~/Article-RPi-temps-reel# ls
LICENSE  load.sh  README.md  results  run-all-cyclic-tests.sh
```

Outils de charge système

Pour que les mesures soient intéressantes et significatives, il faut qu'elles soient réalisées alors que le système est nettement plus chargé que pour son fonctionnement normal. Lorsque je dois vérifier le

comportement d'un système temps réel pour un projet industriel, je fais fonctionner simultanément :

- l'application « métier » pour laquelle le système est développé (en vérifiant qu'elle tourne correctement dans toutes les circonstances),

- des outils de mesures comme **`cyclictest`** qui vont vérifier les fluctuations du système (ces outils doivent être configurés pour s'exécuter avec une priorité supérieure à celle de l'application),

- des scripts de charge qui mettent le système sous pression.

Les vérifications durent habituellement plusieurs jours, si possible sur des machines différentes, afin d'obtenir au final un compte-rendu représentatif du comportement du système.

> Pour charger efficacement un système, j'ai l'habitude de procéder avec plusieurs scripts spécifiques qui solliciteront intensivement différents sous-systèmes du noyau. Par exemple l'ordonnanceur (des dizaines ou centaines de processus en parallèle), le *Virtual File System* (des parcours incessants des fichiers de l'arborescence), le sous-système *Block* (des lectures / écritures volumineuses et aléatoirement réparties sur des périphériques blocs), la pile de protocole réseau (des transferts de données par TCP/IP et UDP/IP), les gestionnaires d'interruption (avec un **`ping`** en mode *flood* depuis un autre poste ou un générateur basses-fréquences connecté à une interruption *GPIO*).
>
> En outre, j'aime bien que mes différents scripts s'exécutent périodiquement, avec des temps de repos pour le système. En effet, les perturbations les plus importantes surviennent souvent lors de périodes transitoires (montées en charge par exemple) plutôt que pendant les régimes permanents. Ceci est dû par exemple au temps de réveil du processeur mis ponctuellement en sommeil.
>
> Pour éviter les effets liés à un fonctionnement cyclique, chaque script dispose d'une période de fonctionnement différente de celles des autres, et d'un rapport variable entre temps d'activité et temps de repos.

Il existe un script livré avec Xenomai dont la vocation est de forcer une charge assez importante sur un système. Son nom est explicite : **`dohell`** ! On peut très bien l'utiliser sur un noyau *vanilla* ou **`PREEMPT_RT`**, puisqu'il ne fait appel qu'à des commandes système classiques (**`cat`**, **`dd`**, **`nc`**, **`ps`**, etc.).

Mon petit script **`load.sh`** invoque en boucle **`dohell`** pour des durées aléatoires de 15 à 25 secondes suivies d'un temps de repos pour une période totale de 30 secondes. On fournit dans la variable **`SERVER`** l'adresse IP d'une machine du sous-réseau vers laquelle **`dohell`** enverra des trames TCP.

Le script s'exécute en arrière-plan. Pour l'arrêter, il suffit d'effacer le fichier **`load.pid`** qu'il crée dans son répertoire de lancement.

Résultats des mesures

Noyau vanilla

Après avoir démarré le script **`load.sh`**, je lance le script qui invoque **`cyclictest`** pour toutes les situations décrites plus haut. Il s'exécute pendant quinze heures.

```
root@raspberrypi:~# ./run-all-cyclictest.sh
Policy OTHER (1h tests)
  [08:58:25] setitimer
  [09:58:25] clock_nanosleep
  [10:58:26] nanosleep

Policy FIFO (2h tests)
  [11:58:26] setitimer
  [13:58:26] clock_nanosleep
  [15:58:26] nanosleep

Policy RR (2h tests)
  [17:58:26] setitimer
```

```
 [19:58:27] clock_nanosleep
 [21:58:27] nanosleep
root@raspberrypi:~# ls   *3.8.13*
results-3.8.13+-fifo-clocknanosleep.txt
results-3.8.13+-fifo-nanosleep.txt
results-3.8.13+-fifo-setitimer.txt
results-3.8.13+-other-clocknanosleep.txt
results-3.8.13+-other-nanosleep.txt
results-3.8.13+-other-setitimer.txt
results-3.8.13+-rr-clocknanosleep.txt
results-3.8.13+-rr-nanosleep.txt
results-3.8.13+-rr-setitimer.txt
root@raspberrypi:~#
```

Chaque fichier contient 10.000 lignes comme celles-ci permettant de construire l'histogramme :

```
000014 000005
000015 000379
000016 007943
```

Chaque ligne correspond à une classe dont l'amplitude est d'une microseconde et dont l'effectif est indiqué dans la seconde colonne. Cela signifie qu'à 5 reprises le timer était en retard de 14 microsecondes, à 379 reprises il l'était de 15 microsecondes et que 7943 fois il était retardé de 16 microsecondes.

Il y a également des lignes de commentaires et de statistiques commençant par un caractère '**#**' :

```
root@raspberrypi:~# grep   '^#'   results-3.8.13+-rr-setitimer.txt
# /dev/cpu_dma_latency set to 0us
# Histogram
# Total: 007199162
# Min Latencies: 00024
# Avg Latencies: 00053
# Max Latencies: 19085
# Histogram Overflows: 00003
# Histogram Overflow at cycle number:
# Thread 0: 1181998 4732967 4807079
```

Ces lignes nous indiquent :

- La latence minimale est de 24 microsecondes. C'est le temps entre le déclenchement théorique du timer et l'exécution du *handler*.

- La latence moyenne est de 53 microsecondes ce qui est tout à fait raisonnable.

- La latence maximale (c'est la valeur qui nous intéressent dans les conceptions de systèmes temps réel) : 19,085 millisecondes.

- Comme l'histogramme est dimensionné par une option du script de lancement avec un maximum de 10.000 microsecondes, soient 10 millisecondes, il a été dépassé à trois reprises.

- Les numéros des cycles où l'histogramme est dépassé sont indiqués sur la dernière ligne.

La latence maximale est très médiocre, mais nous utilisons un noyau *vanilla*. Espérons que les valeurs obtenues seront meilleures avec **`PREEMPT_RT`** et Xenomai.

Nous pouvons réaliser un histogramme des valeurs collectées. J'ai placé dans le dépôt Github de cet article un script nommé **`draw_histogram.sh`** qui crée avec **`gnuplot`** un fichier **`PNG`** à partir de la sortie de **`cyclictest`**.

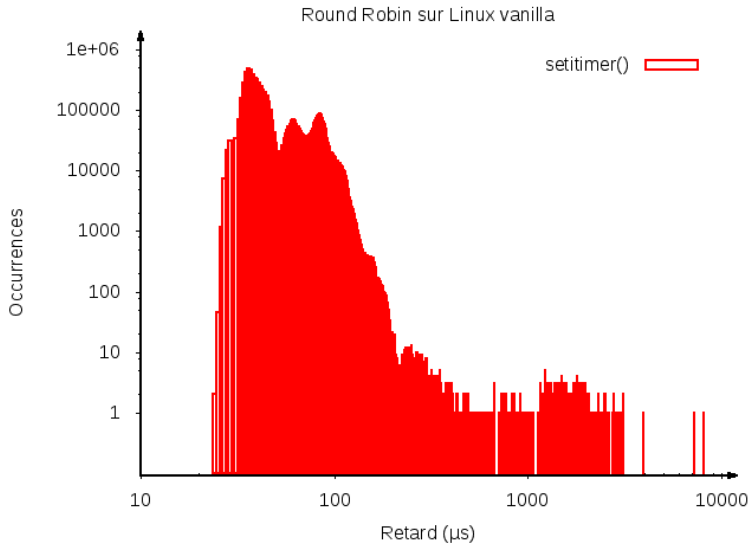

Figure 2 – Histogramme de setitimer() en ordonnancement RR sur un système Linux vanilla

L'histogramme représente en abscisse les latences par rapport au timer programmé, exprimées en microsecondes et en ordonnée le nombre de mesures où la latence considérée a été observée. Ce graphique appelle deux remarques :

- L'échelle des abscisses est logarithmique car les comportements les plus intéressants se situent en dessous de 100 microsecondes, mais il y a quand même des latences de plusieurs millisecondes. Ce mode de représentation permet d'avoir un aperçu lisible de la répartition des latences.

- L'échelle des ordonnées est également logarithmique car certaines latences ont été rencontrées plusieurs centaines de milliers de fois (voire plusieurs millions dans les figures à venir), mais ce qui nous intéresse également ce sont les cas isolés, où une latence a été observée une seule fois (la valeur maximale par exemple).

Nous allons commencer par comparer les trois méthodes possibles pour programmer un traitement périodique : **`setitimer()`**, **`nanosleep()`** et **`clock_nanosleep()`**. Pour cela, un second script, **`compare-results.sh`**, superpose sur le même graphique les histogrammes – représentés sous formes de courbes – issus de plusieurs fichiers.

Voici le comportement de ces trois méthodes pour l'ordonnancement temps réel Fifo :

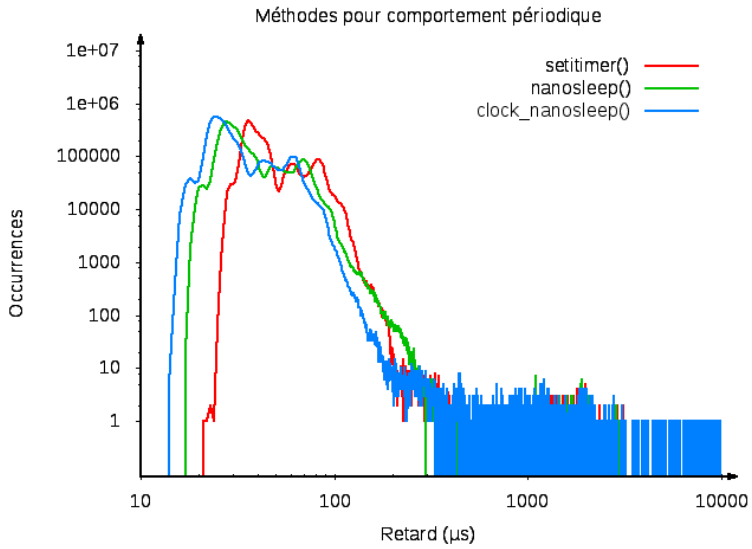

Figure 3 – Comparaison des appels système en ordonnancement Fifo.

Nous voyons que les performances de **`setitimer()`** sont sensiblement moins bonnes que celles de **`nanosleep()`** et **`clock_nanosleep()`**. Cette dernière méthode est la meilleure, et c'est celle que nous utiliserons dans les comparaisons à venir.

Par exemple, nous pouvons examiner les fluctuations d'un système périodique programmé avec **`clock_nanosleep()`** sous les différents ordonnancements d'un noyau *vanilla*.

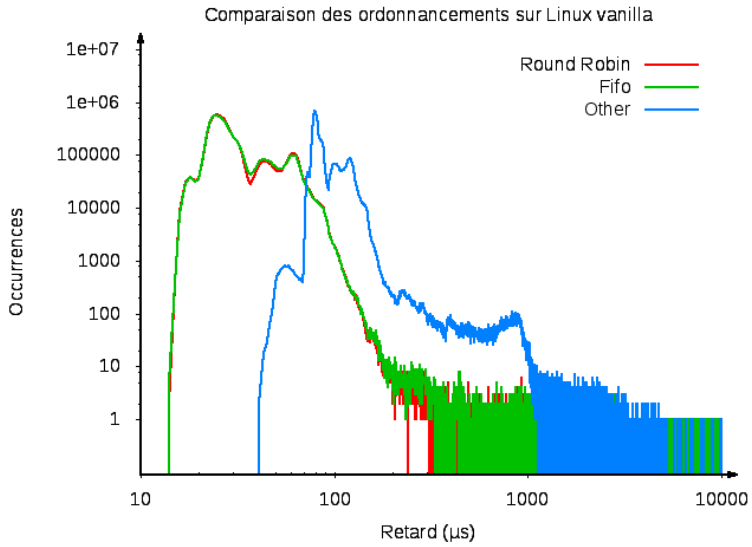

Figure 4 – Comparaison des ordonnancements sur Linux vanilla.

L'ordonnancement **`other`** n'étant pas temps réel, il n'est présent sur cette figure que pour information. Nous voyons que les performances des ordonnancements *Fifo* et *Round Robin* sont globalement équivalentes.

Noyau Linux PREEMPT_RT

Nous allons refaire la même comparaison sur un noyau ayant été modifié par le patch **`PREEMPT_RT`**.

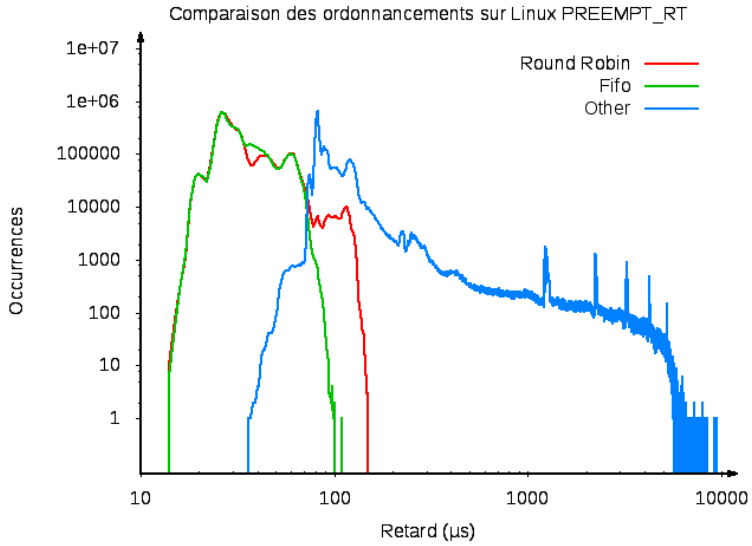

Figure 5 – Comparaison des performances des ordonnancements sur Linux PREEMPT_RT.

Cette fois, il y a une légère différence entre *Round Robin* et *Fifo*, ce dernier étant plus performant. On voit également que les performances globales des ordonnancements temps réel se sont sensiblement améliorées. Alors qu'avec un noyau Linux *vanilla*, des fluctuations pouvaient dépasser la milliseconde, cette fois on dépasse à peine la centaine de microsecondes pour les ordonnancements temps réel.

```
root@raspberrypi:~# grep Max results-3.8.13-rt16+-*-clocknanosleep.txt
results-3.8.13-rt16+-fifo-clocknanosleep.txt:# Max Latencies: 00109
results-3.8.13-rt16+-other-clocknanosleep.txt:# Max Latencies: 15389
results-3.8.13-rt16+-rr-clocknanosleep.txt:# Max Latencies: 00150
```

Xenomai

Il existe une version de **`cyclictest`** pour Xenomai ayant un peu moins d'options que celle de **`rt-test`**, mais elle convient parfaitement pour nos expériences. Elle n'utilise que la méthode **`clock_nanosleep()`** ce qui nous permet une comparaison significative avec les autres systèmes.

J'ai laissé le programme s'exécuter sous une forte charge système pendant douze heures.

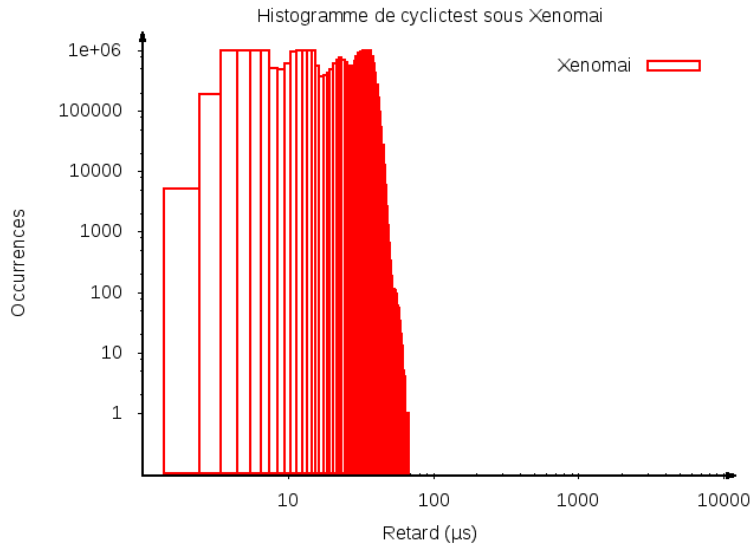

Figure 6 – Histogramme obtenu avec Xenomai.

Cette fois l'échelle des abscisses commence à zéro, car les latences sont beaucoup plus courtes. Nous voyons également que les résultats sont plus tassés, la valeur maximale étant nettement en dessous de 100 microsecondes.

```
[results]$ grep  '#'  results-3.8.13-ipipe+.txt
# Histogram
# Total: 043200000
# Min Latencies: 00002
# Avg Latencies: 00008
# Max Latencies: 00067
# Histogram Overflows: 00000
# Histogram Overflow at cycle number:
# Thread 0:
```

Nous pouvons zoomer sur le graphique pour voir son aspect complet.

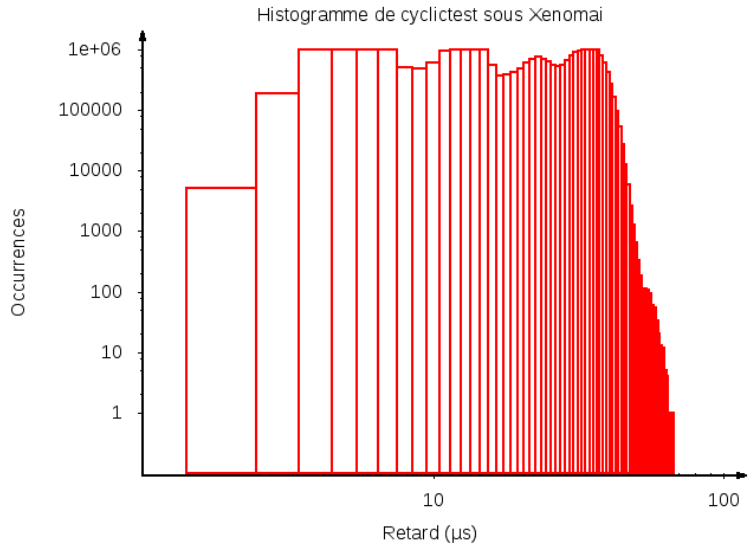

Figure 7 – Histogramme obtenu avec Xenomai (zoom).

Conclusion

Nous voyons que le petit Raspberry Pi a un comportement tout à fait honorable en ce qui concerne les traitements temps réel. Naturellement, il n'est pas question de l'utiliser dans un contexte contrôlant la sécurité des personnes (pas plus que tout autre système sous Linux d'ailleurs) mais il peut très bien servir pour réaliser des tâches impliquant des contraintes temporelles.

Si les tolérances sont de l'ordre de quelques millisecondes, un système Linux *vanilla* peut suffire. Si les contraintes s'expriment plutôt en centaines de microsecondes, le patch **`PREEMPT_RT`** sera adapté. Enfin pour les systèmes dont les tolérances sont en dizaines de microsecondes, on privilégiera Xenomai. (Et si les fluctuations maximale acceptables sont inférieures à la dizaine de microseconde, on n'utilisera pas Linux !)

Bien entendu, il faut être conscient qu'aucune garantie n'est donnée que les valeurs maximales observées ne seront pas dépassées à un moment ou un autre, et que ce risque doit être pris en considération. Les conséquences d'un tel dépassement doivent être soigneusement étudiées avant de choisir un système temps réel. Échouer dans la production d'une pièce sur cent mille par exemple peut être un pari acceptable face à l'utilisation d'un système de contrôle peu coûteux. Si cet échec risque d'entraîner la destruction d'une partie de la chaîne de production, le pari est beaucoup moins raisonnable…

Pour terminer, précisons que les manipulations réalisées ci-dessus (applications des patches, compilations des noyaux, etc.) peuvent paraître complexes, elles sont surtout détaillées ici à titre pédagogique. Pour installer **`PREEMPT_RT`** ou Xenomai sur un système industriel, on fait généralement appel à des environnements de construction comme *Buildroot* ou *Yocto* qui intègrent directement les opérations nécessaires.

Pour en savoir plus

- **xenomai.org** : site de référence de Xenomai ;
- **rt.wiki.kernel.org** : site de référence pour le patch **`PREEMPT_RT`** ;
- **www.linuxembedded.fr/2013/01/17/** : « *PREEMPT_RT sur Raspberry Pi* » - Pierre Ficheux.
- « *Solutions temps réel sous Linux* » Christophe Blaess, éd. Eyrolles, 2012.
- « *Linux embarqué* » Pierre Ficheux, éd. Eyrolles, 2012.

Christophe Blaess (christophe@blaess.fr)
www.blaess.fr/christophe