Temps réel sous LINUX

Pierre Ficheux (pierre.ficheux@openwide.fr)

Patrice Kadionik (kadionik@enseirb.fr)

Mai 2003

Résumé

Cet article a pour but de réaliser un tour d'horizon des solutions temps réel dans l'environnement LINUX. Après une définition du concept de système temps réel, les auteurs s'attacheront à décrire les solutions logicielles disponibles ainsi que quelques exemples de résultats obtenus.

Définition d'un système temps réel

Temps partagé et temps réel

La gestion du temps est l'un des problèmes majeurs des systèmes d'exploitation. La raison est simple : les système d'exploitation modernes sont tous *multitâche,* or ils utilisent du matériel basé sur des processeurs qui ne le sont pas, ce qui oblige le système à partager le temps du processeur entre les différentes tâches. Cette notion de partage implique une gestion du passage d'une tâche à l'autre qui est effectuée par un ensemble d'algorithmes appelé *ordonnanceur* (*scheduler* en anglais).

Un système d'exploitation classique comme UNIX, LINUX ou Windows utilise la notion de *temps partagé,* par opposition *au temps réel.* Dans ce type de système, le but de l'ordonnanceur est de donner à l'utilisateur une impression de confort tout en s'assurant que toutes les tâches demandées sont finalement exécutées. Ce type d'approche entraîne une grande complexité dans la structure même de l'ordonnanceur qui doit tenir compte de notions comme la régulation de la charge du système ou la date depuis laquelle une tâche donnée est en cours d'exécution. De ce fait, on peut noter plusieurs limitations par rapport à la gestion du temps.

Tout d'abord, la notion de priorité entre les tâches est peu prise en compte, car l'ordonnanceur a pour but premier le partage équitable du temps entre les différentes tâches du système (on parle de *quantum* de temps ou *tick*). Notez que sur les différentes versions d'UNIX dont LINUX, la commande `nice` permet de modifier la priorité de la tâche au lancement.

Ensuite, les différentes tâches doivent accéder à des ressources dites *partagées,* ce qui entraîne des incertitudes temporelles. Si une des tâches effectue une écriture sur le disque dur, celui-ce n'est plus disponible aux autres tâches à un instant donné et le délai de disponibilité du périphérique n'est donc pas prévisible.

En outre, la gestion des entrées/sorties peut générer des temps morts car une tâche peut être bloquée en attente d'accès à un élément d'entrée/sortie. La gestion des *interruptions* reçues par une tâche n'est pas optimisée. Le temps de latence - soit le temps écoulé entre la réception de l'interruption et son traitement - n'est pas garanti par le système.

Enfin, l'utilisation du mécanisme de *mémoire virtuelle* peut entraîner des fluctuations importante s dans les temps d'exécution des tâches.

Notion de temps réel

Le cas des systèmes temps réel est différent. Il existe un grande nombre de définition d'un système dit *temps réel* mais une définition simple d'un tel système pourra être la suivante :

Un système temps réel est une association logiciel/matériel où le logiciel permet, entre autre, une gestion adéquate des ressources matérielles en vue de remplir certaines tâches ou

fonctions dans des limites temporelles bien précises.

Un autre définition pourrait être :

"Un système est dit temps réel lorsque l'information après acquisition et traitement reste encore pertinente".

Ce qui signifie que dans le cas d'une information arrivant de façon régulière (sous forme d'une interruption périodique du système), les temps d'acquisition et de traitement doivent rester inférieurs à la période de rafraîchissement de cette information.

Il est évident que la structure de ce système dépendra de ces fameuses contraintes. On pourra diviser les systèmes en deux catégories :

1. Les systèmes dits à contraintes *souples* ou *molles* (*soft real time*). Ces systèmes acceptent des variations dans le traitement des données de l'ordre de la demi-seconde (ou 500 ms) ou la seconde. On peut citer l'exemple des systèmes multimédia : si quelques images ne sont pas affichées, cela ne met pas en péril le fonctionnement correct de l'ensemble du système. Ces systèmes se rapprochent fortement des systèmes d'exploitation classiques à temps partagé. Ils garantissent un temps moyen d'exécution pour chaque tâche. On a ici une répartition **égalitaire** du temps CPU entre processus.
2. Les systèmes dits à contraintes *dures* (*hard real time*) pour lesquels une gestion stricte du temps est nécessaire pour conserver l'intégrité du service rendu. On citera comme exemples les contrôles de processus industriels sensibles comme la régulation des centrales nucléaires ou les systèmes embarqués utilisés dans l'aéronautique. Ces systèmes garantissent un temps maximum d'exécution pour chaque tâche. On a ici une répartition **totalitaire** du temps CPU entre tâches.

Les systèmes à contraintes dures doivent répondre à trois critères fondamentaux :

1. Le déterminisme *logique* : les mêmes entrées appliquées au système doivent produire les mêmes effets.
2. Le déterminisme *temporel* : un tâche donnée doit obligatoirement être exécutée dans les délais impartis, on parle d'*échéance*.
3. La *fiabilité* : le système doit être disponible. Cette contrainte est très forte dans le cas d'un système embarqué car les interventions d'un opérateur sont très difficiles voire même impossibles. Cette contrainte est indépendante de la notion de temps réel mais la fiabilité du système sera d'autant plus mise à l'épreuve dans le cas de contraintes dures.

Un système temps réel n'est pas forcément *plus rapide* qu'un système à temps partagé. Il devra par contre satisfaire à des contraintes temporelles strictes, prévues à l'avance et imposées par le processus extérieur à contrôler. Une confusion classique est de mélanger temps réel et rapidité de calcul du système donc puissance du processeur (microprocesseur, micro-contrôleur, DSP). On entend souvent :

« *Être temps réel, c'est avoir beaucoup de puissance : des MIPS voire des MFLOPS*"

Ce n'est pas toujours vrai. En fait, être temps réel dans l'exemple donné précédemment, c'est être capable d'acquitter l'interruption périodique (moyennant un temps de latence d'acquittement d'interruption imposé par le matériel), traiter l'information et le signaler au niveau utilisateur (réveil d'une tâche ou libération d'un sémaphore) dans un temps inférieur au temps entre deux interruptions périodiques consécutives. On est donc lié à la contrainte de délai entre deux interruptions générées par le processus extérieur à contrôler.
Si cette durée est de l'ordre de la seconde (pour le contrôle d'une réaction chimique par exemple), il ne sert à rien d'avoir un système à base de Pentium IV ! Un simple processeur 8 bits du type micro-contrôleur Motorola 68HC11, Microchip PIC, Scenix, AVR... ou même un processeur 4 bits fera amplement l'affaire, ce qui permettra de minimiser les coûts sur des forts volumes de production.

Si ce temps est maintenant de quelques dizaines de microsecondes (pour le traitement des données issues de l'observation d'une réaction nucléaire par exemple), il convient de choisir un processeur nettement plus performant comme un processeur 32 bits Intel x86, StrongARM ou Motorola ColdFire.

L'exemple donné est malheureusement idyllique (quoique fréquent dans le domaine des télécommunications et réseaux) puisque notre monde interagit plutôt avec un système électronique de façon apériodique.

Il convient donc avant de concevoir ledit système de connaître la durée minimale entre deux interruptions, ce qui est assez difficile à estimer voire même impossible. C'est pour cela que l'on a tendance à concevoir dans ce cas des systèmes performants (en terme de puissance de calcul CPU et de rapidité de traitement d'une interruption) et souvent sur-dimensionnés pour respecter des contraintes temps réel mal cernées à priori. Ceci induit en cas de sur-dimensionnement un sur-coût non négligeable.

En résumé, on peut dire qu'un système temps réel doit être prévisible (*predictible* en anglais*),* les contraintes temporelles pouvant s'échelonner entre quelques micro-secondes (µs) et quelques secondes.

La figure suivante permet d'illustrer la notion de temps réel sur le cas particulier de l'exécution d'une tâche périodique. Idéalement, une tâche périodique doit être exécutée toutes les m secondes (son temps d'exécution reste inférieur à m). Dans le cas d'un système non temps réel, un temps de latence apparaît avant l'exécution effective de la tâche périodique. il varie fortement au cours du temps (charge du système...). Dans le cas d'un système temps réel, ce temps de latence doit être borné et garanti inférieur à une valeur fixe et connue à l'avance. Si ce n'est pas le cas, il y a un défaut de fonctionnement pouvant causer le *crash* du système. Les éditeurs de systèmes temps réel donnent généralement cette valeur qui est fixe (si le système est bien conçu) quelle que soit la charge du système ou du nombre de tâches en fonction du processeur utilisé.

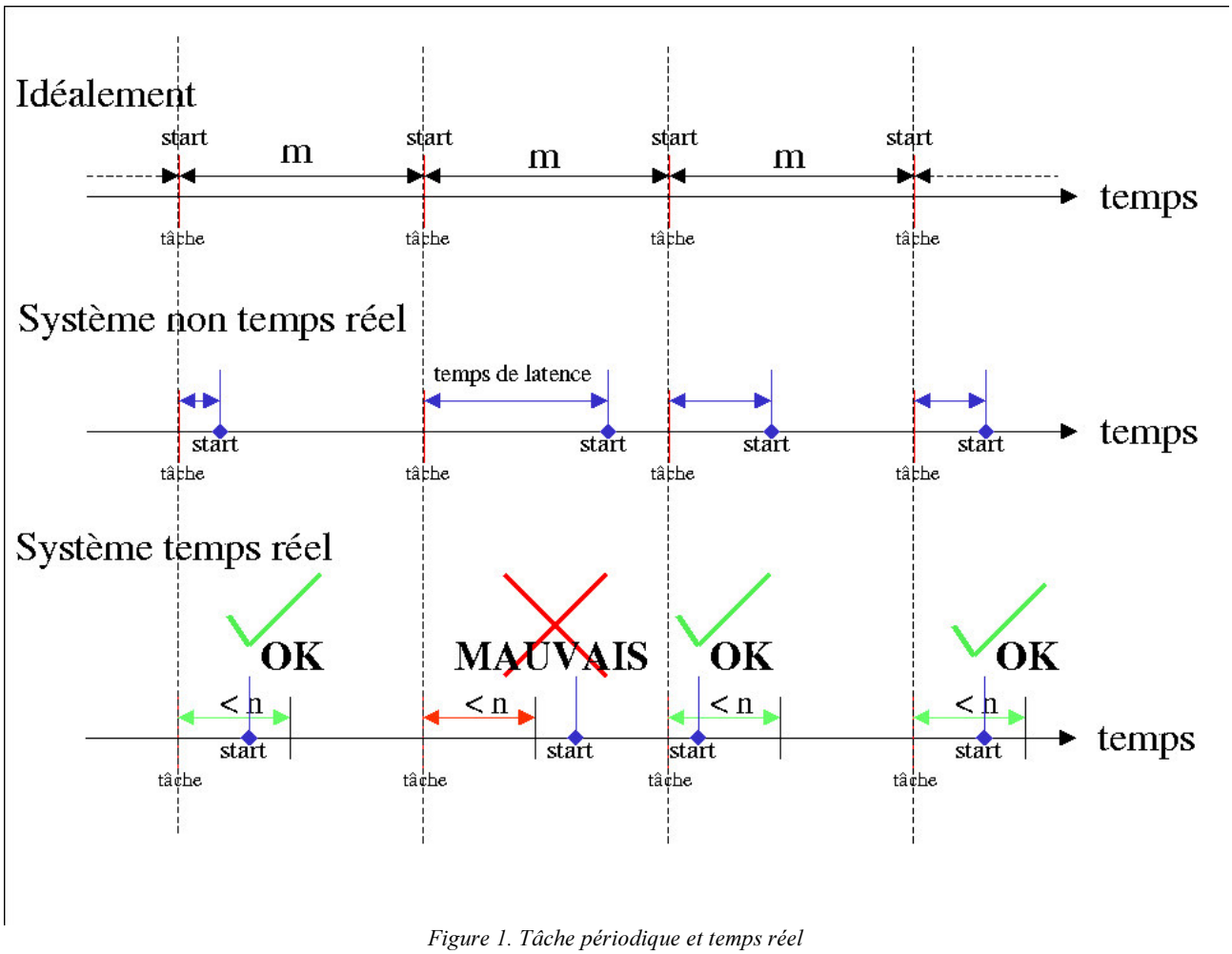

Figure 1. Tâche périodique et temps réel

Une petite expérience

L'expérience décrite sur la figure ci-dessous met en évidence la différence entre un système classique et un système temps réel. Elle est extraite d'un mémoire sur le temps réel réalisé par William Blachier à l'ENSIMAG en 2000.

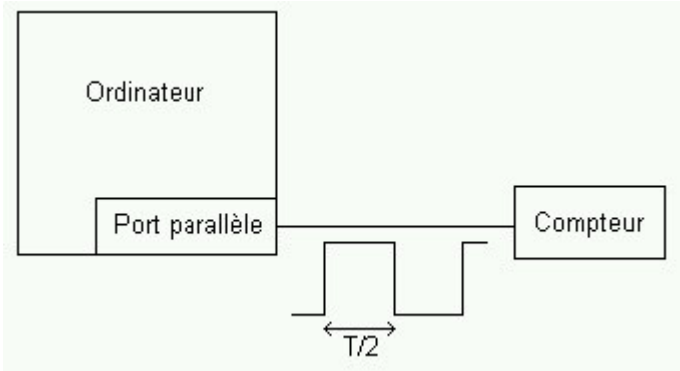

Figure 2. Test comparatif temps réel/temps partagé

Le but de l'expérience est de générer un signal périodique sortant du port parallèle du PC. Le temps qui sépare deux émissions du signal sera mesuré à l'aide d'un compteur. Le but est de visualiser l'évolution de ce délai en fonction de la charge du système. La fréquence initiale du signal

est de 25 Hz (Hertz) ce qui donne une demi-période T/2 de 20 ms.
Sur un système classique, cette demi-période varie de 17 à 23 ms, ce qui donne une variation de fréquence entre 22 Hz et 29 Hz. La figure ci-dessous donne la représentation graphique de la mesure sur un système non chargé puis à pleine charge :

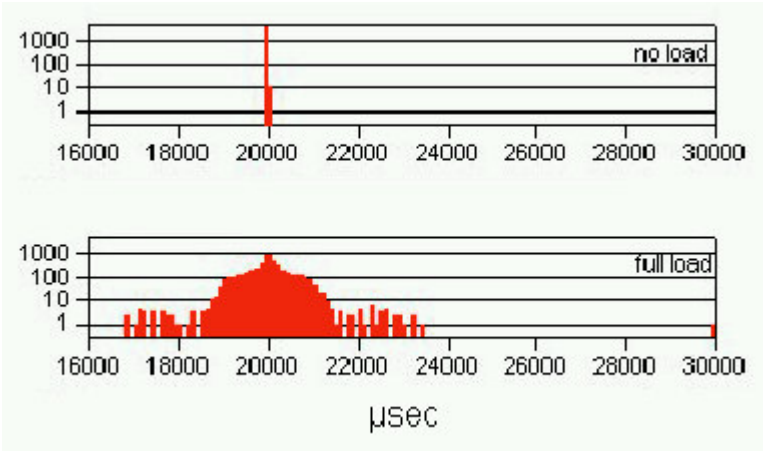

Figure 3. Représentation sur un système classique

Sur un système temps réel, la demi-période varie entre 19,990 ms et 20,015 ms, ce qui donne une variation de fréquence de 24,98 Hz à 25,01 Hz. La variation est donc beaucoup plus faible. La figure donne la représentation graphique de la mesure :

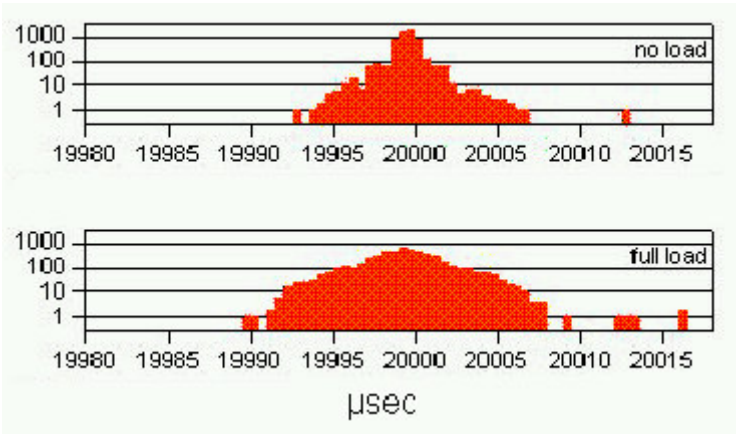

Figure 4. Représentation sur un système temps réel

Lorsque la charge est maximale, le système temps réel assure donc une variation de +/- 0,2 Hz alors que la variation est de +/- 4 Hz dans le cas d'un système classique.

Ce phénomène peut être reproduit à l'aide du programme `square.c` écrit en langage C.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <asm/io.h>

#define LPT 0x378

int ioperm();
```

```
int main(int argc, char **argv)
{
    setuid(0);

    if (ioperm(LPT, 1, 1) < 0) {
          perror("ioperm()");
          exit(-1);
    }

    while(1) {
          outb(0x01, LPT);
          usleep(50000);

          outb(0x00, LPT);
          usleep(50000);
    }
    return(0);
}
```

Le signal généré sur la broche 2 (bit D0) du port parallèle est théoriquement un signal périodique carré de demi-période T/2 de 50 ms. On observe à l'oscilloscope le signal suivant sur un système non chargé (AMD Athlon 1500+).

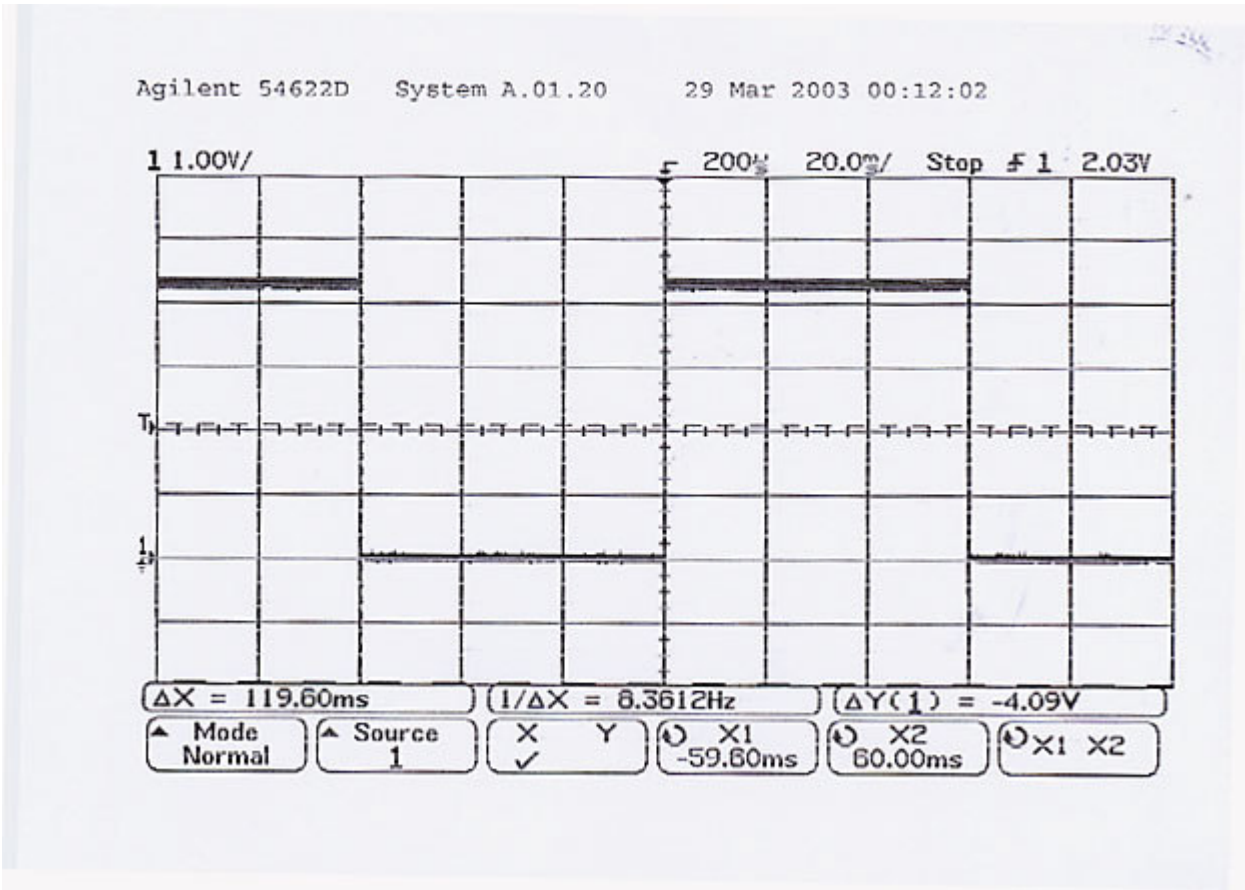

Figure 5. Génération d'un signal carré sous Linux non chargé

On remarque que l'on n'a pas une période de 100 ms mais de 119,6 ms dû au temps supplémentaire d'exécution des appels système. Dès que l'on stresse le système (écriture répétitive sur disque d'un fichier de 50 Mo), on observe le signal suivant :

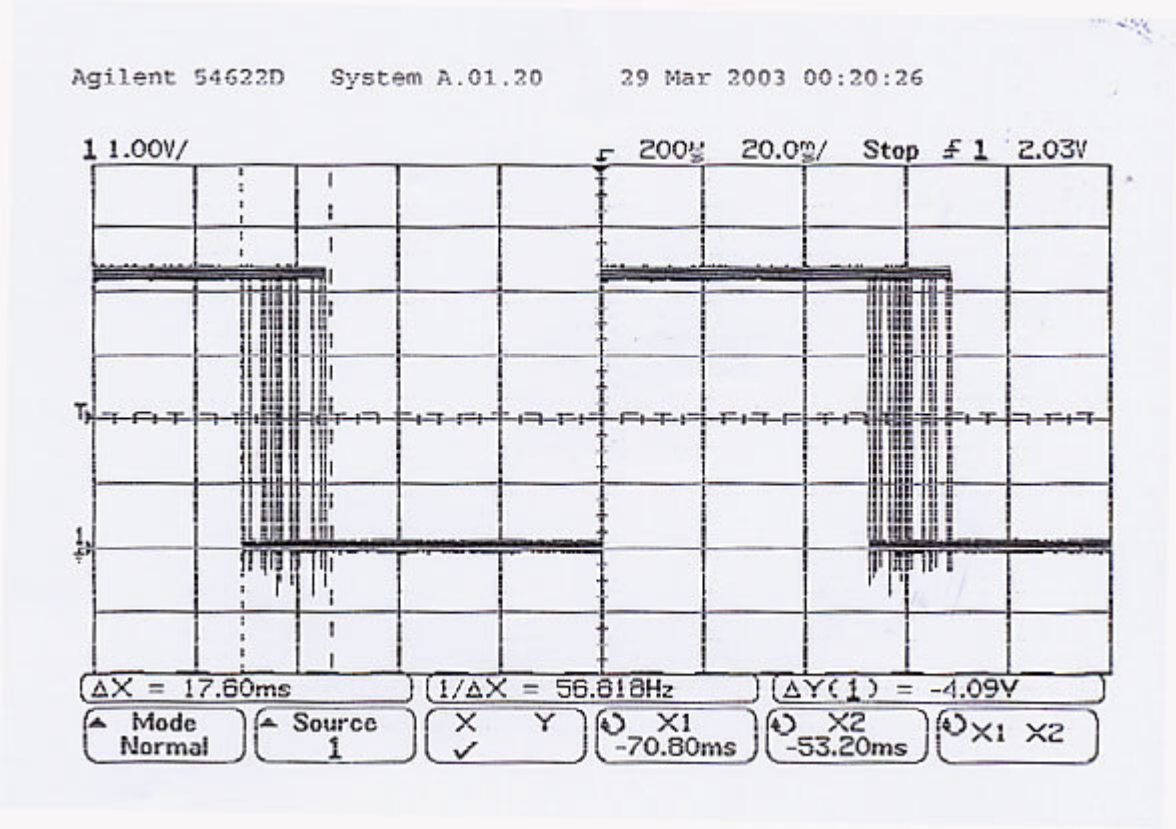

Figure 6. Génération d'un signal carré sous Linux chargé

On observe maintenant une gigue (*jitter*) sur le signal généré. La gigue maximale sur la durée de l'expérience est de 17,6 ms. La forme du signal varie maintenant au cours du temps, n'est pas de forme carrée mais rectangulaire. LINUX n'est donc plus capable de générer correctement ce signal. Il faut noter aussi que le front montant sur la figure précédente apparaît sans gigue car il a servi comme front de synchronisation de l'oscilloscope. La gigue observée est donc à voir comme la contribution de la gigue sur front montant et sur front descendant. Si l'on diminue la valeur de la demi-période, la gigue devient aussi importante que cette dernière et dans ce cas, Linux ne génère plus aucun signal !

Cette expérience met donc en évidence l'importance des systèmes temps réel pour certaines applications critiques.

Préemption et commutation de contexte

Le noyau (*kernel*) est le composant principal d'un système d'exploitation multitâche moderne. Dans un tel système, chaque tâche (ou processus) est décomposée en *threads* (*processus léger* ou *tâche légère*) qui sont des éléments de programmes coopératifs capables d'exécuter chacun une portion de code dans un même espace d'adressage. Chaque thread est caractérisé par un *contexte* local contenant la *priorité* du thread, ses variables locales et l'état de ses registres. Le passage d'un thread à un autre est appelé changement de contexte (*context switch).* Ce changement de contexte sera plus rapide sur un thread que sur un processus car les threads d'un processus évoluent dans le même espace d'adressage ce qui permet le partage des données entre les threads d'un même processus. Dans certains cas, un processus ne sera composé que d'un seul thread et le

changement de contexte s'effectuera sur le processus lui-même.

Dans le cas d'un système temps réel, le noyau est dit *préemptif,* c'est à dire qu'un thread peut être interrompu par l'ordonnanceur en fonction du niveau de priorité et ce afin de permettre l'exécution d'un thread de plus haut niveau de priorité prêt à être exécuté. Ceci permet d'affecter les plus hauts niveaux de priorité à des tâches dites *critiques* par rapport à l'environnement réel contrôlé par le système. La vérification des contextes à commuter est réalisée de manière régulière par l'ordonnanceur en fonction de l'horloge logicielle interne du système, ou *tick timer* système.

Dans le cas d'un noyau non préemptif - comme le noyau LINUX - un thread sera interrompu uniquement dans le cas d'un appel au noyau ou d'une une interruption externe. La notion de priorité étant peu utilisée, c'est le noyau qui décide ou non de commuter le thread actif en fonction d'un algorithme complexe.

Les extensions POSIX

La complexité des systèmes et l'interopérabilité omniprésente nécessitent une standardisation de plus en plus grande tant au niveau des protocoles utilisés que du code source des applications. Même si elle n'est pas obligatoire, l'utilisation de systèmes conformes à *POSIX* est de plus en plus fréquente.

POSIX est l'acronyme de *Portable Operating System Interface* ou interface portable pour les systèmes d'exploitation. Cette norme a été développée par l'IEEE (*Institute of Electrical and Electronic Engineering*) et standardisée par l'ANSI (*American National Standards Institute*) et l'ISO (*International Standards Organisation*).

Le but de POSIX est d'obtenir la portabilité des logiciels au niveau de leur code source. Le terme de *portabilité* est un anglicisme dérivé de *portability,* terme lui-même réservé au jargon informatique. Un programme qui est destiné à un système d'exploitation qui respecte POSIX doit pouvoir être adapté à moindre frais sous n'importe quel autre système POSIX. En théorie, le portage d'une application d'un système POSIX vers un autre doit se résumer à une compilation des sources du programme.

POSIX a initialement été mis en place pour les systèmes de type UNIX mais d'autres systèmes d'exploitation comme *Windows NT* sont aujourd'hui conformes à POSIX. Le standard POSIX est divisé en plusieurs sous-standards dont les principaux sont les suivants :

- IEEE 1003.1-1990 : POSIX Partie 1 : Interface de programmation (API) système. Définition d'interfaces de programmation standards pour les systèmes de type UNIX, connu également sous l'appellation ISO 9945-1. Ce standard contient la définition de ces fonctions (*bindings*) en langage C.
- IEEE 1003.2-1992 : Interface applicative pour le s*hell* et applications annexes. Définit les fonctionnalités du shell et commandes annexes pour les systèmes de type UNIX.
- IEEE 1003.1b-1993 : Interface de programmation (API) temps réel. Ajout du support de programmation temps réel au standard précédent. On parle également de POSIX.4.
- IEEE 1003.1c-1995 : Interface de programmation (API) pour le multithreading.

Pour le sujet qui nous intéresse, la plupart des systèmes d'exploitation temps réel sont conformes partiellement ou totalement au standard POSIX. C'est le cas en particulier des systèmes temps réel LynxOS (http ://www.lynuxworks.com) et QNX (http ://www.qnx.com). Quant à LINUX, sa conformité par rapport à POSIX 1003.1b (temps réel) est partielle dans sa version standard et il nécessite l'application de modification (ou *patch)* sur les sources du noyau.

Tour d'horizon des principaux systèmes temps réel

Le but de cette section est d'effectuer un rapide tour d'horizon des principaux systèmes d'exploitation utilisés dans les environnement embarqués. Ce tour d'horizon n'inclut pas les systèmes à base de LINUX qui seront bien entendu décrits en détails plus loin dans l'article. Il convient avant tout de préciser les différences entre noyau, exécutif et système d'exploitation temps réel :

- Un noyau temps réel est le minimum logiciel pour pouvoir faire du temps réel : ordonnanceur, gestion de tâches, communications inter-tâches, autant dire un système plutôt limité mais performant.
- Un exécutif temps réel possède un noyau temps réel complété de modules/bibliothèques pour faciliter la conception de l'application temps réel : gestion mémoire, gestion des E/S, gestion de timers, gestion d'accès réseau, gestion de fichiers. Lors de la génération de l'exécutif, on choisit à la carte les bibliothèques en fonction des besoins de l'application temps réel. Pour le développement, on a besoin d'une machine hôte (*host*) et de son environnement de développement croisé (compilateur C croisé, utilitaires, debugger) ainsi que du système cible (*target*) dans lequel on va télécharger (par liaison série ou par le réseau) l'application temps réel avec l'exécutif.
- Un système d'exploitation temps réel est le cas particulier où l'on a confusion entre le système hôte et le système cible qui ne font plus qu'un. On a donc ici un environnement de développement natif.

VxWorks et pSOS

VxWorks est aujourd'hui l'exécutif temps réel le plus utilisé dans l'industrie. Il est développé par la société *Wind River* (http ://www.windriver.com) qui a également racheté récemment les droits du noyau temps réel *pSOS*, un peu ancien mais également largement utilisé. VxWorks est fiable, à faible empreinte mémoire, totalement configurable et porté sur un nombre important de processeurs (PowerPC, 68K, CPU32, ColdFire, MCORE, 80x86, Pentium, i960, ARM, StrongARM, MIPS, SH, SPARC, NECV8xx, M32 R/D, RAD6000, ST 20, TriCore). Un point fort de VxWorks a été le support réseau (sockets, commandes r..., NFS, RPC...) disponible dès 1989 au développeur bien avant tous ses concurrents. VxWorks est également conforme à POSIX 1003.1b.

QNX

Développé par la société canadienne QNX Software (http ://www.qnx.com), QNX est un système temps réel de type UNIX. Il est conforme à POSIX, permet de développer directement sur la plateforme cible et intègre l'environnement graphique *Photon*, proche de *X Window System*.

µC/OS (micro-C OS) et µC/OS II

µC/OS, développé par le Canadien Jean J. Labrosse, est un exécutif temps réel destiné à des environnements de très petite taille construits autour de micro-contrôleurs. Il est maintenant disponible sur un grand nombre de processeurs et peut intégrer des protocoles standards comme TCP/IP (µC/IP) pour assurer une connectivité IP sur une liaison série par PPP. Il est utilisable gratuitement pour l'enseignement (voir http ://www.ucos-ii.com).

Windows CE

Annoncé avec fracas par Microsoft comme le système d'exploitation embarqué *qui tue,* Windows CE et ses cousins comme Embedded Windows NT n'ont pour l'instant pas détrôné les systèmes embarqués traditionnels. Victime d'une réputation de fiabilité approximative, Windows

CE est pour l'instant cantonné à l'équipement de nombreux assistants personnels ou *PDA*.

LynxOS

LynxOS est développé par la société LynuxWorks *(*http ://www.lynuxworks.com*)* qui a récemment modifié son nom de part son virage vers LINUX avec le développement de Blue Cat. Ce système temps réel est conforme à la norme POSIX.

Nucleus

Nucleus est développé par la société Accelerated Technology Inc. (http ://www.acceleratedtechnology.com). Il est livré avec les sources et il n'y pas de *royalties* à payer pour la redistribution.

eCOS

Acronyme pour *Embeddable Configurable Operating System*, eCOS fut initialement développé par la société Cygnus, figure emblématique et précurseur de l'open source professionnel, aujourd'hui rattachée à la société Red Hat Software. Ce système est adapté aux solutions à très faible empreinte mémoire et profondément enfouies. Son environnement de développement est basé sur LINUX et la chaîne de compilation GNU avec conformité au standard POSIX.

Les contraintes des systèmes propriétaires

La majorité des systèmes propriétaires décrits à la section précédente souffrent cependant de quelques défauts forts contraignants.

Les systèmes sont souvent réalisés par des sociétés de taille moyenne qui ont du mal à suivre l'évolution technologique : le matériel évolue très vite - la durée de vie d'un processeur de la famille x86 est par exemple de l'ordre de 12 à 24 mois - les standards logiciels font de même et de plus en plus d'équipements nécessitent l'intégration de composants que l'on doit importer du monde des systèmes informatiques classiques ou du multimédia.

De ce fait, les coûts de licence et les droits de redistribution des systèmes (ou *royalties*) sont parfois très élevés car l'éditeur travaille sur un segment de marché très spécialisé, une *niche* dans laquelle les produits commercialisés le sont pour leur fonction finale et non pour la valeur du logiciel lui-même. Contrairement au monde de la bureautique où la pression commerciale peut inciter l'utilisateur à faire évoluer son logiciel fréquemment - et donc à payer un complément de licence - le logiciel embarqué est considéré comme un mal nécessaire souvent destiné à durer plusieurs années, en conservant une compatibilité avec du matériel et des processeurs obsolètes.

Le coût de développement d'applications autour de systèmes propriétaires est souvent plus élevé car les outils de développement sont mal connus de la majorité des développeurs disponibles sur le marché du travail car non étudiés à l'université. Il est donc nécessaire de recruter du personnel très spécialisé donc rare. Les formations autour de ces outils sont également onéreuses car très spécialisées ce qui oblige l'éditeur à pratiquer des coûts élevés pour compenser le manque d'effet de masse.

Tout cela implique un ensemble de spécificités contraignantes pour la gestion globale des outils informatiques de l'entreprise.

Les avantages de l'open source

Les trois points suivants de la définition du logiciel open source sont fondamentaux dans le cas du logiciel embarqué et temps réel :

1. La redistribution sans royalties. Ce point règle le problème économique des droits de redistribution ou *royalties*, très contraignant dans le cas d'un système distribué à grande échelle.
2. La disponibilité du code source. Ce point est encore plus fondamental car il est à la base de la conception d'un logiciel de qualité et surtout maintenable dans le temps.
3. La possibilité de réaliser un développement dérivé de ce code source. Là encore, l'open source permet de réaliser des économies substancielles en intégrant des composants existants dans une solution industrielle.

LINUX comme système temps réel

Forts des arguments concernant l'open source, il est normal d'être tenté par l'utilisation de LINUX comme système temps réel. Outre les avantages inhérents à l'open source, la fiabilité légendaire de LINUX en fait un candidat idéal. Malheureusement, LINUX n'est pas nativement un système temps réel. Le noyau LINUX fut en effet conçu dans le but d'en faire un système généraliste donc basé sur la notion de temps partagé et non de temps réel.

La communauté LINUX étant très active, plusieurs solutions techniques sont cependant disponibles dans le but d'améliorer le comportement du noyau afin qu'il soit compatible avec les contraintes d'un système temps réel comme décrit au début de l'article. Concrètement, les solutions techniques disponibles sont divisées en deux familles :

1. Les patchs dits « préemptifs » permettant d'améliorer le comportement du noyau LINUX en réduisant les temps de latence de ce dernier. Ces modifications ne transforment pas LINUX en noyau temps réel « dur » mais permettent d'obtenir des résultats satisfaisants dans le cas de contraintes temps réel « molles ». Cette technologie est disponible auprès de différents projets open source et elle est également supportée commercialement par divers éditeurs spécialisés dont le plus connu est l'américain MontaVista (http://www.mvista.com). La notion de noyau préemptif est intégrée dans le noyau de développement 2.5.
2. Le noyau temps réel auxiliaire. Les promoteurs de cette technologie considèrent que le noyau LINUX ne sera jamais véritablement temps réel et ajoute donc à ce noyau un véritable ordonnanceur temps réel à priorités fixes. Ce noyau auxiliaire traite directement les tâches temps réel et délègue les autres tâches au noyau LINUX, considéré comme la tâche de fond de plus faible priorité. Cette technique permet de mettre en place des systèmes temps réel « durs ». Cette technologie utilisée par RTLinux et son cousin européen RTAI. RTLinux est supporté commercialement par FSMLabs (http://www.fsmlabs.com) qui a des relations chaotiques avec la communauté open source à cause du brevet logiciel qui couvre l'utilisation de ce noyau auxiliaire. Pour cette raison, le projet RTAI - qui n'est pas associé à une entité commerciale - tend à utiliser une technique similaire mais ne tombant pas sous le coup du brevet.

Les patchs préemptifs

Ce paragraphe présente les deux principaux patchs permettant d'améliorer les performances du noyau LINUX 2.4 au niveau du temps de latence pour la prise en compte d'une interruption. Le document décrit la méthode d'application des patchs puis commente les résultats obtenus avec des outils de mesures spécialisés.

Les différentes solutions disponibles

Cette étude se limitera aux patchs les plus utilisés dans l'environnement LINUX :

- Le patch Prempt Kernel maintenu par Robert M. Love et disponible sur http://www.tech9.net/rml/linux. Le principe du patch est d'ajouter systématiquement des occasions d'appel à l'ordonnanceur et donc de minimiser le temps entre la réception d'un évènement et l'appel à la fonction `schedule()` (ordonnanceur LINUX).
- Le patch Low latency maintenu par Andrew Morton et disponible sur http://www.zip.com.au/~akpm/linux/schedlat.html. Le principe est un peu différent car au lieu d'opter pour une stratégie "systématique", les développeurs du patch ont préféré effectuer une analyse du noyau afin d'ajouter des "points de préemption" subtilement placés dans les sources du noyau afin de "casser" des boucles d'attente trop longues.

Il faut noter que l'étude s'applique uniquement au noyau 2.4.20 (dernière version stable disponible à ce jour). Des versions similaires du patch Prempt Kernel sont intégrées au noyau de développement 2.5. Une fusion des deux patchs est peut être envisageable dans le futur noyau 2.6 (information sans garantie mais évoquée dans certains interviews de Robert M. Love). De même, l'éditeur MontaVista Software diffuse une version de noyau 2.4 incluant un patch préemptif très similaire à celui de Robert M. Love (qui travaille d'ailleurs chez MontaVista !).

Procédure d'application du patch

La procédure est similaire pour les deux patchs :

1. Extraire les sources du noyau LINUX dans le répertoire `/usr/src`. Dans notre cas cela donne un répertoire `/usr/src/linux-2.4.20` que l'on peut suffixer par le type de patch. On peut ensuite positionner le lien linux-2.4 sur cette arborescence.

   ```
   # cd /usr/src
   # mv linux-2.4.20 linux-2.4.20_kpreempt
   # rm -f linux-2.4; ln -s linux-2.4.20_kpreempt linux-2.4
   ```

2. Récupérer le patch correspondant à la version du noyau, puis appliquer le patch :

   ```
   # cd /usr/src/linux-2.4
   # patch -p1 < un_repertoire_de_stockage/patchfile
   ```

3. Configurer le noyau LINUX en activant les options habituelles du noyau non patché ainsi que les options spécifiques du patch. Avant d'appliquer le patch il est **IMPERATIF** de vérifier le bon fonctionnement du noyau non patché sur votre matériel.

   Il faut ensuite configurer puis compiler le noyau avec la procédure habituelle :

```
# make xconfig
# make dep; make clean; make bzImage; make modules
```

Configuration du noyau patché Preempt Kernel

Il est conseillé d'utiliser le paramètre EXTRAVERSION défini au début du Makefile du noyau, soit dans ce cas :

```
VERSION = 2
PATCHLEVEL = 4
SUBLEVEL = 20
EXTRAVERSION = -kpreempt
```

ce qui permet de suffixer l'arborescence des modules suivant le type de noyau (ici kpreempt) et donc de faire cohabiter plusieurs versions du noyau 2.4.20 patché sur le même système.

Ensuite il convient de vérifier l'option de noyau préemptif (CONFIG_PREEMPT) en utilisant `make xconfig` et en validant l'option dans la rubrique *Processor type and features* comme indiqué sur la figure ci-dessous.

Figure 7. Validation de CONFIG_PREEMPT

Il est également nécessaire de valider le support RTC (Real Time Clock) car certains outils de mesure de latence utilisent le device `/dev/rtc`.

Figure 8. Validation du support RTC

Enfin il est nécessaire de valider l'utilisation du DMA (Direct Memory Access) sur le bus PCI coté noyau ce qui normalement fait par défaut. L'utilisation du DMA permet d'améliorer fortement les performances puisque le processeur n'est plus sollicité.

Figure 9. Validation du support DMA

Une fois le noyau compilé, on peut l'installer par :

```
# cp arch/i386/boot/bzImage /boot/bzImage-2.4.20_kpreempt
# make modules_install
```

puis ajouter une entrée supplémentaire dans le fichier `/etc/lilo.conf` (ou `/etc/grub.conf` suivant le chargeur utilisé). Dans le cas de LILO cela donne :

```
image=/boot/bzImage-2.4.20_kpreempt
        label=linux20_kp
        read-only
        root=/dev/hda1
```

Il faut ensuite valider cette nouvelle entrée en tapant :

```
# lilo
```

Configuration du noyau patché Low Latency

La configuration est très similaire sauf que l'on utilisera le suffixe `lowlat`. La validation de l'option CONFIG_LOLAT est également dans la rubrique *Processor type and features*.

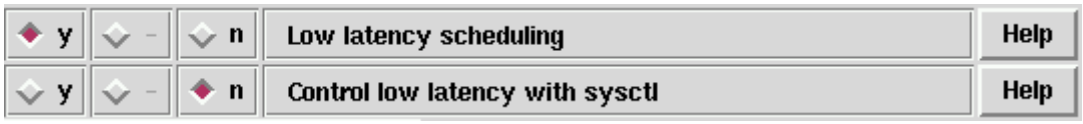

Figure 10. Validation de CONFIG_LOLAT

La deuxième option permet de débrayer dynamiquement l'option en manipulant le système de fichier virtuel `/proc`, nous ne l'utiliserons pas pour l'instant.

Outils de mesure utilisés

Pour effectuer les mesure, nous utilisons principalement l'outil Latency_graph-0.2 disponible sur http://www.linuxdj.com/latency-graph. Nous avons légèrement modifié le programme de test `testlatency.c` afin d'utiliser des sorties graphiques PNG (la libgd ne supporte plus le GIF pour des questions de licence) et de pouvoir spécifier le nom du fichier de sortie. Le principe de l'outil de mesure est de démarrer deux timers (30ms et 60 ms) et de mesurer la dérive de ceux-ci par rapport à la période attendue. Les mesures intéressantes sont obtenues en *stressant* le système, le plus souvent par des accès disque répétitifs. Dans notre cas, nous avons utilisé un petit script nommé `stress.sh` basé sur la commande `dd`.

```
#!/bin/sh
while [ 1 ]
do
        echo "Creating file..."
        dd if=/dev/zero of=foo bs=1M count=50
        sync
        echo "Removing file..."
        rm -f foo
        sync
done
```

Ce script crée puis efface de manière répétitive des fichiers de 50 Mo. Une fois le script lancé depuis une console, on pourra exécuter le test en tapant dans une autre console :

```
testlatency -f 2.4.20_stock_stress.png
```

qui produit une courbe de résultats au format PNG.

Nous utiliserons également ponctuellement le programme `realfeel2` inclus dans le package *amlat* disponible sur le site du patch Low latency. Ce programme produit un historique des temps de latence sur un nombre de points donné. On l'utilisera comme suit :

```
realfeel2 --samples 100000 2.4.20_stock_stress.hist
```

Mesures effectuées

Le système utilisé est basé sur un AMD Duron 900 MHz, équipé de 128 Mo de RAM, d'un disque IDE récent (mais bas de gamme !) et de la distribution Red Hat 7.3. Bien entendu, le résultat absolu de la mesure est très dépendant de l'architecture matérielle du système. Le but est ici d'observer une tendance dans les résultats.

Avant d'effectuer les tests, il convient de vérifier grâce à la commande `hdparm` si les

accès disque utilisent le DMA que nous avons validé coté noyau.

```
# hdparm /dev/hda

/dev/hda:
 multcount    = 16 (on)
 I/O support  =  0 (default 16-bit)
 unmaskirq    =  0 (off)
 using_dma    =  0 (off)
 keepsettings =  0 (off)
 nowerr       =  0 (off)
 readonly     =  0 (off)
 readahead    =  8 (on)
 geometry     = 3648/255/63, sectors = 58615258, start = 0
 busstate     =  1 (on)
```

Nous pouvons positionner le DMA ainsi que deux autres paramètres d'optimisation (cependant moins importants) avec la ligne de commande suivante:

```
# hdparm -d 1 -c 1 -u 1 /dev/hda

/dev/hda:
 setting 32-bit I/O support flag to 1
 setting unmaskirq to 1 (on)
 setting using_dma to 1 (on)
 I/O support  =  1 (32-bit)
 unmaskirq    =  1 (on)
 using_dma    =  1 (on)
```

Les options utilisées sont décrites ci-dessous dans un extrait du *man hdparm*:

- -d
  Disable/enable the "using_dma" flag for this drive.
- -u
  Get/set interrupt-unmask flag for the drive. A setting of 1 permits the driver to unmask other interrupts during processing of a disk interrupt, which greatly improves Linux's responsiveness and eliminates "serial port overrun" errors.
- -c
  Query/enable (E)IDE 32-bit I/O support. A numeric parameter can be used to enable/disable 32-bit I/O support

Noyau 2.4.20 standard

Nous effectuons tout d'abord une mesure sur un noyau 2.4.20 standard. Si le système n'est pas stressé, nous obtenons logiquement le résultat suivant.

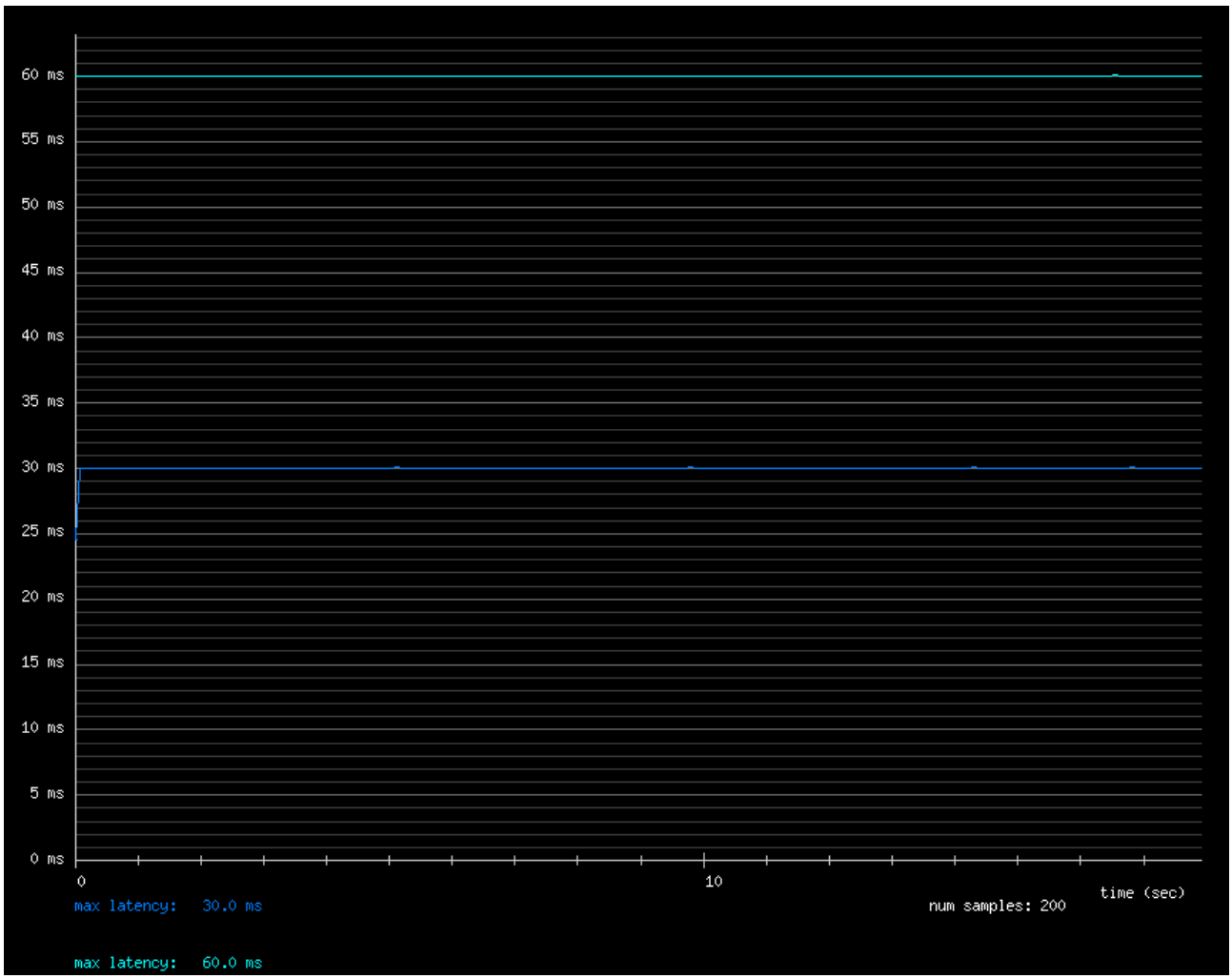

Figure 11. Mesure sur un système standard non chargé

Par contre lorsque le système est stressé par le script *stress.sh,* on observe des irrégularités sur les courbes (36 et 66 ms).

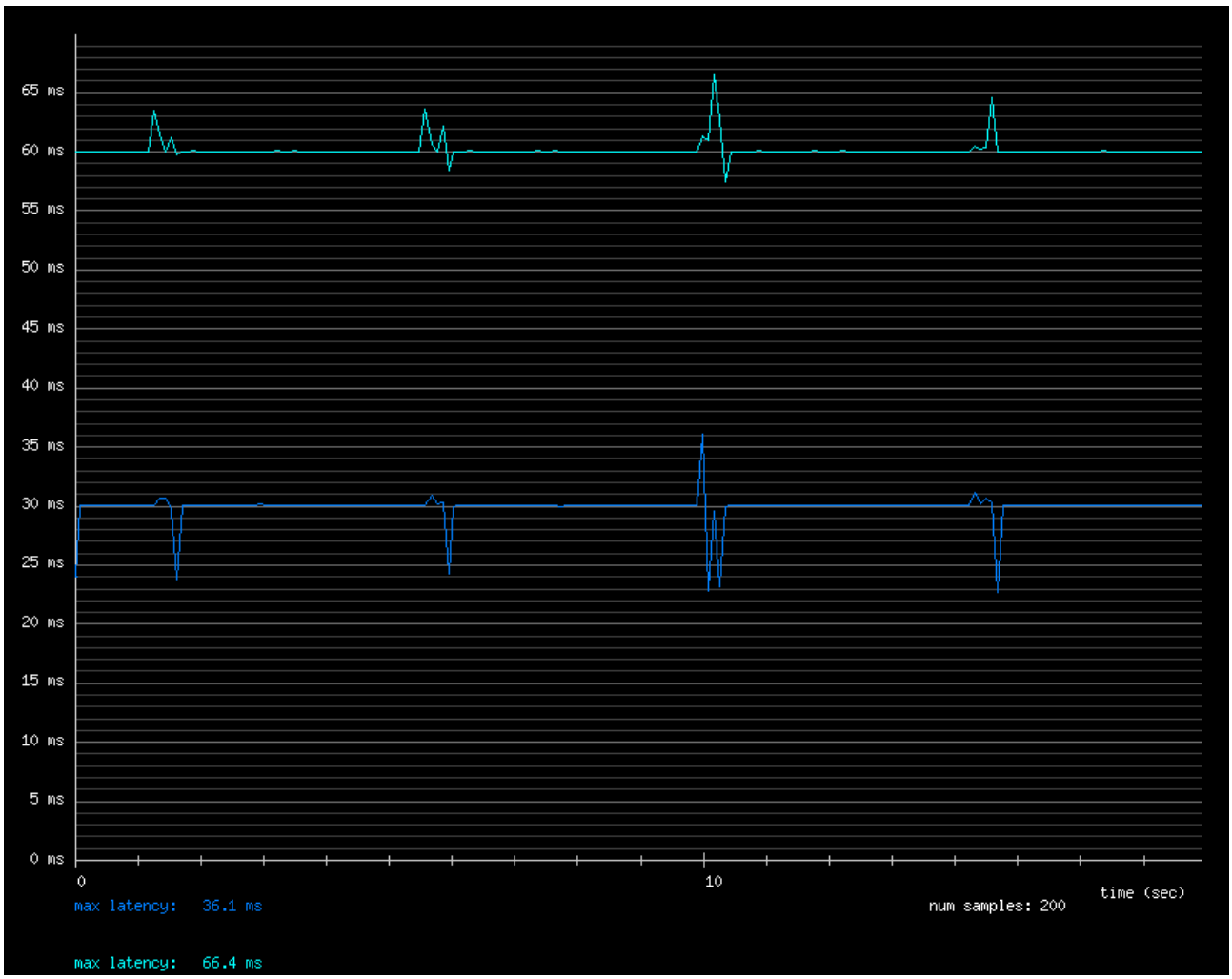

Figure 12. Mesure sur un système standard stressé

Pour se convaincre de l'utilité du DMA, la courbe ci-dessous présente le résultat de la mesure lorsque le DMA n'est pas validé par `hdparm`. Le résultat se passe de commentaires !

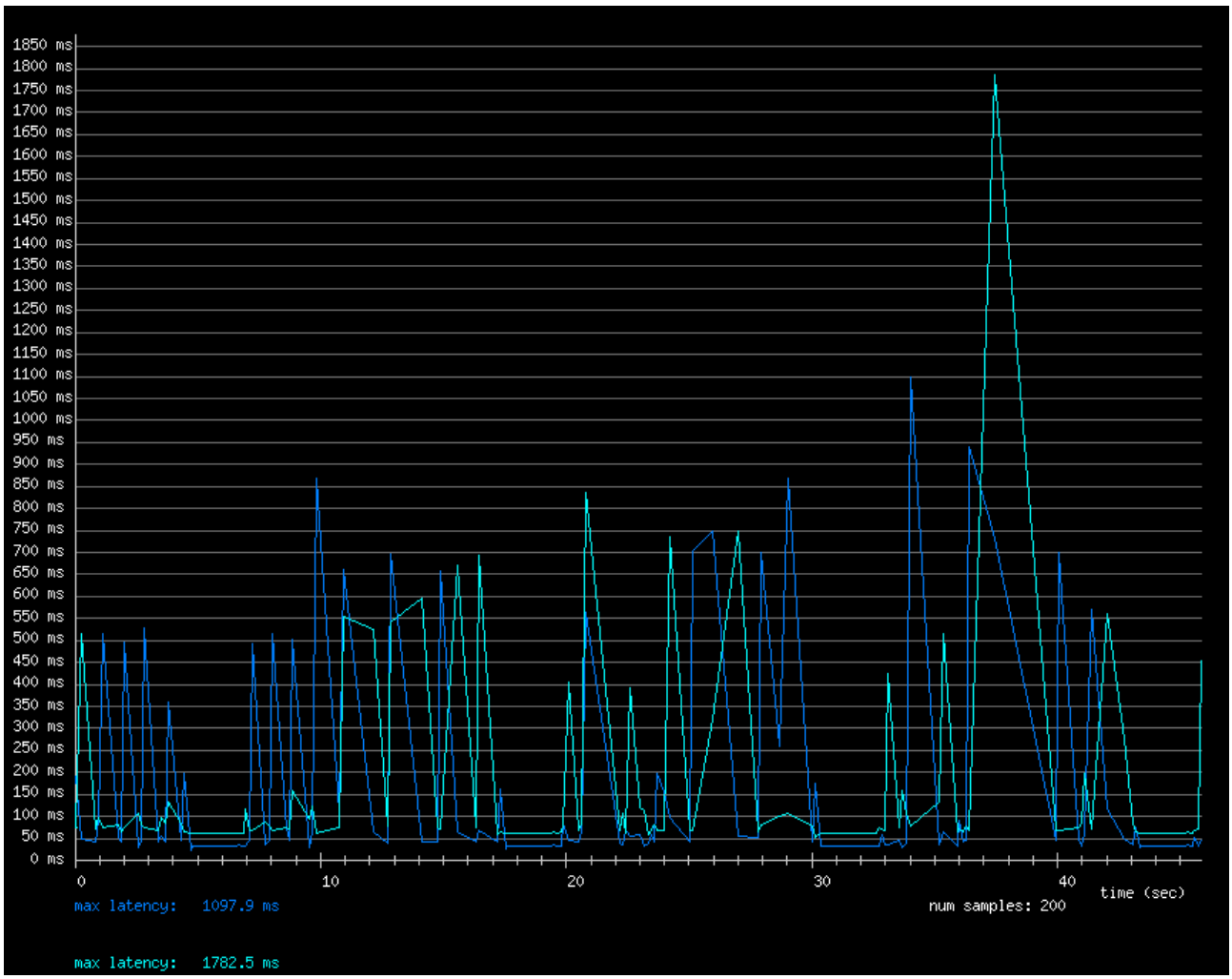

Figure 13. Mesure sur un système standard stressé sans DMA

Noyau 2.4.20 avec patch Preempt Kernel

Avec le patch preempt-kernel, nous obtenons la courbe suivante :

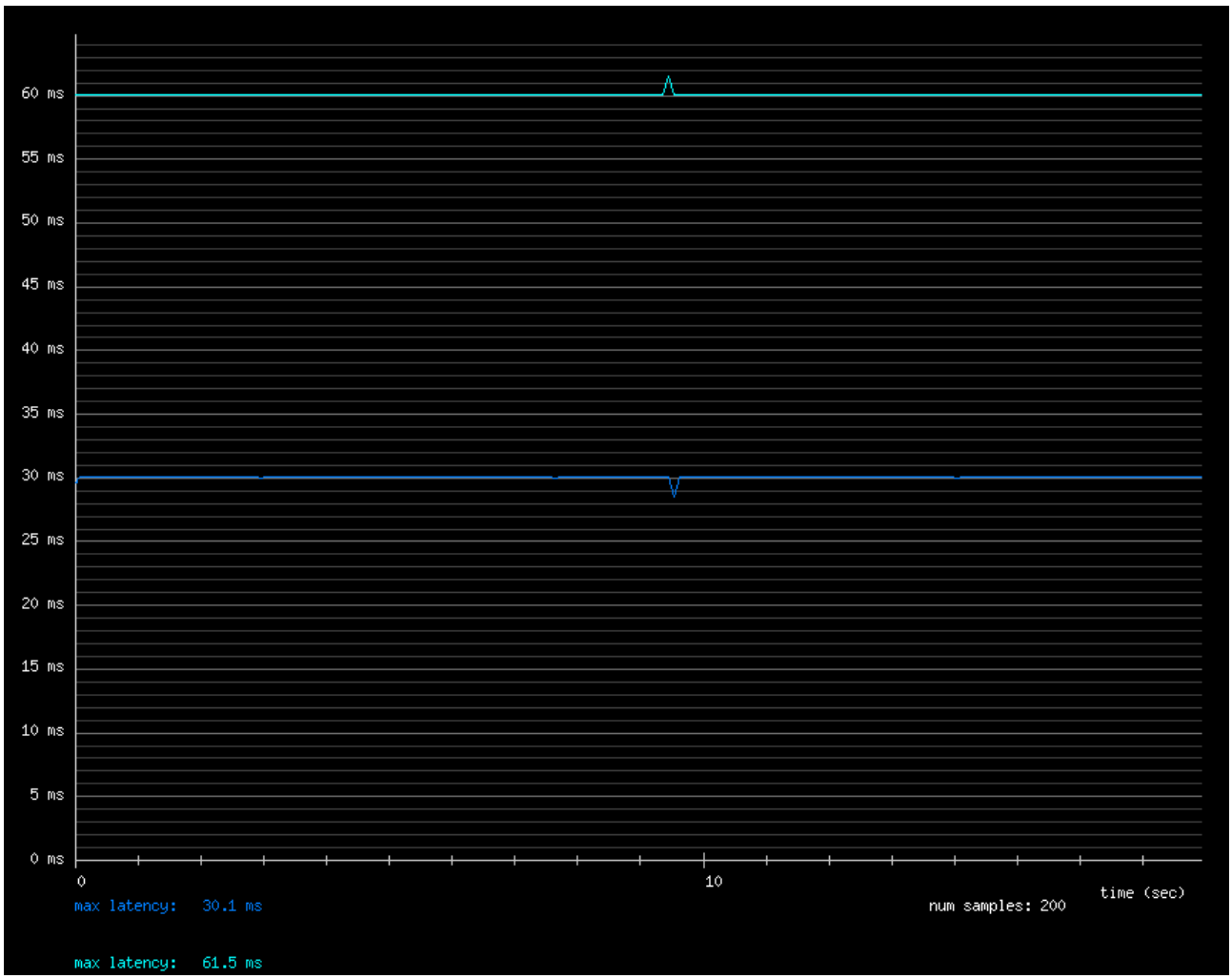

Figure 14. Mesure avec patch Preempt Kernel stressé

Le résultat est très bon malgré un petit "glitch" au milieu de la courbe. Si nous utilisons le programme `realfeel2` sur le même système, nous obtenons le fichier d'historique suivant sur 100000 itérations :

```
0.0 99828
0.1 118
0.2 5
0.3 2
0.5 3
0.6 2
0.7 1
1.1 1
2.4 1
2.5 5
2.6 5
2.7 2
2.8 4
2.9 2
3.4 1
4.0 1
6.3 1
9.5 1
11.4 1
11.9 1
12.0 1
12.1 1
12.2 1
```

```
12.3 1
12.5 1
12.7 1
12.9 1
13.3 1
13.4 2
13.6 1
13.7 1
14.3 1
15.1 1
15.2 1
```

Soit une latence maximale de 15 ms mais avec très peu de points supérieurs à 1 ms.

Noyau 2.4.20 avec patch Low Latency

Avec ce patch appliqué, nous obtenons la courbe suivante :

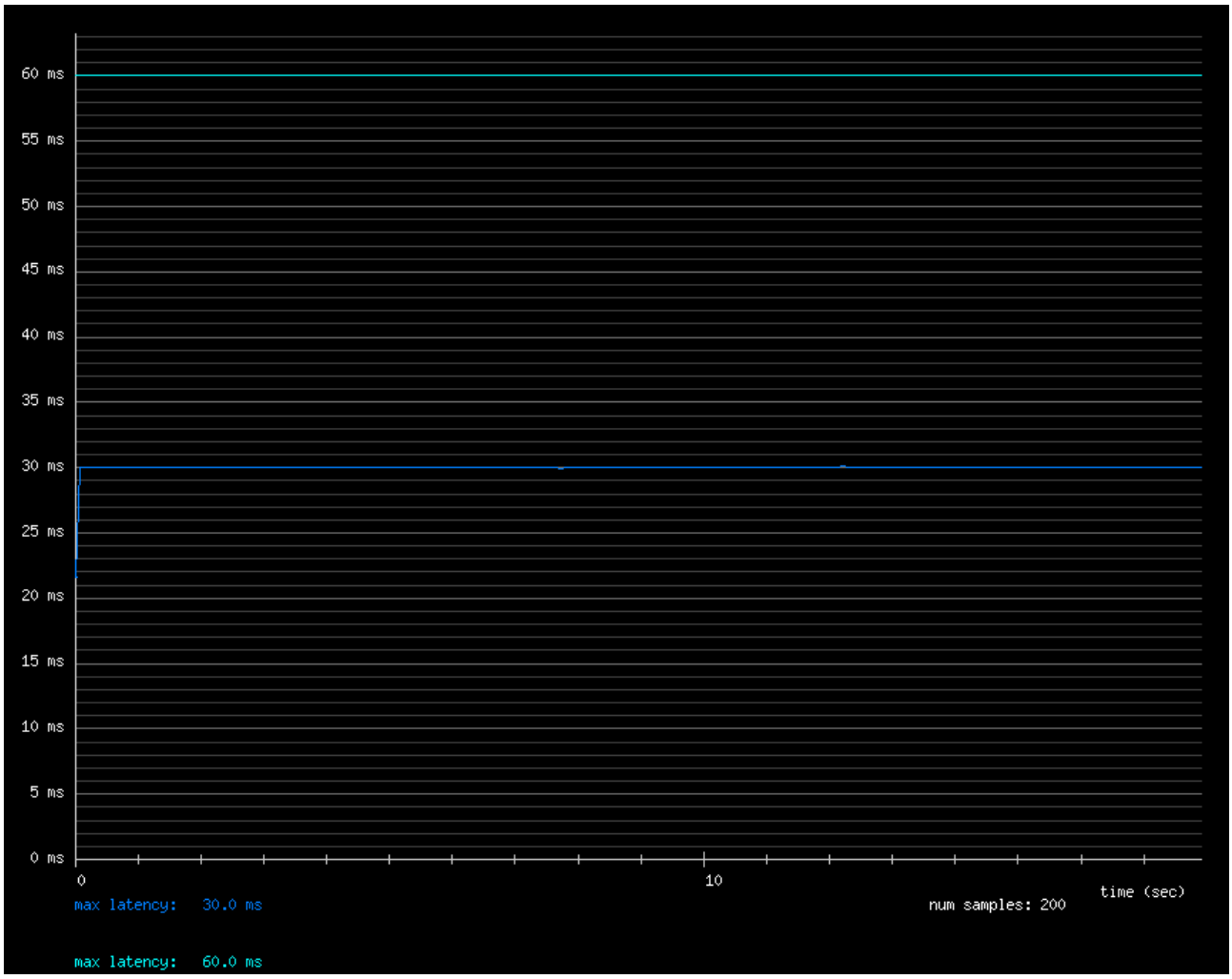

Figure 15. Mesure avec patch Low Latency stressé

Le résultat est parfait. Si nous utilisons le programme `realfeel2` sur le même système, nous obtenons le fichier d'historique suivant sur 100000 itérations :

```
0.0 99858
0.1 134
0.2 5
0.3 3
```

Soit une latence maximale de seulement 0.3 ms !

Test du programme square sur les noyaux patchés

Nous avons également effectué une mesure à l'oscilloscope lors de l'utilisation du programme `square` décrit précédemment. Dans le cas du noyau 2.4.20 modifié par le patch *Preempt Kernel* et subissant la même charge que pour les autres mesures, nous obtenons la courbe suivante, indiquant une latence maximale légèrement supérieure à 200 µs.

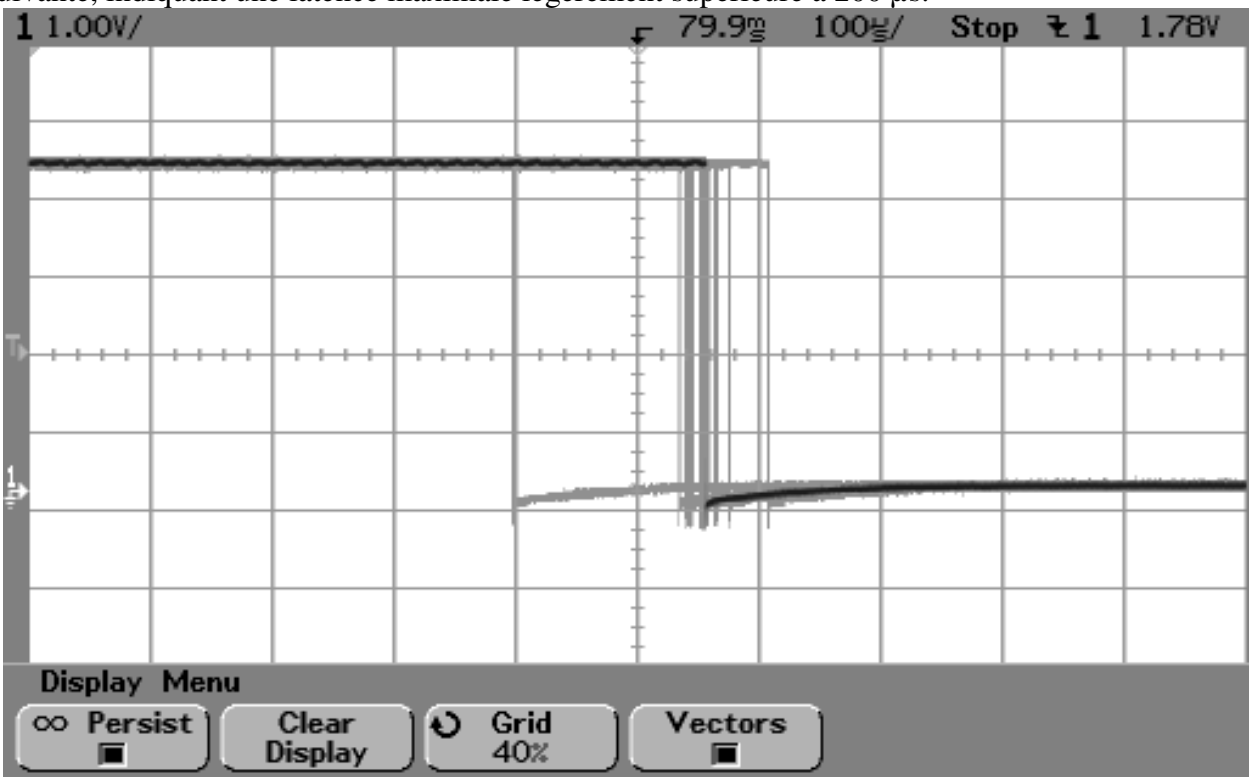

Figure 16. Exécution de square avec le patch Preempt Kernel

Dans le cas du patch L*ow Latency*, nous obtenons un meilleur résultat avec une latence maximale d'environ 80 µs.

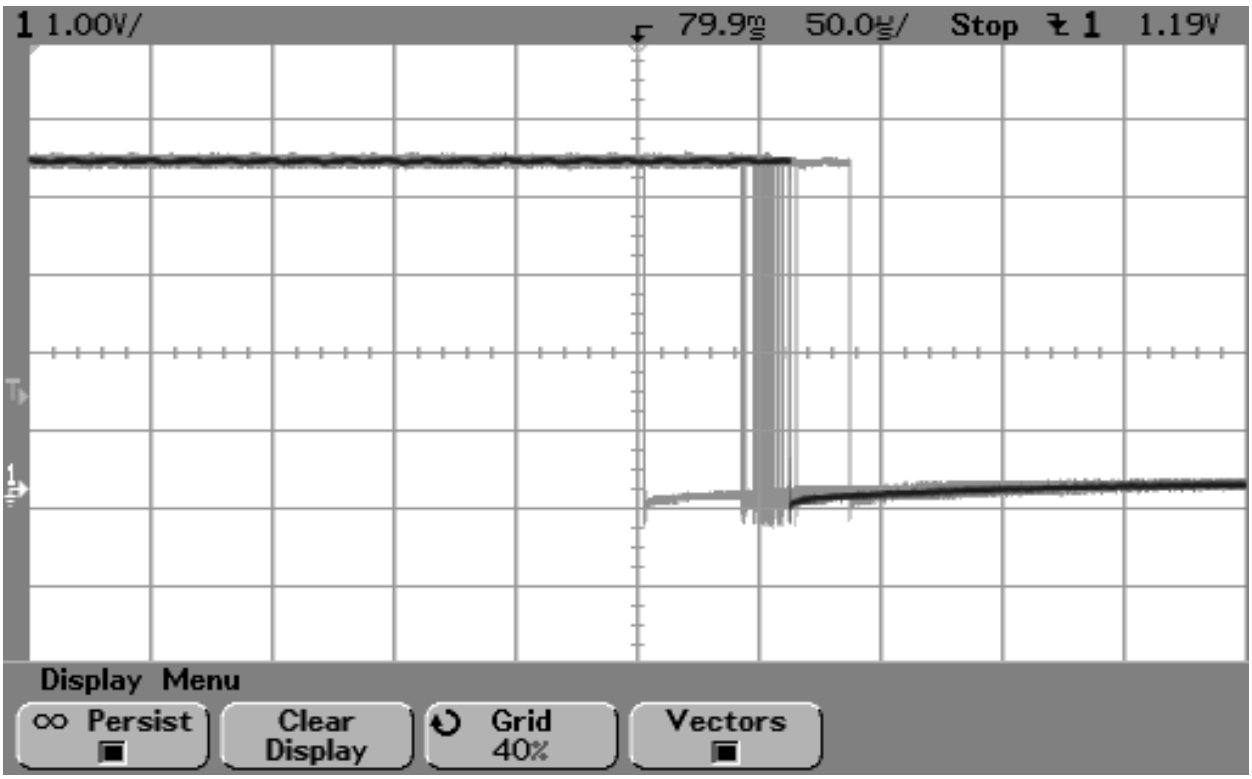

Figure 17. Exécution de square avec le patch Low Latency

Conclusion

Le patch *Low Latency* semble être plus efficace que *Preempt Kernel*, ce qui est également le résultat obtenu par d'autres études citées dans la partie bibliographique. Ceci étant dit, des études plus approfondies sur des cas réels permettront également de valider la stabilité des noyaux modifiés s'ils sont soumis à de fortes sollicitations pendant un grand intervalle de temps. Il conviendra plus tard de tester la version intégrée au noyau 2.6 lorsque celui-ci sera disponible soit probablement pas avant la fin de l'année 2003.

Le système RTLinux

Principe

Les patch testés précédemment permettent d'améliorer les temps de latence sur la noyau mais le concept se rapproche plus d'une amélioration de la qualité de service que du temps réel dur. Forts de cette conclusion des développeurs on choisi une stratégie totalement différente. Puisque le noyau LINUX n'est définitivement pas un noyau temps réel, la meilleure solution est de la faire cohabiter avec un noyau auxiliaire basé sur un vrai ordonnanceur temps réel à priorités fixes. Les tâches temps réel sont gérés par ce noyau et le traitement des autres tâche est délégué au noyau LINUX, lui-même considéré comme une tâche de plus faible priorité (ou *idle task*) par le noyau temps réel. Le schéma ci-dessous décrit l'architecture du système à double noyau. Les interruptions matérielles sont captées par le noyau temps réel. Si l'interruption ne correspond pas à une tâche temps réel, elle est traitée par le noyau LINUX.

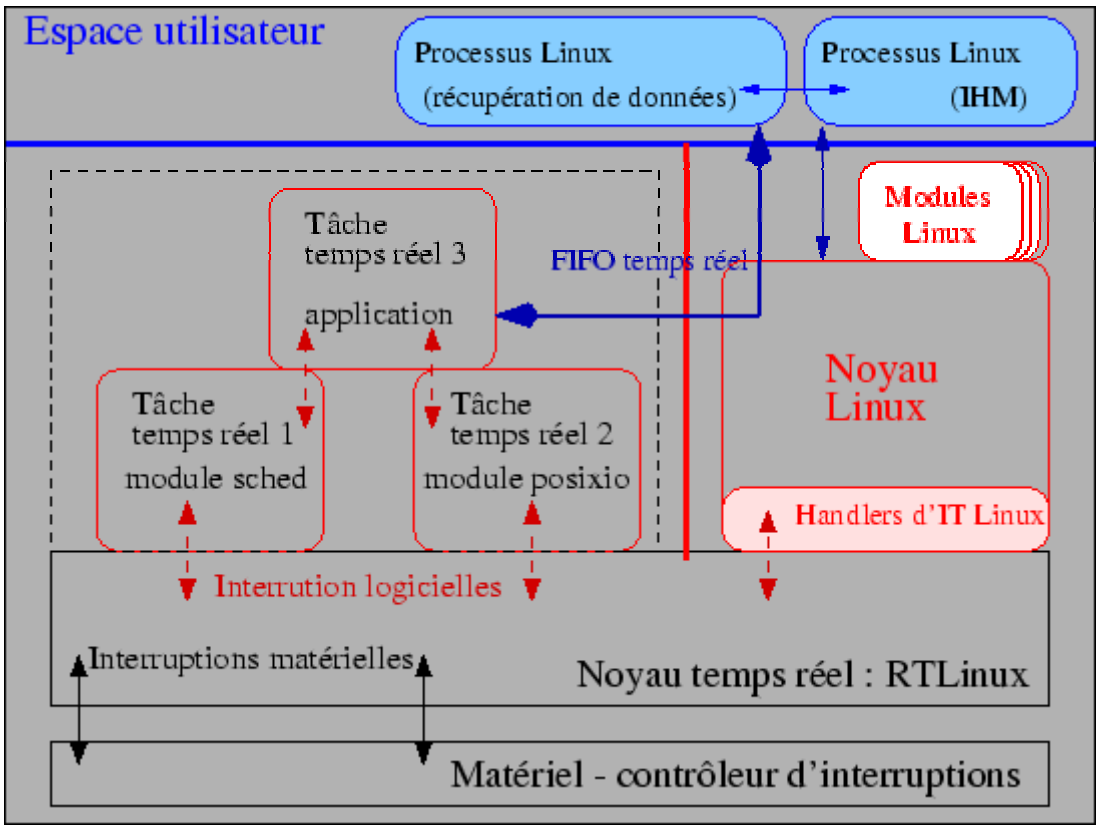

Figure 18. Système à double noyau

Historique, RTLinux et la GPL

Historiquement, le premier projet utilisant ce principe est RTLinux, projet initialement open source issu des travaux de Victor Yodaiken et Michael Barabanov à l'université du nouveau Mexique aux Etats Unis. RTLinux a depuis fait couler beaucoup d'encre dans la communauté open source car Yodaiken et Barabanov ont depuis créé la société FSMLabs (http://www.fsmlabs.com) chargée d'assurer le développement et le support commercial de RTLinux qui est un produit résolument orienté industrie. Le produit est désormais diffusé en version commerciale (RTLinux/Pro) dont les sources ne sont pas publiés (sauf aux clients) mais aussi sous GPL (appelée aussi RTLinux/Free) mais il est clair que FSMLabs ne fait quasiment plus évoluer la version GPL. De plus, l'espace réservé aux composants GPL sur le serveur FSMLabs se réduit de plus en plus au point de ne plus trouver la documentation en ligne pour la dernière version (3.2) de RTLinux/Free.

Le décalage entre la version commerciale et la version GPL n'est pas la seule raison de la grogne autour de RTLinux. Les fondateurs ont par contre déposé un brevet logiciel aux Etats -Unis concernant l'utilisation du noyau auxiliaire avec le noyau LINUX, ce qui n'est pas vraiment bien vu par la communauté, d'autant que le produit actuel a reçu le support de nombreux contributeurs à l'époque où RTLinux était uniquement un projet open source. La licence de RTLinux/Free « affranchit » cependant l'utilisateur des contraintes du brevet mais les projets doivent bien sûr être diffusés sous GPL. Le fait est que FSMLabs est quelque peu en porte à faux avec la GPL car de part la structure de RTLinux, les tâches temps réel sont développées dans l'espace du noyau LINUX, royaume de la GPL. La diffusion de modules en binaire est tolérée dans le cas de pilotes de périphériques car ceux-ci sont assez peu intimes avec les structures du noyau LINUX. Dans le cas d'un noyau temps réel auxiliaire, les conditions sont toutes autres et le cas RTLinux n'a pas fini de faire couler de l'encre virtuelle sur le net.

Structure et installation

Le fait est que RTLinux est cependant un bon produit, assez largement utilisé pour des produits sensibles dans le domaine des télécommunications ou des applications militaires. Le système est concis et bien documenté. L'interface de programmation est assez simple et permet de développer des tâches temps réel concises. Nous insistons sur le fait que seules les tâches temps réel sont gérées par le noyau auxiliaire, les autres tâches sont gérées par le noyau LINUX. Du fait que les tâches temps réel se situent dans l'espace noyau, RTLinux propose des systèmes de communication de type FIFO ou mémoire partagée entre les tâches temps réel et l'espace utilisateur dans lequel s'exécutent habituellement les applications.

La distribution RTLinux/Free est constituée de deux éléments :

1. Un patch à appliquer au noyau LINUX. Ceci implique donc que la distribution est liée à une version donnée du noyau LINUX sur laquelle ce patch s'applique correctement . La version 3.2 de RTLinux/Free fournit des patchs pour les noyau 2.4.0, 2.4.4, 2.4.17, 2.4.18 et 2.4.19. Il est cependant possible d'adapter le patch à une autre version si l'on est un peu entraîné :)
2. Un ensemble de modules LINUX constituant le noyau temps réel auxiliaire.

La procédure d'installation est décrite dans le fichier `doc/Installation.txt` de l'archive de documentation en ligne `rtldoc-3.1.tar.gz`, qui n'est malheureusement plus disponible pour la distribution 3.2. Les utilisateurs de cette dernière version (qui est très proche de la version 3.1) devront se contenter du répertoire `doc` de l'archive des sources contenant des fichier HOWTO associés aux distributions Red Hat 7.x et SuSE 8.0. Le principe de l'installation est cependant assez simple :

1. La première étape est l'extraction de l'archive des sources de RTLinux/Free dans le répertoire `/usr/src` afin de conduire à la création du répertoire `/usr/src/rtlinux`
2. Dans ce répertoire, on doit installer une version de noyau supportée par RTLinux (voir le répertoire `patches` de la distribution). Il est bien entendu FONDAMENTAL d'utiliser pour cela une arborescence de source officielle (provenant de ftp.kernel.org ou l'un de ses miroirs) et NON PAS une arborescence fournie par un éditeur de distribution (sinon le patch ne s'appliquera pas correctement).
3. Après application du patch adéquat, il faut ensuite paramétrer le noyau en fonction de la configuration matérielle, exactement comme on le ferait pour un noyau standard en utilisant une commande de type `make xconfig`. Il est d'ailleurs possible de récupérer le fichier `.config` d'un noyau 2.4 fonctionnel car le patch RTLinux/Free ajoute uniquement le paramètre `CONFIG_RTLINUX` qui est validé par défaut pour les architectures x86 et PowerPC. Notez qu'il est important de NE PAS valider le support APM (*Advanced Power Management*) qui perturbe le fonctionnement de RTLinux.

   Lorsque le noyau est compilé et installé en ajoutant une nouvelle entrée au chargeur LILO ou GRUB, il faut valider son bon fonctionnement en effectuant un redémarrage sur ce nouveau noyau.
4. La prochaine étape est la génération des modules spécifiques à RTLinux/Free, qui sont conformes à l'API des modules du noyau LINUX. Cette action est effectuée par une séquence très proche de celle utilisée pour le noyau LINUX, soit :

```
# make xconfig
# make dep
# make
# make devices
# make install
```

Dans un premier temps, il est conseillé d'utiliser les options par défaut au niveau du `make xconfig`.

5. On peut ensuite charger dynamiquement les modules en utilisant la commande `rtlinux`, soit :

```
# rtlinux start
Scheme: (-) not loaded, (+) loaded
 (+) mbuff
 (-) psc
 (+) rtl_fifo
 (+) rtl
 (+) rtl_posixio
 (+) rtl_sched
 (+) rtl_time
```

Notez que l'on peut passer en paramètre le nom d'un module RTLinux temps réel à exécuter soit par exemple :

```
# rtlinux start my_rt_module
```

Les programmes RTLinux étant des modules, le démarrage d'un programme correspond au chargement du module par la commande `insmod`, l'arrêt du programme correspond au « déchargement » en utilisant la commande `rmmod` ou bien `rtlinux stop`. Le chargement des modules peut être vérifié par la commande :

```
# rtlinux status

Scheme: (-) not loaded, (+) loaded
  (+) mbuff
  (-) psc
  (+) rtl_fifo
  (+) rtl
  (+) rtl_posixio
  (+) rtl_sched
  (+) rtl_time
```

La distribution RTLinux/Free inclut quelques programmes d'exemple dont un classique *Hello World* dans le répertoire `/usr/src/rtlinux/examples`.

Présentation des bases de l'API RTLinux au travers d'un exemple

L'API de programmation RTLinux est relativement simple à utiliser car elle comporte un relativement petit nombre de fonctions. Le point le plus important, au risque de se répéter, est la nécessiter de programmer les modules temps réel dans l'espace du noyau et non dans l'espace utilisateur. Cette contrainte est non négligeable car elle a de fortes implications sur la facilité de programmation (programmer dans l'espace du noyau nécessite une grande rigueur), les outils de debug disponibles (assez peu en fait dans l'espace du noyau) mais aussi la conformité par rapport à la GPL.

Pour notre exemple appelé *rtsquare*, nous allons reprendre le programme de génération de signal carré décrit au début de l'article. Cet exemple avait mis en évidence la forte influence de la charge du système sur la mesure du résultat effectuée grâce à un oscilloscope.

Un programme RTLinux est conforme à l'API des modules du noyau LINUX. Il inclut donc en premier lieu une fonction `init_module`, appelée lors du chargement du module et une fonction `cleanup_module` appelée lors du déchargement.

```
int init_module(void)
{
        pthread_attr_t attr;
```

```
        struct sched_param sched_param;

        /* Priorité à 1 */
        pthread_attr_init (&attr);
        sched_param.sched_priority = 1;
        pthread_attr_setschedparam (&attr, &sched_param);

        /* Création du thread applicatif */
        pthread_create (&thread,  &attr, bit_toggle, (void *)0);

        return 0;
}
```

La fonction `init_module` effectue ici les actions suivantes:

1. Positionnement du niveau de priorité de la tâche, qui est ici fixé à 1.

2. Création du *thread* « applicatif » qui dans notre cas va piloter le port parallèle du PC. Nous remarquons que la création du thread utilise la fonction `pthread_create`, conforme à l'API POSIX.

```
void cleanup_module(void)
{
        pthread_delete_np (thread);
}
```

La fonction `cleanup_module` se limite à la destruction du thread applicatif. Le code du thread est décrit ci-dessous, précédé par les définitions générales du programme.

```
#include <rtl_time.h>
#include <rtl_sched.h>
#include <asm/io.h>

/* Adresse du port parallèle */
#define LPT 0x378

/* Période de sollicitation de 50 ms */
int period=50000000;

/* Valeur envoyée sur le port parallèle */
int nibl=0x01;

/* Identifiant du thread applicatif */
pthread_t thread;

/* Corps du thread applicatif */
void * bit_toggle(void *t)
{
        /* On programme un réveil de la tâche toute les 50 ms */
        pthread_make_periodic_np(thread, gethrtime(), period);

        while (1){
                /* Ecriture de la valeur sur le port // */
                outb(nibl,LPT);

                /* Calcul de la valeur suivante (négation) */
                nibl = ~nibl;

                /* Attente du réveil */
                pthread_wait_np();
```

```
            }
}
```

La compilation du programme s'effectue grâce à un fichier `Makefile` adapté à l'environnement RTLinux/Free. Dans ce fichier, nous considérons que notre exemple est installé avec les autres exemples RTLinux/Free dans un sous-répertoire de `/usr/src/rtlinux/examples`.

```
all: rtsquare.o

include ../../rtl.mk

rtsquare: rtsquare.c
          $(CC) ${INCLUDE} -Wall -O2 -o rtsquare.c

clean:
          rm -f *.o
```

Lorsque le fichier `rtsquare.o` est généré, nous pouvons tester l'application en utilisant la commande `rtlinux start rtsquare`.

Si nous testons cet exemple dans les conditions de charge du système LINUX standard du début de l'article, nous obtenons le résultat ci-dessous à l'oscilloscope. La mesure du *jitter* donne la valeur de 25.6 µs comparés aux 17.6 ms du noyau LINUX standard, ce qui correspond environ à un rapport 1000.

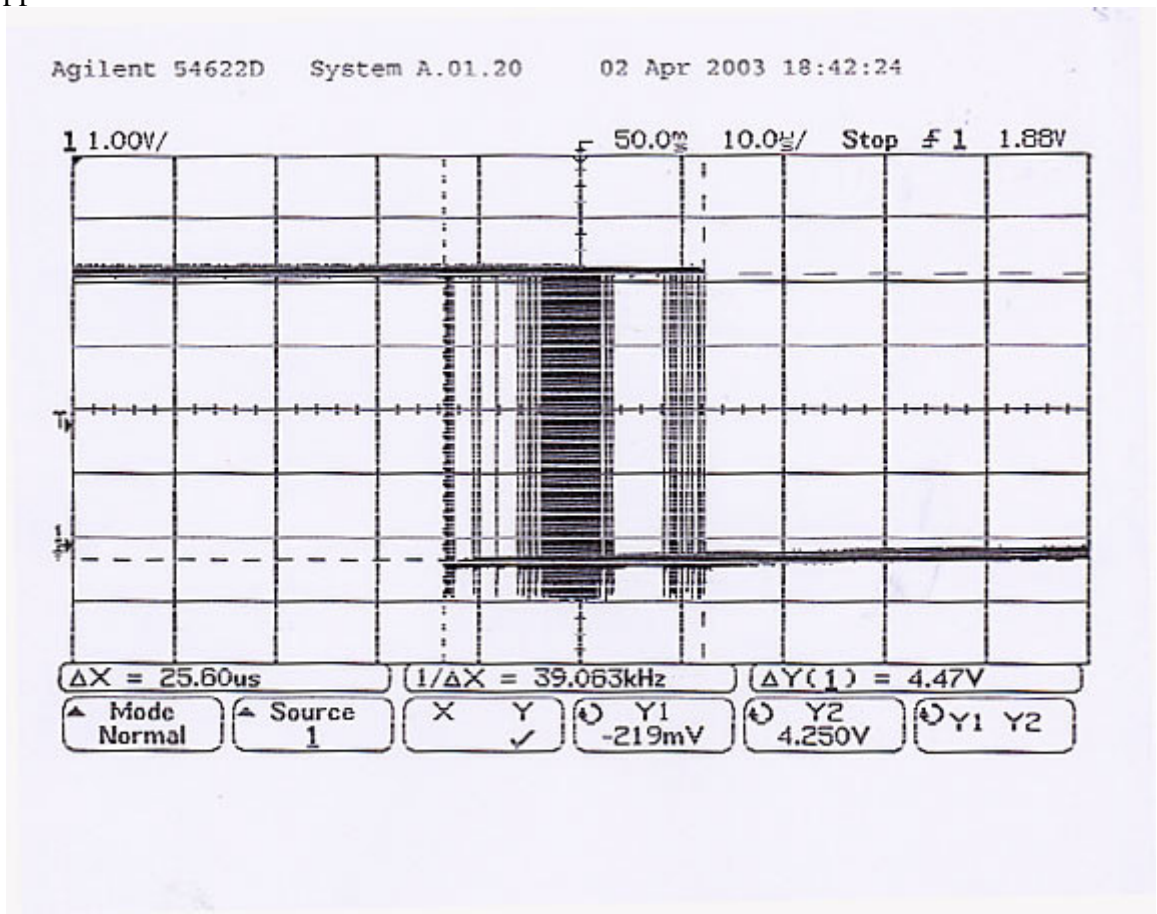

Figure 19. Résultat du test rtsquare sous RTLinux

Les FIFO de communication

De part la structure de RTLinux (tâches temps réel dans l'espace noyau et tâches classiques dans l'espace utilisateur), il est nécessaire de mettre en place un système de communication entre les deux éléments de l'application. Le plus souvent, cette communication est effectuée grâce à des FIFO `/dev/rtfX` (X variant de 0 à 63) créées lors de l'installation par la commande `make devices`. Coté modules noyau, la FIFO est manipulée à l'aide de fonctions spéciales comme `rtf_create` et associée à une fonction de traitement (ou *handler*) appelée lors de la réception d'un caractère sur la FIFO.

Dans l'espace utilisateur, la FIFO est vue comme un fichier généralisé classique (device) et peut donc être exploitée comme telle ce qui facilite grandement la mise au point des programmes. Le petit module ci-dessous permet d'utiliser la FIFO `/dev/rtf0` dans l'espace noyau. Coté utilisateur, on écrit sur la FIFO par une simple commande `echo` redirigée sur le device.

```
#include <rtl.h>
#include <time.h>
#include <pthread.h>
#include <rtl_fifo.h>

pthread_t thread;
char c;

int my_fifo_handler (unsigned int fifo)
{
  int err;
  char c;

  /* Lecture des caractères reçus */
  while ((err = rtf_get (0, &c, 1)) == 1) {
    rtl_printf("my_fifo_handler: got %x\n", c);
    pthread_wakeup_np (thread);
  }

  if (err != 0)
             return -EINVAL;
  else
    return 0;
}

void * start_routine(void *arg)
{
           struct sched_param p;
           p . sched_priority = 1;
           pthread_setschedparam (pthread_self(), SCHED_FIFO, &p);

           /* Attente infinie */
           while (1)
                      pthread_wait_np ();
           return 0;
}

int init_module(void) {
  int f;

  /* Création de la FIFO et de la fonction de traitement */
  f = rtf_create (0, 100);
  rtf_create_handler(0, &my_fifo_handler);

  return pthread_create (&thread, NULL, start_routine, 0);
}

void cleanup_module(void) {
```

```
  /* Détruit la FIFO */
  rtf_destroy(0);

  pthread_delete_np (thread);
}
```

Le système RTAI

Historique

Le projet RTAI (pour *Real Time Application Interface*) a pour origine le département d'ingénierie aérospatiale (DIAPM) de l'Ecole polytechnique de Milan (Politecnico di Milano). Pour des besoins internes, Paolo Montegazza du DIAPM entreprit de développer un produit inspiré de RTLinux mais intégrant quelques améliorations et corrections concernant en particulier les modes temps réel et la gestion des nombres flottants. Contrairement à RTLinux, RTAI ne fut pas développé dans le but de créer un produit commercial mais surtout afin d'éviter des coûts de licences sur des RTOS propriétaires comme QNX, alors utilisé au DIAPM.

Du fait de cette optique de développement interne, l'architecture actuelle de la distribution RTAI apparaît comme plus complexe tant au niveau de la définition de l'API (syntaxe et nombre d'appels) et de la position par rapport aux licences (GPL et LGPL). Il est clair aussi que RTAI est rapidement apparu aux yeux des responsables du projet RTLinux (Yodaken et Barabanov) comme un redoutable concurrent, sachant que RTLinux avait lui une vocation commerciale. Suite à la mise en place du brevet logiciel sur RTLinux, la position de RTAI est devenu quelque peu inconfortable et jusqu'à la refonte récente de RTAI autour du micro-noyau ADEOS (pour remplacer la couche initiale RTHAL), le brevet logiciel FSMlabs apparaissait comme une « épée de Damocles » au dessus de RTAI.

Similitudes et différences avec RTLinux : LXRT

La structure de RTAI est proche de celle de RTLinux. De même, les fonctionnalités du type FIFO de communication ou mémoire partagée sont également disponibles avec RTAI.

Cependant, les développeurs de RTAI ont récemment fait un effort particulier sur une extension appelée LXRT (*LinuX Real Time*) permettant de développer des tâches temps réel dans l'espace *utilisateur* et non plus dans l'espace noyau. Ce dernier point est extrêmement intéressant tant au niveau de la licence (il n'y a plus d'embarras avec la GPL contrairement au cas de RTLinux) que de la facilité de développement car nous savons que développer dans l'espace noyau est relativement plus complexe.

Au niveau des performances, nous verrons dans les exemples que les résultats par rapport au même programme développé dans l'espace noyau sont très honorables et ce bien que LXRT soit encore en cours d'évolution.

Structure et installation

La structure de RTAI étant proche de celle de RTLinux (noyau LINUX modifié co-existant avec un ordonnanceur temps réel) la procédure d'installation est proche de cette de RTLinux. Dans ce document, nous utiliserons la dernière version diffusée à ce jour sur http://www.rtai.org soit la 24.1.11. Lorsque l'archive est extraite (en général sur `/usr/src/rtai-24.1.11`), la procédure d'installation est décrite dans le fichier `README.INSTALL`.

La principale différence est la contrainte concernant le compilateur `gcc` à utiliser. Le fichier `GCC-WARNINGS` indique que le compilateur conseillé est le 2.95.3. Le compilateur 2.96

fourni avec la Red Hat 7.x n'est pas utilisable. Les compilateurs `gcc` 3.x sont susceptibles de marcher mais le résultat n'est pas garanti. Si l'on a un compilateur non supporté, le mieux est donc d'installer la version 2.95.3 sur un répertoire alternatif. Dans notre cas (Red Hat 7.3) nous avons installé ce compilateur compilé à partir des sources GNU sur l'arborescence `/usr/local`.

1. Comme pour RTLinux, la première étape consiste à construire un noyau intégrant le patch RTAI adapté. Le répertoire `patches` de la distribution 24.1.11 contient des fichiers de patch pour le dernier noyau LINUX stable à ce jour, soit le 2.4.20. C'est à ce moment là qu'il faut choisir entre l'ancienne technologie RTHAL et la nouvelle technologie ADEOS. Notez que le choix n'a pas d'influence sur l'API utilisable dans les applications RTAI, qui restent compatibles au niveau du code source. Pour nos essais, nous avons utilisé la nouvelle technologie ADEOS.
2. Avant de recompiler le noyau modifié, il convient de valider l'option adéquate en fonction de la technologie choisie dans le menu *General Setup* dans le cas de ADEOS ou *Processor type and features* pour RTHAL. Le patch effectue automatiquement la modification du champ `EXTRAVERSION` du fichier `Makefile` du noyau LINUX. Ce champ ne DOIT PAS être modifié ensuite sous peine d'échec de la suite de l'installation.
3. Après compilation et installation du noyau, il convient de redémarrer le système avec ce noyau afin de vérifier son bon fonctionnement.
4. La compilation des modules RTAI passe par la configuration via la commande :

   ```
   # make menuconfig
   ```

   au niveau du répertoire des sources.
5. Après validation des options par défaut, on poursuit la compilation par les commandes :

   ```
   # make dep; make; make install; make dev
   ```

   Le système est alors prêt à l'exécution des exemples dans le répertoire `examples` ou du programme de mesure de latence dans le répertoire `latency_calibration`.

Adaptation de l'exemple précédent à l'API de RTAI

Dans le paragraphe concernant RTLinux, nous avons présenté le petit programme `rtsquare` qui génère un signal carré périodique sur le port parallèle du PC. L'API de RTAI étant quelque peu différente de celle de RTLinux, le programme demande quelques modifications. Les lignes suivantes présentent la version RTAI du programme `rtsquare`. La principale différence est la moindre compatibilité avec la syntaxe POSIX, de ce fait, l'API RTAI apparaît comme plus spécifique que celle de RTLinux. Une API POSIX existe dans RTAI au niveau des tâches noyau mais celle-ci est incomplète et les efforts de développement se portent aujourd'hui plus vers LXRT que vers l'API noyau.

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/io.h>
#include <rtai.h>
#include <rtai_sched.h>

#define RT_PRIORITY     0
#define RT_STACK_SIZE   1000
#define LPT             0x378
#define PERIOD          50000000 /* 50 ms = 50 x 1000000 ns */

int nibl=0x01;
static RT_TASK task;
```

```
static void bit_toggle(int t)
{
  while (1){
    outb(nibl,LPT);
    nibl = ~nibl;
    /* Attente du réveil */
    rt_task_wait_period();
  }
}

int init_module(void)
{
  RTIME now, tick_period;

  /* A commenter */
  rt_set_oneshot_mode();

  /* Création de la tâche */
  if (rt_task_init(&task, bit_toggle, 0, RT_STACK_SIZE, RT_PRIORITY, 0, 0) < 0)
    rt_printk ("init_module: rt_task_init error\n");

  /* Démarrage du timer en calculant tout d'abord la valeur de la période en
     codage interne RTAI (RTIME)
  */
  tick_period = start_rt_timer(nano2count(PERIOD));
  now = rt_get_time();

  /* La tâche devient périodique sur la valeur donnée */
  if (rt_task_make_periodic(&task, now, tick_period) < 0)
    rt_printk("task_make_periodic error\n");

  return 0;
}

void
cleanup_module(void)
{
  stop_rt_timer();
  rt_task_delete(&task);
}
```

Au niveau du fichier `Makefile`, la syntaxe est similaire à celle de RTLinux :

```
CC=gcc
RTAISRC=/usr/src/rtai-24.1.11
KERNELSRC = /lib/modules/$(shell uname -r)/build

CFLAGS=-I$(RTAISRC)/include -I. -D__KERNEL__ -I$(KERNELSRC)/include -Wall
-Wstrict-prototypes -Wno-trigraphs -O2 -fno-strict-aliasing -fno-common -fomit-
frame-pointer -pipe -mpreferred-stack-boundary=2 -march=i686 -DMODULE

all: rtsquare_rtai.o

clean:
          rm -f *.o *~
```

Le chargement du module s'effectue grâce au script `rt_modprobe` disponible dans dans le répertoire `scripts` de la distribution RTAI, soit :

```
# rt_modprobe rtsquare_rtai.o
```

Pour arrêter la tâche temps réel, il suffit de « décharger » le module grâce à la commande `rmmod` ou bien à l'aide du script `rt_rmmod`. Le résultat de la mesure de latence

effectuée grâce à l'oscilloscope est donné sur la figure suivante. La valeur maximale de latence se situe aux environ de 25 µs ce qui est équivalent au résultat obtenu avec RTLinux.

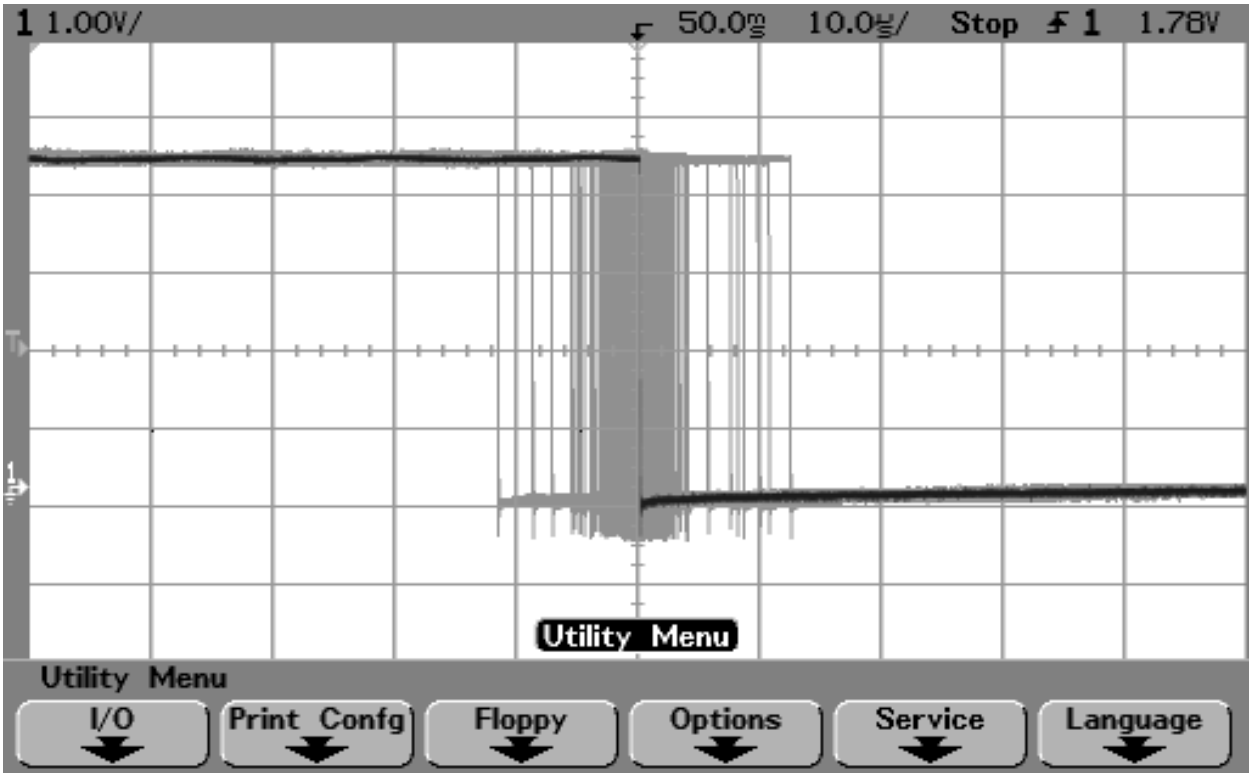

Figure 20. Résultat du test rtsquare sous RTAI (mode noyau)

Adaptation de l'exemple à LXRT

Un document d'introduction à la programmation sous LXRT est disponible sur http://people.mech.kuleuven.ac.be/~psoetens/portingtolxrt.html. Le principe de LXRT est de conserver les appels de l'API RTAI noyau tout en effectuant bien sûr quelques adaptations vu que nous avons maintenant affaire à un programme en mode utilisateur et non plus un module noyau. Le code source de la version LXRT du programme `rtsquare` est décrit ci-dessous. La structure du programme est très compréhensible. Un point important est l'utilisation de la fonction `rt_make_hard_real_time` permettant de placer la tâche sous le contrôle de l'ordonnanceur temps réel.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <sched.h>
#include <sys/mman.h>
#include <math.h>
#include <signal.h>
#include <sys/io.h>

#define KEEP_STATIC_INLINE
#include <rtai_lxrt_user.h>
#include <rtai_lxrt.h>
```

```
#define LPT             0x378
#define PERIOD          50000000 /* 50 ms = 50 x 1000000 ns */
int nibl=0x01;

static RT_TASK *task;

static void got_sigint ()
{
  rt_make_soft_real_time();
  rt_task_delete (task);
  exit (0);
}

main (int ac, char **av)
{
  RTIME now, tick_period;
  unsigned long task_name;

  /* Signal d'interruption du programme */
  signal (SIGINT, got_sigint);

  rt_set_oneshot_mode();
  task_name = nam2num("RTSQUARE_LXRT");
  mlockall(MCL_CURRENT | MCL_FUTURE);

  if (!(task = rt_task_init(task_name, 1, 0, 0))) {
    rt_printk ("init_module: rt_task_init error\n");
    exit (1);
  }

  tick_period = start_rt_timer(nano2count(PERIOD));
  printf ("tick_period = %ld\n",(long)tick_period);

  now = rt_get_time();
  if (rt_task_make_periodic(task, now, tick_period)) {
    printf("task_make_periodic error\n");
    exit (1);
  }

  /* L'appel à ioperm est nécessaire puisque nous sommmes en
     mode utilisateur */
  rt_allow_nonroot_hrt();
  if (ioperm (LPT, 1, 1) < 0) {
    perror("ioperm()");
    exit (1);
  }

  /* Passage en mode temps réel */
  rt_make_hard_real_time();

  while (1){
    outb(nibl,LPT);
    nibl = ~nibl;
    rt_task_wait_period();
  }
}
```

Le fichier `Makefile` utilisé est décrit ci-dessous :

```
CC=gcc
LIBLXRT=../../lxrt/lib/liblxrt.a

CFLAGS= -g -I /usr/src/rtai-24.1.11/include -I.  -I/usr/src/linux-2.4/include
-Wall -Wstrict-prototypes -Wno-trigraphs -O2 -fno-strict-aliasing -fno-common
```

```
-fomit-frame-pointer -pipe -mpreferred-stack-boundary=2 -march=i686

all: rtsquare_lxrt.o
          $(CC) -o rtsquare_lxrt rtsquare_lxrt.o $(LIBLXRT)
clean:
          rm -f *.o *~
```

L'exécution du programme nécessite le chargement des modules principaux de RTAI ainsi que du module LXRT. Le lancement du programme peut donc être effectué par un script similaire à celui décrit ci-dessous :

```
#!/bin/sh
insmod rtai
insmod rtai_sched
insmod rtai_lxrt_old
./rtsquare_lxrt
```

Vu que le programme est exécuté dans l'espace utilisateur, il pourra être interrompu par un signal classique, comme un SIGINT obtenu par la combinaison de touches Ctrl-C. Le résultat de la mesure à l'oscilloscope donne le résultat suivant. Le temps de latence maximal constaté est de 55 µs ce qui est moins bon qu'en mode noyau mais très acceptable pour un bon nombre d'applications.

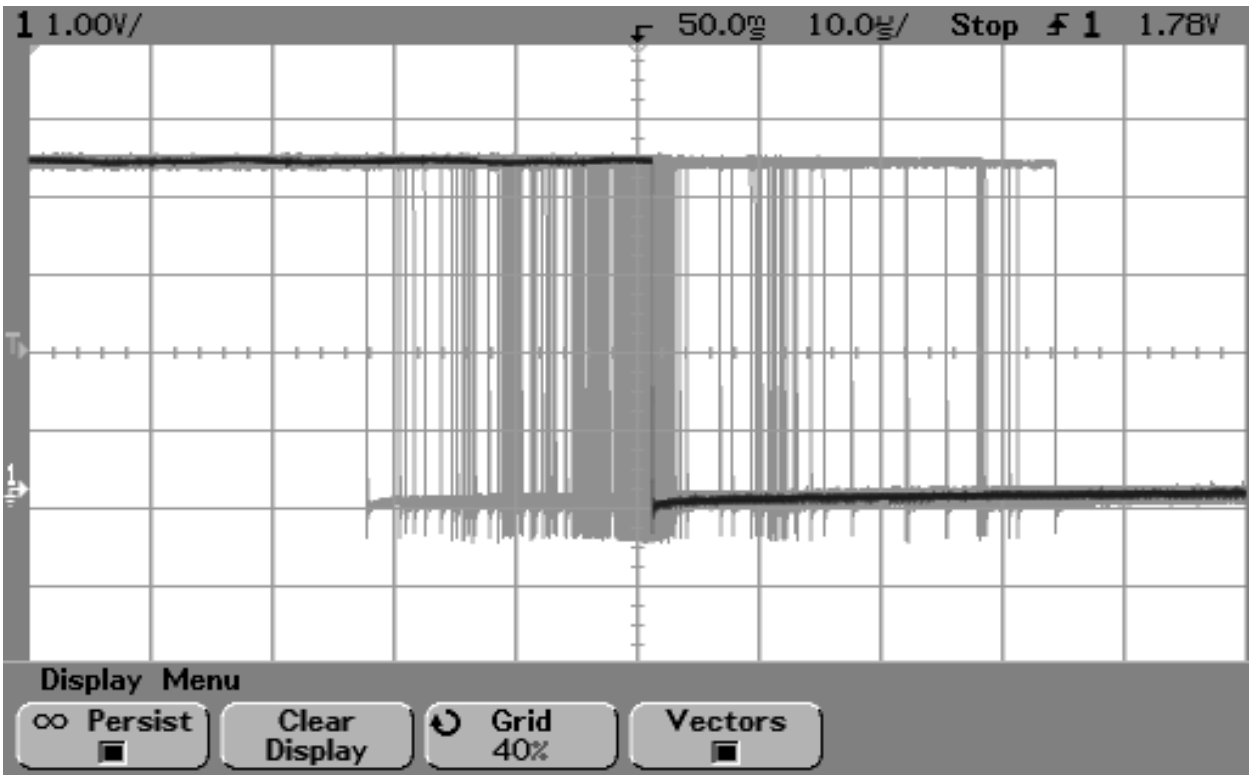

Figure 21. Résultat du test rtsquare_lxrt sous LXRT (mode utilisateur)

Cas particulier des microcontrôleurs (µClinux)

La distribution µClinux a déjà fait le cadre d'une présentation détaillée dans Linux Magazine de février 2002. µClinux est aujourd'hui l'une des distributions les plus utilisées comme Linux embarqué.

µClinux (prononcer « *you see Linux* ») est l'acronyme de *Micro Controller Linux.*
Le projet µClinux lancé en janvier 1998 est un portage de Linux version 2.0.x originellement sur des processeurs ne possédant d'unité de gestion mémoire MMU (*Memory Management Unit*).
C'est en fait à la base un portage de Linux sur microcontrôleur Motorola 68328 et dérivés que l'on trouvait par exemple dans les ordinateurs de poche (*Handheld*) PalmPilot.
Outre la famille Motorola 683xx, il existe maintenant des portages µClinux pour processeurs Motorola ColdFire, i960 d'Intel, ARM7TDMI et depuis peu NIOS d'Altera.
µClinux basé sur le noyau Linux 2.4.x est maintenant opérationnel.

Cartes cibles pour µClinux

Plateforme originelle : le kit µCsimm

Les responsables du projet µClinux ont proposé originellement une plateforme matérielle à des fins de tests : le kit µCsimm. Ce kit comprend un module qui se présente sous la forme d'une barrette SIMM à 30 broches. Le module µCsimm se compose de :

- Un microcontrôleur DragonBall EZ (68EZ328) basé sur un cœur Motorola 68000 à 16 MHz. Il intègre un contrôleur de DRAM, un port série (interface RS.232), une interface SPI, un contrôleur LCD (résolution QVGA), une sortie Timer et une sortie PWM.
- Jusqu'à 18 E/S parallèles.
- Une horloge Temps Réel – calendrier.
- De la mémoire dynamique DRAM de 8 Mo.
- De la mémoire FLASH de 2 Mo.
- Une interface Ethernet IEEE 802.3 10BaseT.

Le module µCsimm est commercialisé par Arcturus Networks Inc sous forme d'un kit de développement comprenant le module µCsimm, un carte d'assemblage / développement (kit µCgardener), le CD de développement, l'alimentation au prix de 495 USD (https://shop.arcturusnetworks.com/ucsimm.dragonball.ez.shtml). A des fins d'enseignement, il existe le kit µCacademix correspondant au kit précédent avec un manuel de TP utilisé par des universités américaines et vendu 295 USD (https://shop.arcturusnetworks.com/ucacademix.shtml).

Figure 22. *Module* ***µCsimm***

Plateformes dérivées

La société Arcturus Networks Inc propose aussi d'autres kits plus performants. On peut citer :

- Le kit µCdimm à base d'un microcontrôleur DragonBall VZ (68VZ328) à 33 MHz (https://shop.arcturusnetworks.com/ucdimm.dragonball.vz.shtml).
- Le kit µCdimm à base d'un microcontrôleur ColdFire 5272 (https://shop.arcturusnetworks.com/coldfire5272.shtml).
- Le kit ARM7TDMI à base du processeur ATMEL AT91 ARM7TDMI (https://shop.arcturusnetworks.com/arm7tdmi.development.kit.shtml).

Autres plateformes

µClinux peut aussi être mis en oeuvre sur des cartes (d'évaluation) à base de processeur ColdFire. On peut citer les cartes suivantes :

- Les cartes Arnewsh SBC5206 et SBC5307.
- Les cartes Motorola MCF5206eLITE, MCF5206C3, MCF5307C3, MCF5407C3 (1000 euros environ), MCF5272C3, MCF5282C3.
- Les cartes Lineo eLIA 5307, NETtel et SecureEdge.
- La carte Netburner CFV2-40.

Extension temps réel pour µClinux

L'extension temps réel de µClinux ne fait malheureusement pas partie intégrante du projet pour des raisons évidentes de maintenance : prise en compte de la version du noyau LINUX avec la prise en compte de la version de l'extension temps réel (RTAI, RTLinux) avec la prise en compte de la version du portage de µClinux pour un processeur cible particulier !

Quelques portages existent pour des versions de µClinux et processeurs cibles bien précis :

1. Un portage de RTLinux sous µClinux existe avec les restrictions suivantes : noyau 2.0.38 et processeur Motorola DragonBall. Le patch pourra être récupéré à l'adresse suivante : https://courseware.vt.edu/users/binoy/4984/project1.html.
2. Un portage de RTAI sous µClinux existe aussi dans le cas particulier suivant : processeur Motorola Coldfire 5272. Le portage a été intégré dans la distribution officielle RTAI 24.1.11 dans la branche `arch/m68knommu` de la distribution RTAI. La procédure de compilation est décrite dans le fichier `README.m68knommu`. L'extension temps réel RTAI doit être mise en oeuvre sur une carte Motorola MCF5272C3. Des informations supplémentaires données par B. Kuhn responsable du portage sont disponibles à l'adresse http://www.uclinux.org/~bkuhn/Platforms/Coldfire/tarifa/20020227/README.

Des mesures de temps de latence dans ce cas très particulier ne sont pas disponibles car le kit µCsimm des auteurs est tombé en panne pendant les tests :(Les résultats sont à priori similaires à ceux trouvés avec RTAI sur PC.

Conclusions et perspectives

Les exemples présentés dans cet article indiquent clairement que LINUX occupe une place de plus en plus importante dans le monde des systèmes d'exploitation temps réel.

LINUX n'est pas fondamentalement un système d'exploitation temps réel car trop généraliste. Par application de différents patchs (*low latency, preempt kernel*), il est possible d'avoir un système LINUX temps réel mou. Il est possible d'avoir un système LINUX temps dur en utilisant

les extensions temps réel RTLinux et RTAI. Le choix final se fera en fonction des contraintes temporelles imposées par le processus à contrôler depuis LINUX en prenant aussi en compte la complexité de mise en oeuvre dans chaque cas (configuration du noyau, écriture de l'application temps réel).

Nous nous sommes attachés à la description des technologies les plus utilisées (et également disponibles en open source) mais d'autres systèmes plus ou moins propriétaires s'appuient également sur LINUX pour offrir des fonctionnalités temps réel. Nous pouvons citer par exemple le système Jaluna 2, récemment présenté par l'ex-équipe de Chorus Systèmes et qui présente une architecture extérieurement similaire à celle de RTLinux ou RTAI (noyau LINUX associé à un noyau auxiliaire temps réel).

Du coté de RTAI, nous pensons que l'évolution vers ADEOS et LXRT permettra rapidement (d'ici quelques mois) de faciliter la migration d'un environnement propriétaire vers un système entièrement open source sans rencontrer les soucis techniques et légaux du développement industriel dans l'espace du noyau LINUX.

Remerciements

Philippe Gerum (rpm@xenomai.org) contributeur majeur des nouvelles versions de RTAI intégrant la technologie ADEOS. Architecte « nettoyeur » de la future distribution RTAI :)

Bibliographie

- Site web de Patrice Kadionik sur

  http://www.enseirb.fr/~kadionik
- Ouvrage *Linux Embarqué* par Pierre Ficheux aux éditions Eyrolles
- *Low-Latency 2.4.x with ALSA HOWTO* par Paul Winkler sur
  http://www.djcj.org/LAU/guide/Low_latency-Mini-HOWTO.php3
- *Linux scheduler latency* par Clark Williams sur
  http://www.linuxdevices.com/articles/AT8906594941.html
- *Realfeel Test of the Preemptible Kernel Patch* par Andrew Webber sur
  http://www.linuxjournal.com/article.php?sid=6405
- *Can Linux be a real-time operating system ?* par Kevin Morgan sur
  http://www.linuxdevices.com/articles/AT5152980814.html
- *Real-Time and Embedded Guide* par H. Bruyninckx.
  http://people.mech.kuleuven.ac.be/~bruyninc/rthowto
- Site de FSMlabs/RTLinux.
  http://www.fsmlabs.com
- Site du projet RTAI.
  http://www.rtai.org
- Site du projet ADEOS.
  http://www.nongnu.org/adeos
- Le temps réel par William Blachier sur
  http://www-ensimag.imag.fr/cours/Systeme/docs.etudiants.html