Rapport de Stage de fin d'Etudes :

Finalisation du portage de PUT et portage de MPI sur la machine MPC/Linux.

Non confidentiel

Emmanuel Dreyfus

Stage effectué au Laboratoire d'Informatique de l'Université de Paris 6 (LIP6)
4 place Jussieu, 75252 Paris Cedex 05, sous l'encadrement de Olivier Glück

(Février 2000 - Juin 2000)

Télécom INT **Stage ingénieur 1999/2000**

Nom du stagiaire : Emmanuel Dreyfus

Option : Systèmes et Architectures Répartis et Parallèles

Entreprise : Laboratoire d'Informatique de l'Université de Paris VI (LIP6)

Dates : Février - Juillet 2000

Sujet : Finalisation du portage de PUT et portage de MPI sur la machine MPC/Linux.

Résumé

Ce stage s'inscrit dans le cadre du projet de recherche MPC, qui a démarré en 1995, sous la direction d'Alain Greiner. Il a pour but la réalisation d'une machine parallèle coût réduit, dont les nœuds de calcul sont des PC interconnectés par un réseau haut débit.

Mon travail au cours du stage a consisté pour une première partie à finaliser le portage sous Linux des couches logicielles exploitant les cartes réseaux. Ces couches étaient initialement développées pour FreeBSD. La deuxième partie du travail consistait en le portage de l'interface de programmation MPI sur la machine MPC/Linux.

Abstract

This training period is a part of the global research project called MPC, which started in 1995, under the direction of Pr. Alain Greiner. The project's aim is to build a low cost parallel computer, based on standard PC as processing node interconnected with a high performance network.

My role in the project was first to port to Linux the software in charge of operating the network boards. This software was first written for FreeBSD. Secondly, I had to work on the port to MPC/Linux of the MPI programming interface.

Remerciements

Je remercie l'ensemble de l'équipe de l' ASIM pour m'avoir accueilli dans leur département. En particulier, je remercie Alain Greiner pour m'avoir permis d'effectuer ce stage.

Egalement, je remercie mes responsables de stage de l'INT, Daniel Millot et Philippe Lalevée, pour le soutient qu'ils m'ont apporté tout au long de mon stage.

Enfin, je remercie Olivier Glück qui m'a encadré durant ce stage, ainsi que Alexandre Fenyö, Cyril Spasevski, et Manuel Bouyer, qui m'ont aidé à résoudre nombre de problèmes tournant autour du projet MPC.

Table des matières

Table des figures

I Introduction et contexte du stage

Cette partie a pour but de situer le contexte du stage. En particulier, il présente l'organisme d'accueil, ainsi que le projet sur lequel j'ai travaillé pendant cinq mois.

I.1 Présentation de l'organisme d'accueil

Le Laboratoire d'Informatique de l'Université de Paris 6 (LIP6) est une Unité Mixte de Recherche de l'Université Pierre et Marie Curie (Paris 6) et du Centre National de la Recherche Scientifique (CNRS). Environ 300 personnes travaillent au LIP6, assurant des travaux d'enseignement et de recherche dans le domaine de l'informatique. Plus de la moitié de cet effectif est composé d'étudiants préparant leur thèse de doctorat.

Le LIP6 est réparti sur deux sites : d'une part le campus de Jussieu (Paris V^{ème}), et d'autre part l'annexe située rue de l'Amiral Scott (Paris XV^{ème}). Il se divise en 9 départements, répartis sur ces deux sites :

ANP	Algorithmique numérique et parallélisme
APA	Apprentissage et acquisition de connaissances
ASIM	Architecture des systèmes intégrés et micro-électronique
CALFOR	Calcul formel
OASIS	Objets et Agents pour Systèmes d'Information et de Simulation
RP	Réseaux et performances
SPI	Sémantique, preuve et implantation
SRC	Systèmes répartis et coopératifs
SYSDEF	Systèmes d'aide à la décision et à la formation

Plus d'informations sur le LIP6 : http://www.lip6.fr

Le département ASIM se consacre à la conception de circuits intégrés, au développement de logiciels de CAO pour la VLSI, et l'étude des architectures matérielles des systèmes. Il regroupe 65 personnes sur le campus de Jussieu.

Les enseignements dispensés par le départements sont

- Le DEA en architecture des Systèmes Intégrés et Micro-Electronique (ASIME)
- Le DESS de Circuits Intégrés et Systèmes Analogiques Numériques (CISAN)
- Les cours d'option architecture de la maîtrise d'informatique de Paris VI
- Les cours d'option MEMI de la maîtrise EEA de Paris VI

Les activités de recherche du département se répartissent autour de trois projets :

- Le projet MPC
- Le projet d'indexation multimédia
- Le projet CAO de circuits et systèmes (Alliance)

Plus d'information sur l'ASIM : http://www-asim.lip6.fr

I.2 Le projet MPC

Dans le cadre de l'activité de design de circuit intégrés du département, un circuit routeur pour réseau hautes performances, nommé RCube a été développé. Ce circuit se destine a plusieurs applications, telles que les commutateurs gigabit ethernet ou les machines parallèles. C'est cette sur cette deuxième voie que s'est bâti le projet MPC.

Le projet MPC regroupe des chercheurs de l'ASIM, du PRiSM de l'Université de Versailles Saint Quentin (UVSQ), du LARIA de l'Université de Picardie Jules Verne, et des départements informatiques de l'Institut National des Télécommunications d'Evry (INT-Evry) et de l'Ecole Nationale Supérieure de Télécommunications de Paris (ENST-Paris).

Le but du projet est de développer une machine parallèle à coût réduit, en utilisant du matériel standard sur le marché, à savoir des compatibles PC, auxquels on fournit un réseau hautes performances. Les compétences mises en œuvre vont du design de circuits intégrés au développement de composants logiciels pour exploiter les circuits. C'est dans le cadre de cette activité de développement logiciel que j'ai effectué mon stage de fin d'études d'ingénieur INT.

II Introduction : Généralités sur le parallélisme

Cette partie fait une brève introduction au enjeux du parallélisme, et explique les motivations du projet MPC.

Depuis la fin des années 60, les circuits intégrés des systèmes informatiques évoluent selon la loi de Moore, c'est à dire en suivant un doublement du nombre de transistors intégrés sur une surface donnée tous les 18 mois, pour un prix restant inchangé. Aux premiers jours, ces progrès en miniaturisation des circuits intégrés permettaient d'obtenir un doublement de performances sans augmentation du coût, sur une échelle de 18 mois. Grossièrement, plus un circuit est miniaturisé, moins il dissipe de chaleur, et plus on peut lui imposer une haute fréquence de fonctionnement.

Mais cette évolution à des limites. A cause des difficultés d'intégrations de circuits de plus en plus miniaturisés, la loi de Moore a commencé a subir une inflexion en 1992. De nos jours, pour continuer à augmenter la puissance des machines, on doit ajouter le parallélisme à la miniaturisation. L'introduction du parallélisme permet d'utiliser plusieurs circuits d'une puissance donnée en même temps, afin d'obtenir une puissance de calcul Supérieure à ce qu'un seul circuit aurait pu faire. Dans les micro-ordinateurs, on retrouve le parallélisme un peu partout : pipelines et unités de calcul multiples dans les processeurs, contrôleurs ou coprocesseurs déchargeants le processeur de certaines taches, systèmes multiprocesseurs, etc...

Pour obtenir plus de parallélisme, les approches sont nombreuses. Parmi elles, le cluster de type "grappe de PC" est assez populaire aujourd'hui. Dans de tels systèmes, on utilise des compatibles PC, qui sont des machines peu chères, comme nœud de calcul. Chaque nœud possède sa propre mémoire, et exécute ses instructions sur ses données propres. On est en présence d'un système Multiple Instruction Multiple Data à Mémoire Distribuée (MIMD-MD)

Ce choix d'architecture permet d'éviter l'utilisation de machines dédiées coûteuses : une machine Single Instruction Multiple Data (SIMD) requiert un processeur spécifique (donc cher), un système à mémoire partagée demande des mécanismes complexes de gestion de la cohérence des caches et de concurrence d'accès à la mémoire. Utiliser un réseau de PC permet d'éviter cela.

Mais ce choix présente aussi un inconvénient majeur : les PC ne sont pas fais pour faire du calcul parallèle. Si du côté du processeur, grâce à la forte demande en puissance de calcul pour les jeux vidéo, on a des performances tout à fait correctes, du côté des communications entre processeurs, les moyens peu chers (réseaux ethernet, token ring...) dont on dispose sur un PC ne sont pas à la hauteur.

C'est sur ces constatations que se positionne le projet MPC : D'une part, aujourd'hui, le processeur d'un PC a des performances permettant de concurencer un nœud de calcul d'une grosse machine parallèle. D'autre part, l'une des plus fortes limitations aujourd'hui dans les machines parallèles de type "grappes de PC", c'est le manque de performances des moyens de communications.

III Architecture générale de la machine MPC

Cette partie est consacrée à la description du fonctionnement général de la machine MPC. Elle décrit les motivations des choix de fonctionnement (le mode zéro copie), la façon dont les données sont transférées, ainsi que la façon dont le logiciel peut gérer tout cela.

III.1 Objectif : zéro copie

Les problèmes de recopie mémoire sont parmi ceux qui nuisent le plus aux performances des communications entre machines dans un réseau de PC. La Fig. 1 illustre les différentes recopies qui se produisent lors d'un échange quelconque sur le réseau

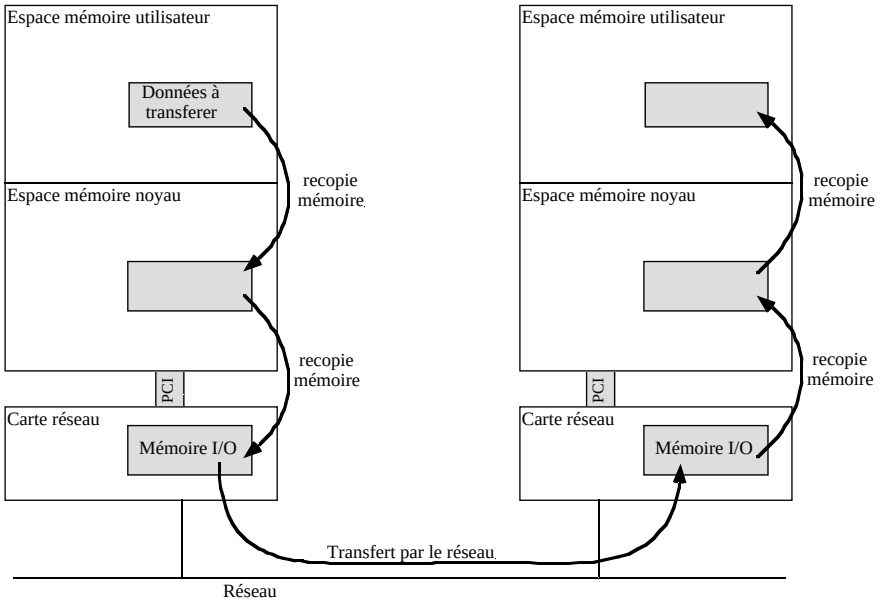

Fig 1: Différentes recopies mémoires mises en jeu lors d'un transfert de paquet sur un réseau classique (par exemple ethernet). On a parfois des recopies suplémentaires pour l'enpaquetage.

Typiquement, un processus utilisateur utilise une primitive d'envoi, un *send()*. Cette primitive est implémentée par un appel système ou par un appel de fonction dans une librairie. Dans ce dernier cas, la librairie effectue quelques traitements, et finit par faire un appel système. On en revient donc toujours à un appel système, c'est à dire à un passage en mode noyau. Ce cheminement est inévitable, car pour des raisons de sécurité et de partage des ressources, seul le noyau peut avoir accès au matériel.

Au cours de l'appel système, le processeur va recopier les données à transmettre depuis l'espace mémoire du processus utilisateur vers l'espace mémoire du noyau. Une fois en mode noyau, c'est le pilote de la carte réseau qui travaille. Il va généralement recopier les données à transmettre dans une mémoire d'entrée/sortie situé sur la carte. Une fois les données copiées dans cette mémoire d'entrée/sortie, c'est au matériel de s'occuper de leur acheminement sur le réseau.

Les données arrivent alors sur la machine réceptrice, dans la mémoire d'entrée sortie de la carte

réseau. Quand un paquet de donnée est reçu en entier, la carte va lever une interruption pour signaler au processeur de la machine destination que des données sont arrivées. Cette interruption fait passer le processeur de la machine réceptrice en mode noyau, dans le pilote de la carte réseau. Les données vont alors être recopiées de la mémoire d'entrée/sortie de la carte vers un tampon dans l'espace mémoire du noyau.

Après la réception des données, différents traitements sont possibles. Si les données s'avèrent être à destination d'un processus ayant requis des communication asynchrones, il faut par exemple lui notifier l'arrivée des données par un signal (typiquement un *SIGIO*). Si le processus destinataire a déjà exécuté une primitive de réception, un *receive()*, il avait été endormi par un *sleep()* en attendant l'arrivée des données et il faut le réveiller par un *wakeup()*. Enfin si le processus destinataire n'a rien demandé, il faut conserver les données dans l'espace mémoire noyau en attendant qu'il veuille bien les demander par un appel à une primitive de réception de données.

Une fois que le processus destinataire est prêt à recevoir les données, le processeur doit finallement copier l'ensemble des données pour les ramener de l'espace mémoire noyau vers l'espace mémoire utilisateur.

Durant cette communication toute simple, les processeurs des machines source et destination ont chacun recopié 2 fois les mêmes données. Ceci représente un temps considérable si de grosses quantités de données sont échangées. Et en tout état de cause, ceci représente du temps perdu pour le calcul.

Le circuit routeur RCube est capable de débiter 1Gb/s. C'est un débit assez considérable, mais si ce circuit est exploité avec les mécanismes classiques décrits ci dessus, on comprend que les performances ne seront pas au rendez-vous, car les processeurs seront essentiellement occupés à copier des données.

L'objectif dans la machine MPC est donc d'implémenter des mécanismes de communication présentant la propriété suivante : les données à transférer ne doivent jamais être copiées par le processeur. On parle de mode zéro copie.

Pour arriver à cet objectif, on va faire appel à la DMA (*Direct Memory Access*). La DMA est un mécanisme permettant à un périphérique d'avoir un accès direct en lecture ou en écriture à la mémoire centrale de l'ordinateur, sans passer par le processeur.

On a donc adjoint à RCube un second composant matériel, nommé PCIDDC, implémentant une primitive d'écriture dans la mémoire centrale d'un noeud de calcul distant (protocole DDSLRP : *Direct Deposit StateLess Receiver Protocol*). La DMA est utilisée au départ et à l'arrivée pour éviter les recopies mémoires par le processeur. PCIDDC réalise l'intermédiaire entre le bus PCI et RCube. Les couches logicielles n'ont qu'à indiquer via le bus PCI à PICDDC l'adresse source et la longueur du bloc à transférer, ainsi que l'adresse destination où il doit être écrit sur le noeud distant.

Plus d'informations sur PCIDDC : http://mpc.lip6.fr/pciddc/spec_pciddc

Un tel mécanisme permet de tirer partie des possibilités offertes par un routeur haute perform-

ance tel que RCube, tout en conservant un maximum de temps pour le calcul sur les noeuds.

III.2 Aspects matériels

Les circuits RCube et PCIDDC sont montés sur une carte PCI, appelée carte FastHSL, qui prend place dans un slot d'extension du PC. Suivant sa révision, la carte FastHSL comprend 7 connecteurs Harting ou 8 connecteurs Lemo, sur lesquels on peut brancher des câbles bi-coaxiaux. Du fait du nombre de connecteurs disponibles, il est possible de réaliser des réseaux fortement connexes pour des machines parallèles de 7 noeuds de calcul.

Pour les cartes à 8 connecteurs, seuls 7 connecteurs sont utilisables lors de l'utilisation dans un PC. En effet, le routeur RCube possède 8 interfaces, dont une est utilisée pour la communication avec PCIDDC. Le huitième connecteur permet l'utilisation de RCube comme noeud flottant, c'est à dire non raccordé à un PC. Cette possibilité n'a pas été actuellement exploitée.

Le circuit PCIDDC original (PCIDDC first run) comptait un certain nombre d'erreurs. Certaines d'entre elles ont été corrigées dans PCIDDC second run. Parmi les cartes à connecteurs Harting, certaines sont en PCIDDC first run, et d'autres en PCIDDC second run. Toutes les cartes à connecteur Lemo utilisent des circuits PCIDDC second run. Pour s'y retrouver, on parle de révisions de cartes FastHSL :

Rev. A PCIDDC first run, 7 connecteur Harting
Rev. B PCIDDC second run, 7 connecteurs Harting
Rev. C PCIDDC second run, 8 connecteurs Lemo

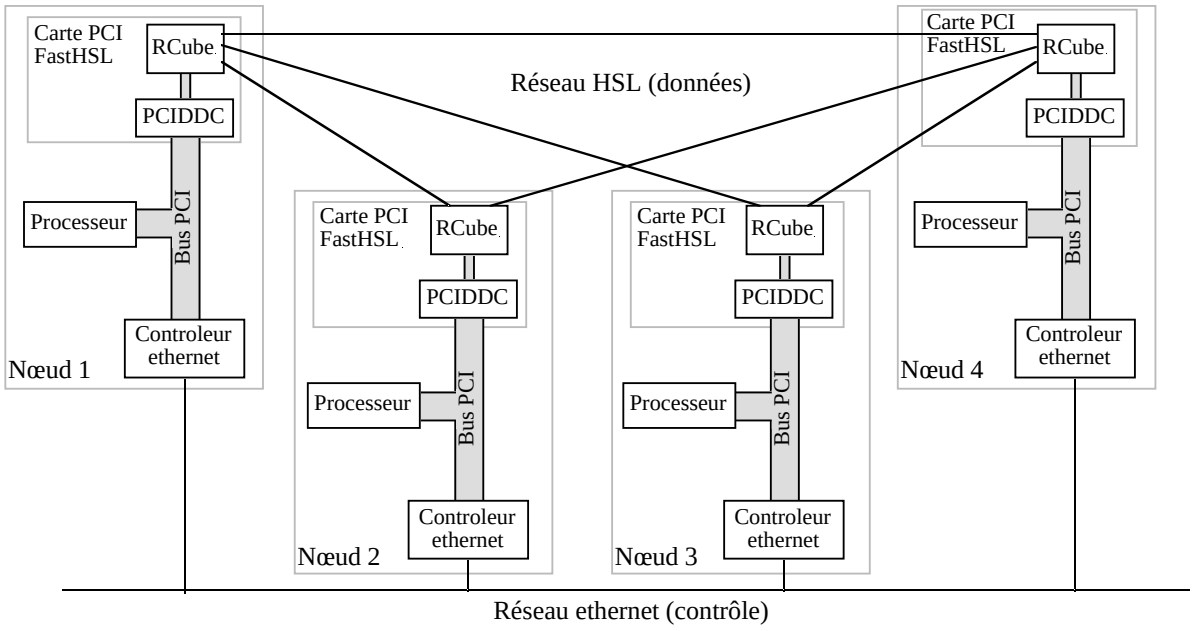

Fig. 2 : Machine MPC à 4 nœuds. On a choisi ici d'utiliser un réseau HSL fortement connexe.

Chaque nœud de calcul de la machine MPC est équipé d'une carte FastHSL. Les cartes FastHSL sont liées entre elles par le réseau de données, composé de liens HSL. Par ailleurs, pour les opérations de configuration, la machine MPC requiert un réseau de contrôle. Celui ci peut être n'importe quel réseau capable de transporter le protocole TCP/IP. Actuellement, c'est un réseau ethernet qui est employé. La Fig. 2 donne un aperçu général d'une machine MPC à 4 nœuds.

III.3 Exploitation logicielle des cartes FastHSL

L'exploitation logicielle des cartes FastHSL est assurée par MPC-OS (pour *MPC Operating System*). MPC-OS est implémenté sous forme de modules à charger dynamiquement dans les noyaux des systèmes FreeBSD et Linux :

• Le module CMEM, sur lequel nous reviendrons plus tard.

• Le module HSL, chargé de l'exploitation proprement dite de la carte, implémente un certain nombre de services pour les couches logicielles de plus haut niveau. Il aussi implémente l'interface de programmation PUT, qui permet l'écriture dans un nœud distant en utilisant la carte fastHSL.

Le module HSL doit donc dialoguer avec PCIDDC pour envoyer des pages de mémoire sur les noeuds distants. Le reste de cette partie aborde les moyens employés par le module HSL pour communiquer avec PCIDDC.

La norme PCI impose aux cartes PCI de fournir des registres de configuration. Ces registres sont accessibles en lecture et en écriture, et ils permettent des fonction d'auto détection et d'identification de la carte, ainsi que sa configuration lors du démarrage de la machine.

Habituellement, les registres de configuration ne sont utilisés qu'au démarrage de la machine pour configurer la carte. Lors de ces opérations, la mémoire d'entrée/sortie de la carte est remappée dans l'espace de mémoire virtuelle du noyau. Dans cet espace se trouvent des registres de contrôle et de statut qui servent à contrôler la carte lors du fonctionnement courant.

La carte FastHSL présente une particularité dans son usage des registres de configuration : il n'y a pas de mémoire d'entée/sortie sur la carte, et les registre de statut et de commande de PCIDDC sont accessible dans l'espace de configuration de la carte, et ceci durant le mode d'opération normal.

Au démarrage de la machine, le système d'exploitation remappe tous les espaces de configuration des cartes PCI dans la mémoire virtuelle du noyau. Il suffira donc d'aller lire et écrire à la bonne adresse pour communiquer avec PCIDDC. De même, des opérations sur le bus PCI lui même seront nécessaires, et pour cela on agira de la même façon sur les registres d'état et de contrôle du bus PCI (ils sont eux aussi remappés en mémoire virtuelle noyau lors du démarrage du système d'exploitation).

III.4 Vocabulaire

La spécification de PCIDDC utilise un certain nombre de termes qu'il convient de définir précisément avant d'aborder le fonctionnement global de la machine MPC.

Message : Il s'agit d'un bloc de mémoire dont on a demandé le transfert.

Page : Habituellement, la page est l'unité minimale de gestion de la mémoire. Sur la plupart des processeurs 32 bits (tous?), une page fait 4096 octets. Du point de vue de l'unité de gestion de la mémoire (MMU : *Memory Management Unit*) du processeur, la mémoire virtuelle se découpe en pages, dont

certaines sont mappées sur de la mémoire physique et d'autres non.
Numéro de nœud : Chaque routeur RCube sur le réseau HSL possède un numéro de nœud. Les numéros de nœuds sont statiques et assignés à l'initialisation du réseau par MPC-OS.
Identifiant de message (MI : *Message identifier*) : Chaque message sur le réseau possède un identifiant unique. Pour assurer l'unicité des MI, chaque processus utilisateur sur chaque nœud se voit attribuer un intervalle de MI par MPC-OS.
Message court (SM : *Short Messages*) : Pour transmettre des messages de moins de 8 octets, PCIDDC propose un mécanisme de messages courts, qui évitent un certain nombre d'étapes dans le transfert.

III.5 Fonctionnement global de la machine MPC

PCIDDC maintient dans la mémoire centrale de l'ordinateur deux tables essentielles au fonctionnement du protocole DDSLRP :
• la liste des pages à émettre (LPE : *List of Page to Emit*)
• la liste des pages reçues (LMI : *List of Message Identifiers*)

L'emplacement et la taille de ces deux tables en mémoire centrale est indiquée à PCIDDC par un accès en configuration. PCIDDC accède ensuite aux tables par DMA.

La table LPE contient des entrées de type `lpe_entry_t`, définies ainsi :

```
typedef struct _lpe_entry{
     u_short   page_length;    /* Longueur du bloc à transférer */
     u_short   routing_part;   /* Numéro du nœud destination */
     u_long    control;        /* Options diverses */
     caddr_t   PRSA;/* Adresse destination (sur nœud cible) */
     caddr_t   PLSA;/* Adresse source (sur nœud local) */
} lpe_entry_t;
```

Pour envoyer une page de mémoire sur un noeud distant, il suffit d'écrire l'entrée de la table LPE correspondante, et signaler à PCIDDC la présence de cette entrée par un accès en configuration. L'entrée de LPE contient l'adresse locale du bloc à transférer (`PLSA`), sa longueur (`page_length`) le Numéro de nœud sur lequel il faut le transférer (`routing_part`), et l'adresse à laquelle le bloc doit être transféré sur le nœud distant (`PRSA`).

Le champ `control` permet de déterminer certains comportements optionnels, comme par exemple l'envoi d'un message court. Dans ce cas, les champs `PRSA` et `PLSA` contiennent les données à transférer, au lieu d'indiquer leurs adresses source et destination.

La table liste des pages reçues (LMI), est mise à jour par PCIDDC dès qu'une page de a été écrite dans la mémoire. Ses entrées sont de type `lmi_entry_t`, définies ainsi :

```
typedef struct _lmi_entry{
     u_short   packet_number; /* Identifiant du message (MI) */
     u_short   r3_statut;     /* registre R3 */
     u_short   reserved;      /* Reserved for future use */
     u_short   control;       /* Options diverses */
     u_long    data1;         /* Adresse source */
     u_long    data2;         /* Adresse destination */
```

```
} lmi_entry_t;
```

Le champ `packet_number` contient le MI du message.

Le champ `r3_statut` contient la valeur du registre R3 de PCIDDC, qui indique l'état des 8 liens de RCube.

Le champ `control` corresponds au champ `control` de l'entrée de LPE associé au message.

Enfin, les champs `data1` et `data2` correspondent aux adresses sources et destination du bloc transféré, comme les champs `PRSA` et `PLSA` de l'entrée de LPE. Dans le cas des messages courts, ces champs contiennent les données qu'on a mis dans les champs `PRSA` et `PLSA` de l'entrée de LPE.

PCIDDC transfère toutes les données par DMA, donc de façon totalement asynchrone vis-à-vis du processeur. Il est donc nécessaire de pouvoir détecter la complétion des différentes tâches effectuées par PCIDDC. Ceci peut se faire de deux façons :

• par scrutation de la LPE et de la LMI

• en demandant à PCIDDC de générer une interruption lorsque la page correspondant à une entrée de LPE a été envoyée ou quand une entrée de LMI correspondant à une page arrivée, est ajoutée.

Chacune de ces deux méthodes présente son avantage : par scrutation, la latence d'un envoi est réduite, car elle ne comprend pas les interruptions. Par contre, le processeur doit être sollicité pour scruter périodiquement la LMI et la LPE. Ce temps de scrutation est ainsi perdu pour les calculs. La

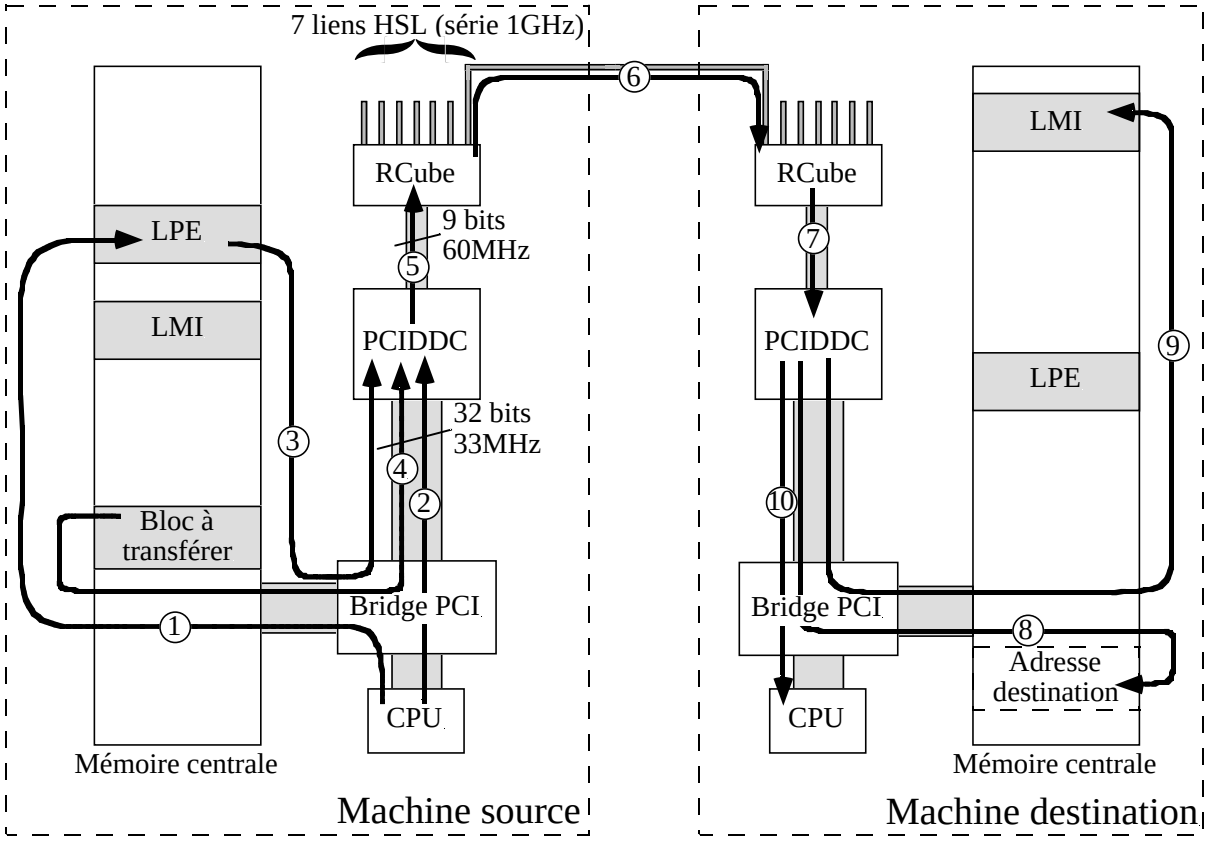

Fig. 3 : Différentes étapes mises en jeu lors du Transfert d'un bloc de mémoire par la machine MPC.

méthode par interruption permet d'éviter de perdre du temps à la scrutation, mais la latence des échanges en est allongée.

Les deux modes sont disponibles, mais pas simultanément. On choisit l'une ou l'autre en compilant MPC-OS avec l'option `PUT_MODEL_INTERRUPT_DRIVEN` (pour les interruptions) ou `PUT_MODEL_POLLING` (pour la scrutation).

La Fig. 3 donne une vue d'ensemble des différentes étapes nécessaires dans la transmission d'un bloc de mémoire :
• Etape 1 : écriture de l'entrée de LPE correspondante au bloc que l'on souhaite transférer.
• Etape 2 : accès en configuration à PCIDDC pour lui signaler la présence d'une nouvelle entrée à traiter.
• Etape 3 : PCIDDC lit l'entrée de LPE par DMA.
• Etape 4 : PCIDDC lit le bloc de mémoire à transférer par DMA.
• Etape 5 : transfert des données à acheminer de PCIDDC à RCube.
• Etape 6 : transfert des données d'un routeur RCube à l'autre, à travers le réseau HSL. Il serait possible durant cette étape de traverser des routeurs RCube intermédiaires.
• Etape 7 : transfert des données de RCube à PCIDDC, sur le nœud destination.
• Etape 8 : transfert des données de PCIDDC vers la mémoire centrale sur le nœud destination. Une fois de plus, la DMA est utilisée pour éviter les recopies mémoires

Les étapes 4 à 8 se déroulent en réalité en même temps : elles forment un pipeline.

• Etape 9 (optionnelle) : mise à jour de la LMI par PCIDDC. Cette mise à jour est faite par une opération de DMA, au cours de laquelle l'entrée correspondant au bloc arrivé est ajoutée à la LMI. Le champ `control` de l'entrée de LPE permet d'activer ou non la mise à jour de la LMI.
• Etape 10 (optionnelle) : levée d'une interruption pour signaler à la machine destination qu'un bloc de mémoire est arrivé. Le champ `control` de l'entrée de LPE permet d'activer ou non cette levée d'interruption.

IV Présentation de MPC-OS/Linux

Dans la partie III, certains composants de MPC-OS ont été vu de façon assez rapide. Cette partie décrit plus en détail les différents éléments constituants MPC-OS, ainsi que leurs interactions.

IV.1 Structure de la distribution MPC-OS/Linux

La distribution de MPC-OS pour Linux se présente sous la forme d'un fichier `.tar.gz`, qui une fois décompacté donne l'arborescence suivante (seuls les répertoires sont indiqués, dans un soucis de clarté) :

`mpc-linux`	Racine de la distribution MPC-OS/Linux
`mpc-linux/modules`	Modules CMEM et HSL
`mpc-linux/doc`	Documentation
`mpc-linux/routing`	Configurateurs de tables de routage HSL
`mpc-linux/src`	Utilitaires divers
`mpc-linux/src/emu`	Daemons de contrôle MPC (hslclient, hslserver)
`mpc-linux/src/emu/conf`	Fichiers de configuration (table de routage...)
`mpc-linux/src/testput`	Utilitaires de tests (testputbench...)
`mpc-linux/src/toys`	Recopie d'écran via MPC
`mpc-linux/mpcview`	Application MPCView pour voir l'état des liens HSL

Pour faire fonctionner la machine MPC, il faut charger les modules CMEM et HSL, et lancer les daemons de contrôle `hslclient` et `hslserver`. Il est alors possible d'exécuter des processus utilisant PUT. La Fig. 4 donne un aperçu de l'agencement des différents composants logiciels de MPC-OS.

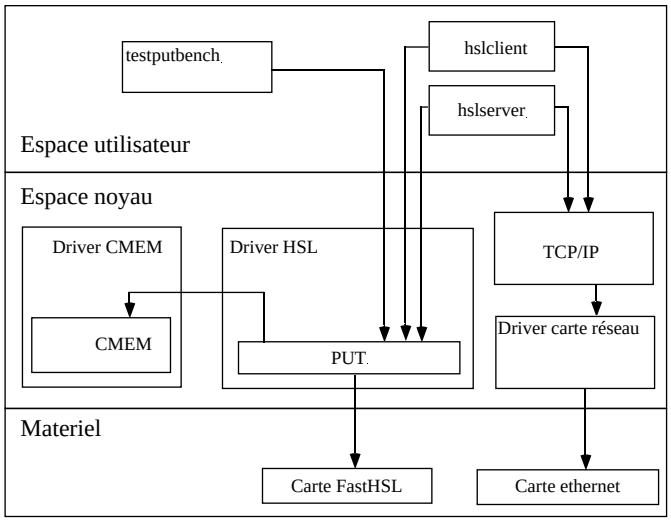

Fig. 4 : Agencement des différents éléments logiciels de la machine MPC. Dans cet exemple, un processus utilise la machine, il s'agit de `testputbench`.

La suite détaille les rôles de ces différents éléments logiciels.

IV.2 Le module CMEM

Le module CMEM est là pour repondre à un besoin d'allocation de mémoire physiquement continue dans la machine MPC. Voyons comment ce besoin apparait :

PCIDDC voit la mémoire depuis le bus PCI. Contrairement au processeur, il n'a pas de vision translatée de la mémoire par la MMU : le processeur voit la mémoire virtuelle, et PCIDDC voit la mémoire physique. On ne peut donc demander à PCIDDC de transférer que des blocs de mémoire physique : il n'a pas connaissance de la translation d'adresses virtuelles vers physique.

Lorsque l'on souhaite transférer une zone de mémoire quelconque d'une machine à une autre, on doit donc examiner chaque page de mémoire virtuelle source et destination, déterminer leurs adresses physiques, et demander à PCIDDC de faire le transfert des pages de mémoire physique. Cette opération de translation page par page peut se révéler coûteuse en temps de calcul, et c'est pourquoi on préfère utiliser des blocs de mémoire physiquement contigus.

Si on manipule un bloc de mémoire physiquement continue, une seule translation d'adresse est nécessaire pour un envoi de bloc de mémoire : il suffit de déterminer l'adresse physique de la première page du bloc. Il y a là un gain très appréciable pour des blocs de taille importante, et on préfère donc travailler avec des blocs de mémoire contigus.

Le système d'exploitation fournit une fonction pour allouer des blocs de mémoire continus, mais elle n'est pas destinée à l'allocation massive de gros blocs. Lors du démarrage, on dispose d'une quantité de mémoire contiguë libre importante, mais à mesure que le système fonctionne, des blocs de mémoire sont alloués et désalloués, et la mémoire libre physiquement contiguë se fait rare. Ceci est illustré dans la Fig. 5 :

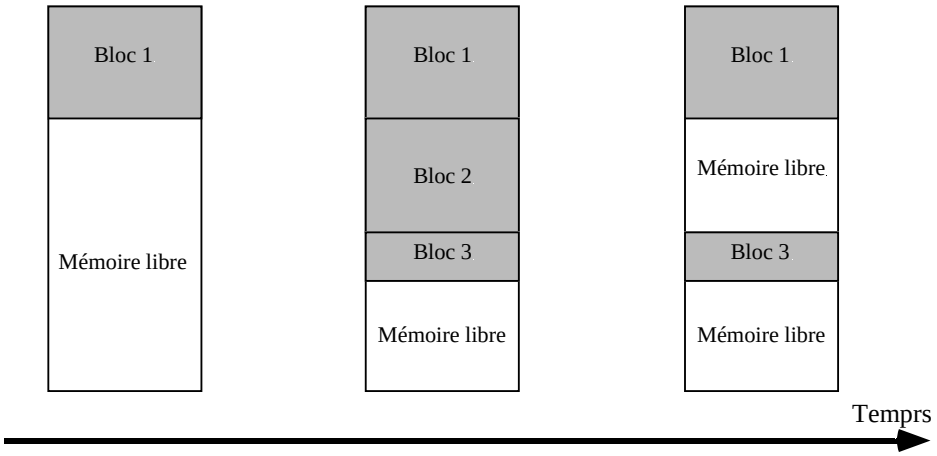

Fig. 5 : Exemple d'évolution de la mémoire physique au gré des allocations et désallocations. Au début, la mémoire contiguë est omniprésente, et à mesure que la mémoire est utilisée, elle disparaît. Dans l'état de droite, on a encore beaucoup de mémoire libre, mais du fait du bloc 3, la mémoire libre physiquement contiguë est très reduite.

Ce problème de pénurie de mémoire physiquement contiguë ne se pose pas pour les allocation de mémoire classiques, car le système de mémoire virtuelle sait allouer des blocs de mémoire virtuelle fragmentés en plusieurs blocs de mémoire physique continue, en les faisant apparaître à l'utilisateur comme virtuellement contigus. Mais pour MPC, nous devons faire de la DMA, donc nous avons besoin

de blocs physiquement contigus.

Le module CMEM est un gestionnaire de blocs contigus (on parle de mémoire CMEM, ou de bloc CMEM). Son rôle est de réserver un gros bloc de mémoire physiquement continue lors de son lancement, et ensuite de fournir un service d'allocation de blocs aux applications utilisant PUT. CMEM implémente 4 fonctions :

- `cmem_getmem()` permet à un utilisateur d'allouer un bloc CMEM.
- `cmem_releasemem()` libère un bloc précédemment alloué.
- `cmem_virtaddr()` donne l'adresse virtuelle associée à une adresse physique.
- `cmem_phys_addr()` donne l'adresse physique associée à une adresse virtuelle.

IV.3 Le module HSL

Comme on l'a vu, le module HSL a la charge de l'exploitation des cartes FastHSL. Cette exploitation est assurée par la couche PUT. PUT implémente un certain nombre de fonctions, qui se repartissent en quatre catégories : configuration, signalisation, gestion des interruptions, et émission. Voici les fonctions essentielles, voir les pages man (en annexe B) pour plus de détails sur leur rôles.

Configuration

- `put_init()` Initialisation du PUT (au chargement du module HSL)
- `put_end()` Terminaison de PUT (au déchargement du module HSL)
- `put_register_SAP()` Enregistrement comme utilisateur de PUT
- `put_unregister_SAP()` Fin d'utilisation de PUT.

Signalisation

- `put_get_node()` Renvoie le numéro de nœud HSL.
- `put_get_lpe_high()` Renvoie un pointeur sur l'entrée de LPE courante
- `put_get_lpe_free()` Renvoie le nombre d'entrées libres dans la LPE
- `put_attach_mi_range()` Attribue une plage de MI à l'utilisateur
- `put_get_mi_start()` Renvoie le premier MI libre (dans la plage attribuée)

Emission

- `put_add_entry()` Ajoute une entrée dans la LPE

Gestion des interruptions

- `put_interrupt_handler()` Gestionnaire d'interruption

Du point de vue des sources, le module HSL se découpe en 4 fichiers :

- `hsldriver.c` implémente le driver en mode caractère,
- `put.c` implémente la couche PUT,
- `pci_ioctl.c` contient les `ioctl()` d'accès au bus PCI
- `put_ioctl.c` regroupe les `ioctl()` de PUT.

IV.4 Les daemons de contrôle

Comme la Fig. 4 l'indique, la machine MPC utilise deux daemons de contrôle : `hslclient` et `hslserver`. Leur rôle est d'échanger les information de configuration de la machine via le réseau de contrôle lors de l'initialisation de PUT. Si la machine MPC fonctionne en émulation (c'est à dire en utilisant le réseau de contrôle pour envoyer les messages devant normalement passer par le réseau HSL), ce sont ces deux daemons qui assurent le transport des données sur le réseau de contrôle.

V Portage de PUT sous Linux

MPC-OS était initialement disponible sous FreeBSD. Le portage sous Linux n'a pas été une opération triviale. Cette partie explique les spécificités de la version Linux, et les différents problèmes de portage rencontrés.

V.1 Problèmes spécifiques au module CMEM

CMEM essaye de réserver une quantité de mémoire assez importante. Sous FreeBSD, une fois que le système a fini de démarrer, il est encore possible de charger le module CMEM, et d'obtenir l'allocation de blocs de mémoire physiquement contiguë assez importants. Il ne faut toutefois pas trop tarder : si la machine est trop utilisée avant le chargement de CMEM et qu'elle ne dispose pas d'énormément de mémoire, le chargement de CMEM échouera. Par exemple, sur une machine dotée de 32Mo de mémoire, si on compile MPC-OS avant de charger CMEM, il n'y aura plus assez de mémoire physiquement contiguë de libre, et le chargement du module échouera. Il faut alors redémarrer la machine pour pouvoir charger CMEM.

Sur certaines configurations, la mémoire physiquement contiguë disparaît extrêmement rapidement, pendant le démarrage. Il est alors nécessaire de charger CMEM au tout début du processus de démarrage, c'est à dire au début du script `/etc/rc`, alors que le système d'exploitation est encore en mode mono-utilisateur.

Sous Linux, les choses se passent différemment. la gestion de la mémoire est telle que même en tentant le chargement de CMEM tout au début du mode mono-utilisateur, il est impossible d'obtenir de gros blocs de mémoire contiguë. Il est donc nécessaire de réserver de la mémoire continue pour CMEM. Ceci se fait en spécifiant au noyau Linux de ne pas utiliser toute la mémoire. Il reste ainsi en haut de l'espace d'adressage physique une zone continue non gérée par le noyau. Il est possible de réserver ainsi une zone d'une taille arbitraire, pourvu qu'on laisse assez de mémoire au système d'exploitation pour fonctionner.

La réservation de la mémoire se fait en passant au noyau l'option `mem` lors du démarrage. Ceci peut être fait soit de façon interactive à chaque démarrage, soit automatiquement en configurant de façon adéquate Lilo, le programme de démarrage du noyau Linux (Lilo est une contraction de *LInux LOader*).

Par exemple, si la machine dispose de 64Mo de mémoire et que l'on désire réserver 14Mo pour CMEM, on va démarrer avec l'option `mem=50M`. Si on désire configurer Lilo pour cela, on va modifier le fichier `/etc/lilo.conf` ainsi :

```
# /etc/lilo.conf original
boot=/dev/hda
map=/boot/map
install=/boot/boot.b
prompt
timeout=30
image=/boot/vmlinuz-2.2.5-15
      label=linux
      root=/dev/hda1
      read-only
```

```
# /etc/lilo.conf modifié
boot=/dev/hda
map=/boot/map
install=/boot/boot.b
prompt
timeout=30
image=/boot/vmlinuz-2.2.5-15
      label=linux
      append = "mem=50M"
      root=/dev/hda1
      read-only
```

...et valider la modification en invoquant le programme `/sbin/lilo`. Pour plus de détails concernant le fonctionnement de lilo, voir les pages man `lilo(8)` et `lilo.conf(5)`.

Une fois ce bloc de mémoire réservé, on peut charger le module CMEM à n'importe quel moment : la mémoire réservée pour CMEM étant tout simplement ignorée par le noyau, ne peut pas être fragmentée.

Nous verrons dans la partie consacrée à l'implémentation de services de plus haut niveau basés sur PUT, les techniques utilisées pour accéder à de la mémoire CMEM depuis un processus utilisateur. En bref, sous FreeBSD, il s'agit de faire l'appel système `mmap()` sur le fichier spécial `/dev/kmem` qui donne accès à la mémoire virtuelle du noyau. Ceci permet de remapper en mémoire utilisateur le bloc CMEM qui est jusque là géré dans l'espace noyau. Le bloc est ensuite accessible normalement dans la mémoire virtuelle du processus utilisateur.

L'implémentation de CMEM étant assez particulière sous Linux, cette technique n'est pas utilisable : comme le noyau ne gère pas la mémoire CMEM dans son espace de mémoire virtuel, il n'est pas accessible par le fichier `/dev/kmem`. Il a donc été nécessaire de créer une fonction `mmap()` pour le fichier spécial `/dev/cmem` donnant accès au module CMEM.

Cette fonction ayant été mise en place, un processus utilisateur désirant remapper un bloc CMEM dans son espace virtuel doit utiliser `mmap()` sur `/dev/cmem` sous Linux, au lieu de `/dev/kmem` sous FreeBSD. Les applications FreeBSD fonctionnant en mode utilisateur et utilisant PUT doivent donc être modifiées à chaque usage de `open()` puis `mmap()` sur `/dev/kmem`, de la façon suivante :

```
/* programme original
(FreeBSD) */

fd=open("/dev/kmem",O_RDONLY);
if (fd<0) {
  perror("open /dev/kmem fai-
led");
  exit -1;
}
vptr=mmap( /* ... */
```

```
/* programme modifié pour Linux */

#ifdef __FreeBSD__
fd=open("/dev/kmem",O_RDONLY);
#endif /* __FreeBSD__ */

#ifdef linux
fd=open("/dev/cmem",O_RDONLY);
#endif /* linux*/

if (fd<0) {
  perror("open /dev/kmem failed");
  exit -1;
}
vptr=mmap( /* ... */
```

Cette modification est la seule à apporter au code source d'un processus utilisant PUT sous FreeBSD pour qu'il fonctionne sous Linux. Tout le reste de l'interface de programmation de PUT en espace utilisateur est parfaitement identique sous FreeBSD et sous Linux. ependant, comme on le verra dans la partie suivante, la situation est malheureusement assez différente en ce qui concerne le code source développé pour tourner dans le noyau.

V.2 Particularités diverses du noyau Linux

Cette partie examine les difficultés liées au portage d'une partie du module HSL sous Linux. Comme on l'a vu, le module HSL implémente l'exploitation de la carte FastHSL, l'interface de programmation PUT, plus un certain nombre d'autres services (SLR/P, SLR/V...). Dans la version Linux, seul l'exploitation de la carte FastHSL et l'interface de programmation PUT ont été portés.

Comme on l'a expliqué plus haut, le portage d'applications fonctionnant en mode utilisateur se fait sans peine de FreeBSD à Linux. Dans le cas d'applications pour MPC, seules les opérations sur `/dev/cmem` au lieu de `/dev/kmem` changent lorsque l'on passe de FreeBSD à Linux. Cette différence est due au fait que l'implémentation de CMEM sous Linux ne suit pas exactement la même interface de programmation que sous FreeBSD. On aurait fait le remappage des blocs CMEM sur le même fichier spécial sous Linux et sous FreeBSD, les programmes en mode utilisateur auraient été parfaitement identiques.

Cette situation confortable existe grâce aux normes POSIX : il s'agit d'un ensemble de spécifications développées par l'IEEE pour assurer que tous les systèmes dérivants du système UNIX de AT&T possèdent un dénominateur commun en terme d'interface utilisateur et d'interface de programmation utilisateur (API : *Application Programming Interface*). Il existe un certain nombre de normes POSIX. Voici la description des plus importantes :

• POSIX.1 définit les API. Par exemple les sémantiques des appels systèmes tels que `open()`, ou des appels de librairie tels que `malloc()`. La spécification indique également des choses telles que le fichier d'en tête où une fonction doit être déclarée. La norme POSIX.1 assure qu'un program-

me écrit en C sera portable d'un système dérivé d'UNIX à un autre.

• POSIX.2 définit l'interface utilisateur, c'est à dire les comportements minimaux du shell et des utilitaires. POSIX.2 assure qu'un shell-script sera portable d'un système à un autre.

Les normes POSIX ont été incluses dans celles du consortium open group, qui a racheté la marque déposée UNIX. Ceci a donné lieu à la norme *Single UNIX Spécification*, aussi appelée UNIX95, et plus récemment à la norme UNIX98. L'open group possédant la marque UNIX, il a la faculté d'autoriser une compagnie à dire que son système d'exploitation est un système UNIX. Cette autorisation s'obtient en passant un programme de certification onéreux. Ceci explique que la plupart des UNIX commerciaux (Solaris, HP-UX, etc...) l'aient, et que les systèmes UNIX libres (FreeBSD, NetBSD, OpenBSD, Linux) ne l'aient pas.

S'il est vrai que les systèmes UNIX libres n'ont pas le droit de se réclamer du nom UNIX, il reste qu'ils respectent les normes. On obtient donc une portabilité des programmes en C fonctionnant en mode utilisateur.

Plus d'informations sur l'open group : http://www.opengroup.org

Hélas, les couches PUT sont implémentées sous forme de module à charger dans le noyau. Ceci est indispensable, car il faut être en mode noyau pour pouvoir accéder au matériel (en l'occurrence aux cartes FastHSL). Et si en mode utilisateur, les normes POSIX sont là pour assurer une compatibilité facile, en mode noyau aucune norme n'a jamais été écrite pour assurer une homogénéité entre systèmes d'exploitation.

Les implémentations des noyaux sont donc totalement libres d'un système UNIX à un autre. Depuis le noyau, une action aussi élémentaire qu'afficher un message d'urgence sur la console de la machine se fait de façon différente sous Linux ou sous FreeBSD. Entre FreeBSd et Linux, l'inexistence de standard est en plus aggravée par le fait que ces deux systèmes n'ont aucune racine commune : FreeBSD dérive de 4.4BSD, la branche d'UNIX développée à l'Université de Californie à Berkley (UCB), elle même provenant de l'UNIX original de AT&T (1BSD dérive de UNIX time sharing system, sixth edition, en 1978). Linux, de son coté a été développé de 1991 à 1994, de toutes pièces, sans utilisation de la moindre ligne de code provenant d'AT&T.

Ceci nous amène à des différences assez importante dans la façon dont les choses sont faites en mode noyau. Dans l'esprit, les fonctions sont les mêmes car il s'agit dans les deux cas d'un système "à la UNIX", avec un noyau monolithique, mais les noms des fonctions, les arguments, les codes de retours, et les endroits où elles sont déclarées sont différents.

Tout le code source du module HSL n'utilisant pas de service fourni par le noyau peut donc rester inchangé. Par contre, dès qu'une fonction située dans le noyau est appelée, il faut faire une modification pour obtenir la version Linux.

Plutôt que de modifier la version FreeBSD pour obtenir une version Linux, il a été chois d'introduire un niveau d'abstraction, pour pouvoir utiliser exactement le même code source pour Linux et FreeBSD : tous les appels de fonction du noyau, les types utilisés et les variables globales du noyau ont été remplacées par des macros. Ces macros sont définies dans le fichier `osdep.h` (voir annexe A).

En fonction du système, FreeBSD ou Linux, elles sont définies de façon adéquate.

Voici des extraits du fichier osdep.h, pour illustrer le propos :

```
#ifdef linux
#define pcireg_t u_char
#define WAKEUP wake_up
#define PCI_STATUS_PARITY_DETECT (PCI_STATUS_DETECTED_PARITY << 16)
#endif /* linux */

#ifdef __FreeBSD__
#define pcireg_t u_long
#define WAKEUP wakeup
#endif /* __FreeBSD__ */
```

Dans ces exemples, on voit par exemple l'introduction du type `pcireg_t`. Une fonction disponible en mode noyau permettant d'accéder au bus PCI utilisait un argument de type `u_long` sous FreeBSD. Sous Linux, la fonction équivalente utilisait un `u_char`. Pour régler le problème, l'argument est déclaré de type `pcireg_t`, et suivant le système d'exploitation, `pcireg_t` est un `u_char` ou un `u_long`. Autre exemple : la fonction `wakeup` de FreeBSD s'appelle `wake_up` sous Linux. Pour unifier le code, on a introduit la macro `WAKEUP`.

On constate aussi que la partie Linux contient plus de définitions que la partie FreeBSD. Ceci est dû au fait qu'il a toujours été tenté de rester le plus proche possible du code source de FreeBSD, car c'est la version FreeBSD qui est en cours de développement actuellement. Par exemple, FreeBSD utilise la macro `PCI_STATUS_PARITY_DETECT` pour désigner un registre de statut d'un périphérique PCI. Cette macro a été laissée telle quelle dans le code source, et Linux ne la connaissant pas, on lui définit de façon adéquate dans `osdep.h`.

Cette dernière définition de macro est d'ailleurs intéressante pour illustrer à quel point Linux et FreeBSD arrivent à avoir des visions différentes de choses pourtant rigoureusement identiques : les registres de statut et de contrôle du bus PCI sont les mêmes sous Linux et sous FreeBSD. La fonction des différents bits dans ces registres est une donnée matérielle, définie par la norme PCI. La Fig. 6 donne les fonctions des premiers champs de l'en tête PCI, tel que la norme les définit :

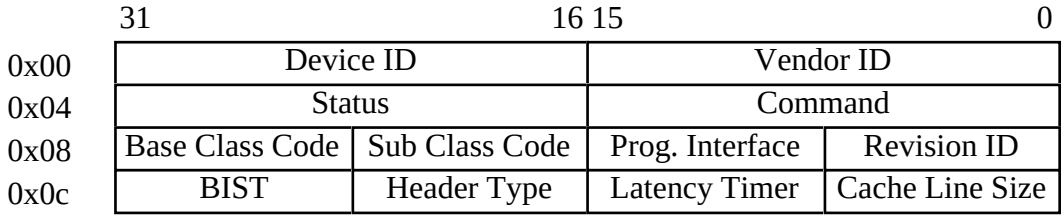

Fig 6 : Début de l'en tête PCI d'après la norme

Voici comment Linux définit les champs Command et Status dans le fichier `<linux/pci.h>`. Par soucis de clarté, seules quelques définitions de bits des registres `PCI_COMMAND` et `PCI_STATUS` ont été conservés.

```
#define PCI_VENDOR_ID            0x00     /* 16 bits */
#define PCI_DEVICE_ID            0x02     /* 16 bits */
#define PCI_COMMAND              0x04     /* 16 bits */
#define  PCI_COMMAND_IO          0x1      /* Enable resp. in I/O space */
/* [...] */
#define  PCI_COMMAND_SERR        0x100    /* Enable SERR */
#define  PCI_COMMAND_FAST_BACK   0x200    /* Enable back-to-back writes */

#define PCI_STATUS               0x06     /* 16 bits */
/* [...] */
#define  PCI_STATUS_SIG_SYSTEM_ERROR 0x4000 /* Set when we drive SERR */
#define  PCI_STATUS_DETECTED_PARITY 0x8000 /* Set on parity error */
```

Et voici comment FreeBSD définit les mêmes bits des mêmes registres :

```
#define PCI_COMMAND_STATUS_REG           0x04
#define PCI_COMMAND_IO_ENABLE            0x00000001
/* [...] */
#define PCI_COMMAND_SERR_ENABLE          0x00000100
#define PCI_COMMAND_BACKTOBACK_ENABLE    0x00000200
/* [...] */
#define PCI_STATUS_spécial_ERROR         0x40000000
#define PCI_STATUS_PARITY_DETECT         0x80000000
```

Les données décrites sont exactement, les mêmes, mais FreeBSD voit un registre unique `PCI_COMMAND_STATUS_REG` de 32 bits, et repère les bits à l'intérieur de ce registre, alors que Linux voit deux registres de 16 bits, `PCI_COMMAND` et `PCI_STATUS`, et repère les bits à l'intérieur de ces deux registres de 16 bits. Non seulement les bits des registres de commandes et de statut n'ont pas les mêmes noms, mais en plus, leurs position ne sont pas déterminées de la même façon.

V.3 Chargement des modules

FreeBSD et Linux fournissent tous les deux une fonctionnalité intéressante : la possibilité de charger dynamiquement des modules dans le noyau. Ceci permet d'ajouter des fonctionnalités au noyau sans avoir à le redémarrer, c'est à dire sans avoir à interrompre les services fonctionnant sur la machine. De plus, cela permet au développeur travaillant sur des drivers (qui sont donc forcement implémentés en mode noyau, puisqu'ils doivent accéder au matériel) de tester ses modifications sans redémarrer la machine à chaque fois.

Cette fonctionnalité de chargement de modules dynamiques n'est pas standardisée, ni du point de vue des commandes pour charger un noyau en espace utilisateur, ni du point de vue de la façon dont le module se compile et s'intègre dans le noyau. Ceci crée des zones de code totalement dépendantes du système d'exploitation dans le module HSL.

Pour compliquer les choses, FreeBSD propose deux interfaces de chargement de module :

• l'interface LKM, développée à l'origine par Sun Microsystems pour Solaris, et intégrée à

FreeBSD, NetBSD et OpenBSD. LKM a l'avantage de sa large adoption. On peut lui voir un certain aspect standard. Le projet IPFILTER propose d'ailleurs un module LKM multi-plateforme faisant du filtrage et de la translation d'adresses IP pour Solaris, FreeBSD, NetBSD et OpenBSD.

• L'interface KLD, qui essaye de combler des manques dans LKM, en particulier sur la façon dont les modules sont liés au noyau : il faut une table des symboles complète du noyau utilisé. KLD propose un chargement plus simple.

Etant donné que les modules pour FreeBSD sont déjà assez compliqués du fait du support de deux interfaces différentes à la fois, on a préféré ne pas se lancer dans une fusion de code avec les modules Linux, qui ont une interface encore différente. Les parties concernant le chargement du module sont donc isolées dans des fichiers qui n'ont pas été unifiés : chaque système d'exploitation a le sien.

Ces fichiers dépendants du système d'exploitation font néanmoins une chose très similaire : ils déclarent un driver en mode caractère. Dans les deux cas, il s'agit de déclarer au système d'exploitation une table d'opérations supportées par le driver : ouverture, fermeture, lecture, écriture, `ioctl()`, `mmap()`, etc... Ceci se fait en passant une table de pointeurs vers les méthodes correspondantes.

Abordons maintenant le chargement des modules pour Linux : une fois le module compilé, pour le charger, il suffit de le placer dans le répertoire `/lib/modules/2.2.5-15/MPC-OS` (le "`2.2.5-15`" correspond à la version du noyau en cours d'utilisation), et invoquer les commandes

`depmod -a`	pour reconstruire l'arbre de dépendances entre modules,
`modprobe cmem`	pour charger le module CMEM,
`modprobe hsl`	pour charger le module HSL.

La commande `depmod -a` n'est nécessaire qu'après la première installation des modules.

Le module étant chargé, il faut créer le fichier spécial correspondant dans le répertoire `/dev`. Le fichier spécial permet d'accéder au driver depuis le mode utilisateur. Un `open()` sur le fichier spécial invoque la méthode `open()` du driver, un `mmap()` invoque la méthode `mmap()`, et ainsi de suite.

Les fichiers spéciaux sont caractérisés par deux numéros : le numéro de mineur et le numéro de majeur. Ce dernier permet de lier un fichier spécial au driver correspondant. Chaque driver a son Numéro de majeur. Pour certains ce numéro est statique, pour d'autre il est dynamique et déterminé au chargement du driver. C'est le cas pour les modules HSL et CMEM sous Linux. Pour créer le fichier spécial, il faut donc aller chercher le numéro de majeur attribué au driver lors de son chargement.

Cette information est disponible par le pseudo-système de fichier `/proc`. `/proc` est une arborescence de fichiers qui n'existent pas réellement : ce sont des points d'accès au noyau. Des opérations de lecture et d'écriture sur ces fichiers permettent de lire et d'écrire directement certaines données du noyau. Parmi les données disponibles en lecture, on trouve la table des drivers en mode caractère et la table des drivers en mode bloc. Ces deux tables indiquent pour chaque driver le numéro de majeur et le nom. On les obtient en tapant la commande `cat /proc/devices`.

Le numéro de mineur permet à un driver de contrôler plusieurs périphériques. Le mineur correspond au numéro du périphérique que l'on souhaite atteindre. Par exemple, le driver fd gère les lecteurs de disquettes. Si on a plusieurs lecteurs de disquettes, on va créer un fichier `/dev/fd0` de mineur 0 pour accéder au premier, et un fichier `/dev/fd1` de mineur 1 pour accéder au second. Les

deux fichier spéciaux ont le même majeur, qui correspond au driver fd.

Les fichiers spéciaux se créent avec la commande `mknod`. On peut donc les faire avec la ligne de commande suivante :

```
mknod /dev/cmem c `awk '{if ($2=="cmem") {print $1}}' /proc/devices` 0
mknod /dev/hsl c `awk '{if ($2=="hsl") {print $1}}' /proc/devices` 0
```

Pour plus de détails sur la commande `mknod`, voir sa page man : `mknod(1)`. Point intéressant, la création du fichier spécial peut être automatisée au chargement du module, moyennant de configurer le fichier `/etc/conf.modules`. Voir la page man `modprobe(8)` pour plus de détails.

Enfin, la méthode de déchargement des modules :

`modprobe -r hsl` pour décharger HSL,
`modprobe -r cmem` pour décharger CMEM.

Il faut signaler qu'il subsiste une bogue dans le module HSL : une fois que la fonction `put_init_SAP` a été appelée, si le module est déchargé, il ne pourra pas être rechargé correctement sans redémarrer la machine. Il semblerait que ceci soit dû à un problème de ressource non libérée : l'IRQ utilisée par la carte FastHSL ne serait pas libérée correctement, ce qui fait qu'au chargement suivant, le driver HSL ne peut pas se l'attribuer. (le driver se charge alors en indiquant que la carte est à l'IRQ 0, et rien ne fonctionne correctement.)

VI Performances de PUT/Linux et PUT/FreeBSD

Une fois la version Linux de PUT réalisée, la question des performances comparées de PUT/Linux et PUT/FreeBSD se pose. Cette partie aborde les problèmes de mesures de performances, ainsi que les résultats obtenus.

VI.1 Conditions expérimentales

Tous les tests de performances ont été faits sur les mêmes machines : deux PC identiques, équipés de pentium MMX 166MHz, 32Mo de mémoire, et d'une carte mère HX. Les cartes FastHSL étaient en révision C, donc équipées d'un PCIDDC *second run*.

Le test consiste à essayer d'envoyer le plus de données possible sur le lien HSL : le rôle du programme de test consiste à ajouter des entrées de LPE dès qu'une entrée est libre. On mesure le temps pris pour un million de transferts pour des tailles de paquets allant de 1 octet à 65535 octets (qui se trouve être la taille maximum de paquet transférable par PCIDDC, car dans la LPE, le champ indiquant la taille de la page à transférer fait 2 octets).

Pour effectuer ces tests, le programme `testputsend_loop` a été modifié. A l'origine, ce programme ne servait qu'à tester le bon fonctionnement des cartes FastHSL. Il lui a été adjoint un mécanisme de mesure du temps et des options permettant de faire automatiquement des séries de mesures.

La mesure du temps n'est pas chose triviale : l'appel système `gettimeofday()` renvoie l'heure système à la microseconde près. Les microsecondes sont évaluées de façon très précise : elles sont calculées depuis un registre du processeur indiquant le nombre de cycles écoulés depuis la mise sous tension. Par contre, les secondes sont incrémentées par un mécanisme d'interruption : une interruption à lieu N fois par seconde et à chaque fois un compteur est incrémenté. Lorsque l'on arrive à N, l'heure système est incrémentée d'une seconde. Or PUT contient beaucoup de sections critiques, qui fonctionnent en niveau d'interruption le plus haut possible, inhibant les autres interruptions. Lorsque la machine MPC tourne sous une forte charge, des interruptions de remise à jour des secondes de l'heure système vont sauter, et l'horloge va prendre du retard. L'heure système indique donc l'heure réelle modulo une seconde, avec une précision d'une microseconde.

Pour obtenir une mesure fiable, le programme `timer.c` développé par Philippe Lalevée de l'INT a donc été utilisé. Ce programme lit les microsecondes directement dans le registre du processeur, et seul cette mesure du temps est utilisée. Moyennant d'avoir correctement étalonné la vitesse de son processeur, on arrive à des déviations minimes par rapport à l'heure réelle.

VI.2 L'option LPE_SLOW

Sur les cartes mères BX, le bridge 440BX présente des comportements étranges lors de raffales d'accès en configuration sur le bus PCI : certains accès en lecture échouent, en renvoyant 0xFFFF. Sachant qu'un bit du registre de status de PCIDDC est relié à la masse, il est impossible que PCIDDC ait renvoyé cette valeur. Les soupçons se portent donc sur le bridge. Ce comportement n'est pas visible en mode de fonctionnement normal d'un PC équipée d'une carte mère BX, car les cartes PCI n'utilisent les accès en configuration que lors de l'initialisation. Seule notre carte FastHSL en fait un usage

intensif.

Ce problème est extrêmement contraignant. Le seul moyen existant actuellement pour le contourner est de ne mettre qu'une entrée à la fois dans la LPE. Pour obtenir cela, il faut compiler MPC-OS avec l'option LPE_SLOW. Ansi, la machine MPC fonctionne de façon fiable (du moins vis à vis de ce problème), mais c'est au prix d'une diminution notable des performances, car le processeur doit régulièrement ajouter des entrées de LPE, au lieu de toutes les ajouter d'un coup.

Les machines ayant servi à faire les tests de performance étaient équipées d'une carte mère HX. L'option LPE_SLOW n'était donc pas indispensable. Toutefois, faire des tests avec et sans l'option LPE_SLOW s'est révélé assez intéressant, comme nous allons le voir plus loin.

VI.3 Résultats

La Fig. 7 donne les résultats des premiers tests de performances qui ont été faits. On déduit le débit de la mesure du temps pris par l'envoi d'un million de paquets.

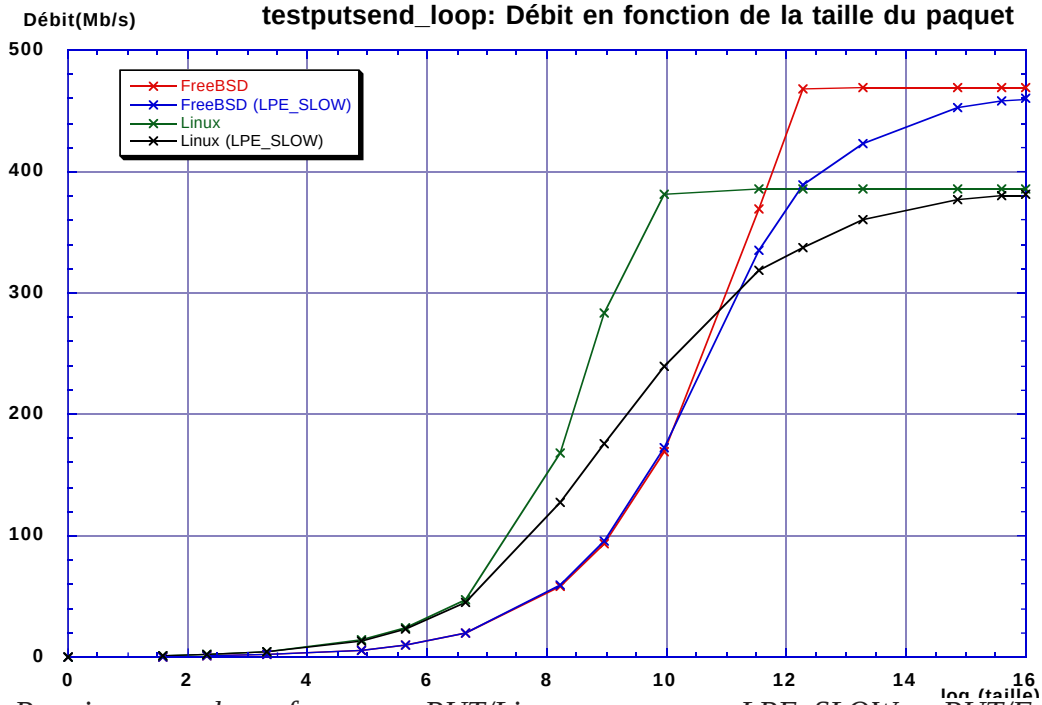

Fig. 7 : Premiers tests de performance, PUT/Linux avec et sans LPE_SLOW, et PUT/FreeBSD avec et sans LPE_SLOW.

Tout d'abord, intéressons nous aux courbes avec et sans LPE_SLOW. Autant pour FreeBSD que pour Linux, elles sont confondues pour des petits paquets, car PCIDDC envoie plus vite les entrées que le processeur ne les ajoute. On n'a donc jamais plus d'une entrée dans la LPE, qu'on ait positionné l'option LPE_SLOW ou non.

A partir d'une certaine taille de paquet, PCIDDC commence à ne plus pouvoir suivre le processeur, et les entrées vont commencer à s'accumuler dans la LPE si LPE_SLOW n'a pas été utilisé. On s'attend donc à voir un point de séparation entre les courbes avec et sans LPE_SLOW, qui corres-

pond au moment où la LPE commence à être pleine.

Ce point a un intérêt : si la LPE ne contient pas au moins un élément en permanence, il y aura des moments où PCIDDC va attendre que le processeur vienne ajouter une entrée, donc des moments où le réseau HSL ne fonctionne pas à 100% du temps. Passé le point de séparation des courbes, on sait que la courbe sans LPE_SLOW correspond à un fonctionnement continu de PCIDDC.

La courbe rouge (PUT/FreeBSD sans LPE_SLOW) sature à 470Mb/s, pour des paquets de 5ko. Ceci correspond à la saturation du lien entre PCIDDC et RCube, qui est un lien 9 bits à 60MHz, d'où un débit maximum de 540Mb/s. Les 70Mb/s de différence s'expliquent par le fait que pour chaque paquet, PCIDDC ajoute un en tête. Ce test donnant le débit utile maximum, il ne tient pas compte de l'en-tête.

La comparaison des courbes PUT/Linux et PUT/FreeBSD, fait apparaître deux choses inattendues :

• Linux est beaucoup plus rapide que FreeBSD pour des petits paquets. il arrive à remplir sa LPE pour une taille de 100 octets, alors que FreeBSD ne parvient à remplir la LPE qu'à 1ko (ces points correspondent aux points de séparation des courbes avec et sans LPE_SLOW, comme on l'a expliqué plus haut). La différence de débit entre Linux et FreeBSD est d'environ un facteur 2 sur toute la courbe avant que Linux atteigne saturation. Pour une taille de 500 octets, on a presque un facteur 3.

• Linux sature beaucoup plus tôt que FreeBSD, à 380Mb/s au lieu de 470Mb/s

La différence pour des petits paquets peut s'expliquer aisément : des appels systèmes plus rapides sous Linux que sous FreeBSD, ou des traitements faits avant ou après l'appel du gestionnaire d'interruption de PUT plus lourd sous FreeBSD que sous Linux. Par contre, la différence de 90Mb/s à la saturation est difficilement expliquable.

La première piste a été qu'un autre périphérique prenne de la bande passante sur le bus PCI, car Linux l'aurait configuré différemment que FreeBSD, d'une façon plus bruillante.. Quelques tests ont été faits dans cette voie, par exemple en travaillant sans carte vidéo, ils n'ont pas été concluants, mais nous ont appris quelque-chose : les performances sont encore pires si on enlève la carte vidéo (les transferts sont alors ralentis d'un facteur 6).

Une deuxième idée a été lancée : Linux sature complètement pour une taille de paquet de 1ko, or, 1ko correspond à 256 transactions sur le bus PCI (le bus faisant 32 bits de large, il transporte 4 octets par transaction), ce qui est le nombre maximum de transactions qu'une carte puisse faire sur le bus PCI. Coïncidence? Après une étude approfondie : oui, c'en est une. Mais cette idée a permit de mettre le doigt sur le vrai problème.

Dans les registres de configuration du bus PCI, on trouve le registre `PCI_LATENCY_TIMER`, qui permet de fixer le nombre de transactions maximum autorisées par périphérique PCI. Ce registre fait un octet, et peut donc prendre une valeur de 256 maximum, d'où la limite précédemment citée.

Lorsqu'un périphérique libère le bus, si aucun autre ne le prend, il faut quelques transactions avant qu'il ne le récupère. On comprend donc que plus le registre `PCI_LATENCY_TIMER` a une valeur élevée, meilleures seront les performances de PUT.

`PCI_LATENCY_TIMER` a une valeur par défaut de 32 fournie par le BIOS du PC (on peut la régler dans le setup). Cette valeur est récupérée par le système d'exploitation, qui peut éventuellement la modifier. C'est ce qui est fait dans MPC-OS : le module HSL pousse normalement la valeur de `PCI_LATENCY_TIMER` à 256, pour maximiser les performances.

Après examen des registres de configuration du bus PCI, il s'est avéré que sous Linux, `PCI_LATENCY_TIMER` n'était pas positionné à 256, mais gardait la valeur par défaut de 32. Ceci faisait que sous Linux, PCIDDC lâchait le bus au bout de 32 transactions, puis devait attendre un certain temps (de l'ordre de 5 transactions) avant de reprendre le bus. Sous FreeBSD, PCIDDC transférait pendant 256 transactions avant de devoir attendre et reprendre le bus. Cette différence suffit à expliquer la différence en débit qui était observée à saturation.

La bogue (car c'en etait une) sous Linux était due au fait que la macro indiquant la position du registre `PCI_LATENCY_TIMER` n'avait pas été correctement définie. Le module HSL écrivait donc la valeur 256 dans le registre voisin de `PCI_LATENCY_TIMER`, et `PCI_LATENCY_TIMER` restait égal à 32. Ce problème stupide étant réglé, les courbes de la Fig. 8 ont été obtenues.

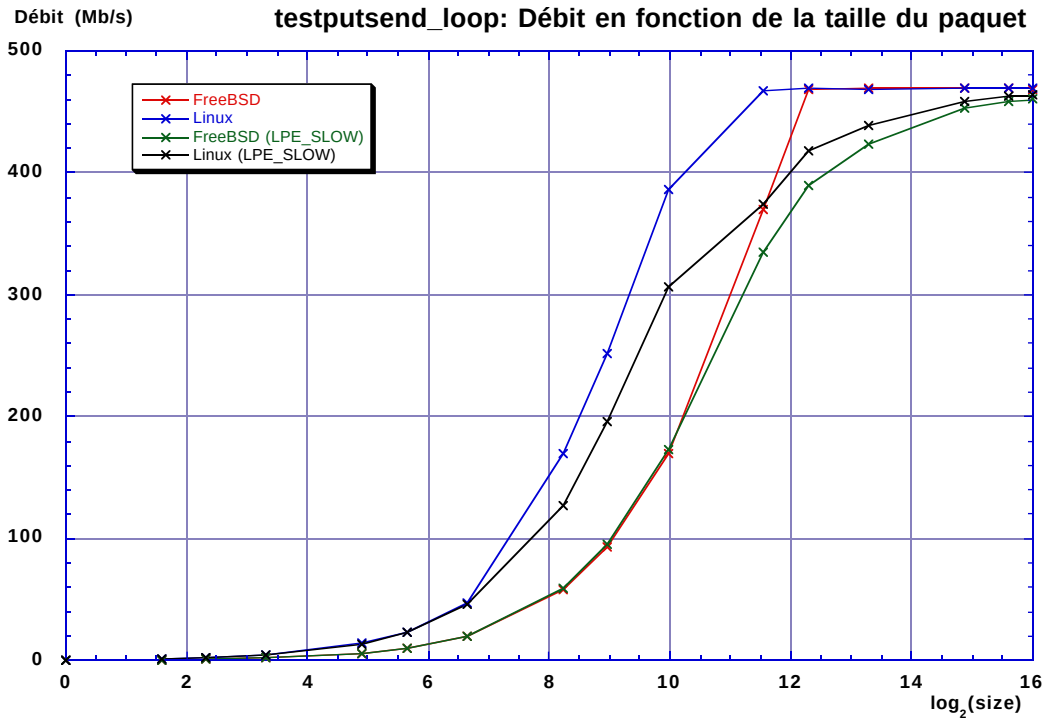

Fig 8 : Performances comparées de PUT/Linux et PUT/FreeBSD, versions finales

Sur ces nouvelles courbes, on obtient saturation à 470Mb/s pour Linux et pour FreeBSD. Le reste des courbes reste inchangé : la plus grande célérité de Linux sur les petits paquets se confirme.

On peut s'inquiéter de l'inflexion de la courbe Linux sans LPE_SLOW à 1ko. En effet, la limite actuelle de débit à 470Mb/s est due à la saturation du lien entre PCIDDC et RCube. Ce lien fait 9 bits de large et est actuellement cadencé à 60MHz. La fréquence du lien pouvant changer dans le futur, on espère pouvoir atteindre des débits nettement supérieurs. Le tassement de la courbe de Linux sans

LPE_SLOW est donc préoccupant.

En réalité, il s'agit juste d'un problème lié au manque de points sur la courbe. Si on trace avec plus de points, on obtient la Fig.9.

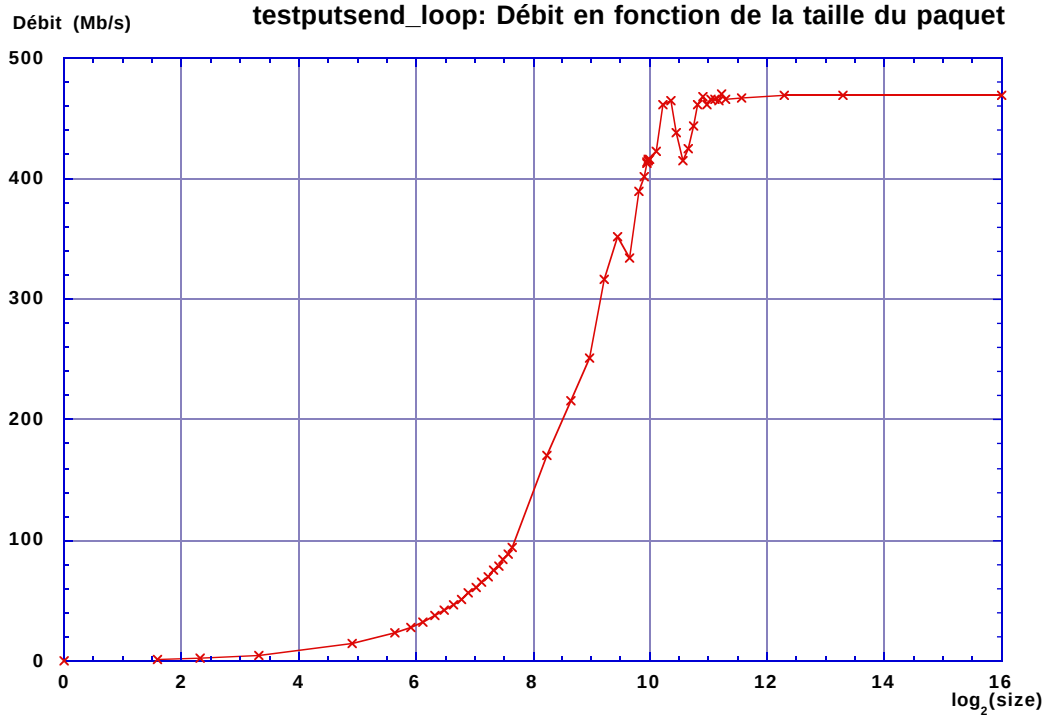

Fig. 9 : Performances de PUT/Linux sans LPE_SLOW, avec une meilleure résolution.

On y voit apparaître des chutes de débits pour certaines tailles de paquets. Ceci est dû au xmoments où PCIDDC doit découper un message en deux paquets. L'ajout des 27 octets d'en-tête du second paquet fait chuter le débit utile. Les découpages de paquets peuvent survenir dans 3 cas :

- On a atteint la taille maximum de paquet pour PCIDDC (540 octets)
- PCIDDC a du relâcher le bus (du à `PCI_LATENCY_TIMER`)
- Les FIFO de PCIDDC étaient presque pleines. PCIDDC commence un processus de vidange des FIFO et ceci provoque un découpage avec un nouveau paquet.

En conclusion sur cette partie, on peut dire que Linux offre un gain de performances significatif par rapport à FreeBSD pour du débit de petits paquets. Une fois la saturation atteinte, les deux systèmes se valent, mais il reste que si la cadence du bus entre PCIDDC et RCube est augmentée, la taille de paquets pour lesquels les deux systèmes seront équivalents sera repoussée plus haut.

VII Vers des services de plus haut niveau

Cette partie aborde le problème du développement d'une API de haut niveau au dessus de PUT, en examinant le cas de MPI. Mais avant d'aborder l'exemple de MPI, un rappel du contexte est nécessaire

VII.1 Mode utilisateur et mode noyau

Un système UNIX possède deux modes de fonctionnement : le mode noyau et le mode utilisateur. En mode utilisateur, le processeur est en mode non privilégié. Certaines opérations sont interdites, en particulier la modification des tables de la MMU et les vecteurs d'exceptions. Le processeur ne voit donc que la mémoire qu'on veut lui faire voir, à savoir l'espace de mémoire virtuelle utilisateur du processus. Il exécute ainsi le code du processus, qui agit sur les données du processus. A moins de remapper les zones d'entrées sorties dans l'espace de mémoire virtuelle du processus, il est impossible de réaliser des entrées sorties en mode utilisateur.

Lorsque le système d'exploitation fonctionne en mode noyau, le processeur tourne en mode privilégié. Dans ce mode, toutes les opérations sont permises. On y voit l'ensemble de la mémoire, y compris les zones d'entrées sorties, et la zone de mémoire du noyau. Lorsqu'il est en mode privilégié, le processeur exécute le code du noyau, et il peut réaliser des opérations d'entrées/sorties.

La transition du mode utilisateur au mode noyau peut se faire par exception logicielle ou par exception matérielle :

• Par exception matérielle : le noyau met en place une interruption à intervalle de temps régulier pour pouvoir récupérer le contrôle du processeur. Ceci lui permet de faire des opérations de scrutation diverses, et aussi d'assurer la commutation d'un processus utilisateur à l'autre. Des interruptions en provenance des périphériques peuvent aussi redonner la main au noyau.

• Par exception logicielle : il peut s'agir d'une exception levée suite à la lecture d'une page non résidente en mémoire centrale, d'une erreur d'adresse, d'une division par zéro, ou tout simplement par un appel système.

Un appel système se fait en levant une exception logicielle : le processeur passe alors en mode privilégié, et exécute la fonction adéquate dans le noyau. Comme il a été vu, PUT tourne en mode noyau, car il doit manipuler du matériel. Si l'on désire rendre des fonctions de PUT accessibles à un processus utilisateur, il faut gérer le passage du mode utilisateur au mode noyau

On pourrait ajouter des appels systèmes pour rendre les fonctions de PUT accessible depuis l'espace utilisateur, mais il existe une méthode plus simple (et surtout plus standard, étant donné que PUT est implémenté dans un driver en mode caractères) : l'appel système `ioctl()`.

Les drivers en mode caractère et en mode bloc fonctionnent en mode noyau, et ils implémentent un ensemble de méthodes : `open()`, `read()`, `write()`, `close()`, `ioctl()`,et `mmap()`, qui sont rendues accessibles en mode utilisateur par l'intermédiaire d'un fichier spécial (généralement placé dans `/dev`). Les appels systèmes sur le fichier spécial invoquent les fonctions correspondantes du driver dans le noyau.

Ces drivers en mode caractère et bloc ont pour rôle de gérer un périphérique, sur lequel on doit

lire et écrire des données. En général, ceci se fait avec `open()`, `read()`, `write()`, et `close()` : on lit sur un port série comme on lit dans un fichier. Il existe toutefois des opérations spéciales à effectuer sur le périphérique, qui ne peuvent pas se faire par une lecture ou une écriture : éjecter une disquette, changer la vitesse d'un port série...

Pour ces opérations spéciales, on utilise l'appel système `ioctl()`, qui s'applique sur un fichier spécial. On lui passe un identifiant d'IOCTL (une lettre et un nombre), ainsi qu'un pointeur sur les données à transmettre. L'appel système `ioctl()` passe le contrôle à la méthode `ioctl()` du driver en mode noyau, et celui ci identifie l'opération à effectuer, traite les arguments et les renvoie modifiés au processus ayant fait l'appel `ioctl()` (ceci est vrai pour un IOCTL en lecture/écriture. On peut aussi en faire en lecture seule, en écriture seule, et aussi sans aucun argument).

`ioctl()` permet donc d'invoquer des opérations spécifiques à un driver de façon simple. On l'utilise donc dans le driver HSL pour rendre accessible depuis le mode utilisateur des opérations de PUT et de CMEM telles qu'ajouter une entrée de LPE ou allouer un bloc de mémoire contiguë.

On pourrait programmer PUT depuis le mode utilisateur par des `ioctl()` sur le driver HSL. L'usage veut plutôt que l'on fournisse une librairie contenant des fonctions appelant les `ioctl()` adéquats. Ceci permet de modifier l'interface avec le noyau sans briser le bon fonctionnement des applications : il suffit de modifier la librairie.

Ce travail a été fait pour PUT : la librairie `libuseraccess.a` donne accès aux fonctions de PUT en mode utilisateur. Des programmes tels que `testputbench` ou `testputsend_loop` l'utilisent pour avoir accès à PUT en mode utilisateur. D'autres programmes, comme `hslclient` ou `hslserver` (ce sont les daemons de configuration de la machine MPC, qui ont pour rôle l'émulation à travers le réseau de contrôle et le chargement des tables de routage) utilisent les `ioctl()` directement, sans passer par `libuseraccess.a`.

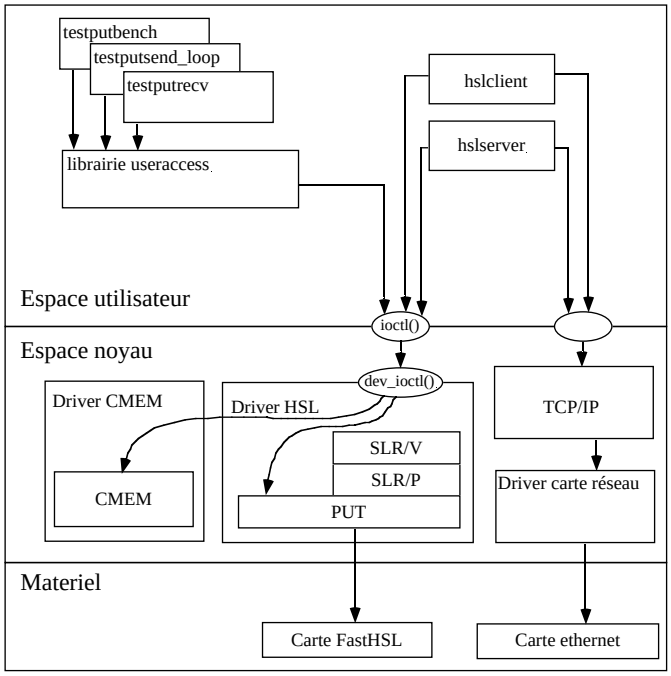

Fig. 10 : Architecture logicielle de la machine MPC

La Fig 10 résume l'architecture logicielle de la machine MPC, en distinguant bien ce qui fonctionne en mode noyau et en mode utilisateur.

La Fig. 11 retrace le cheminement typique d'appels de fonctions, correspondant au moment ou un programme écrit pour utiliser PUT récupère son numéro de nœud :

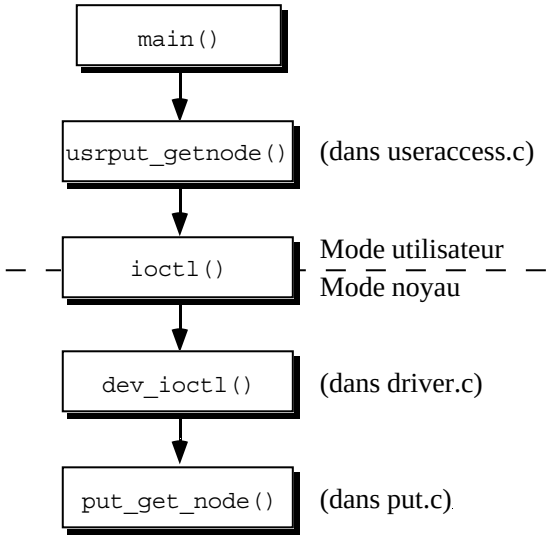

Fig. 11 : Appels de fonctions typiques d'une application utilisant PUT via la librairie useraccess.

VII.2 L'exemple de MPI

La machine MPC permet d'obtenir de bonnes performances sur les communications point à point, sans charge pour le processeur; elle est donc une bonne candidate pour faire du calcul parallèle. Mais pour pouvoir faire tourner une application parallèle dessus sans avoir à la porter, il faut offrir des interfaces de programmation (API : *Application Programming Interface*) parallèles standards. PVM sur MPC est déjà disponible, et la machine MPC à 4 nœuds du LIP6 est même ouverte au monde entier pour faire tourner des applications PVM. Voir http://mpc.lip6.fr/PVM.html pour plus d'informations.

Une autre API parallèle sur MPC est actuellement en cours de développement par Olivier Glück, au LIP6, il s'agit de MPI sur MPC. Ce projet se base sur MPICH, une implémentation de MPI prévue pour être facilement portable sur des nouveaux moyens de communication.

MPICH utilise deux sortes de messages : les messages de contrôle (pour les paquets de petite taille), et les messages de données (pour des paquets plus importants). L'envoi de messages de données est précédé d'un échange de messages de contrôle, pour indiquer où doit être fait le transfert. L'implémentation des messages de contrôle est aujourd'hui faite, et le problème de l'implémentation des messages de données se pose.

Lorsque des programmes tels que `testputbench` ou `testputsend_loop` utilisent PUT,

ils n'utilisent que des blocs de mémoire CMEM, car PCIDDC n'est capable de transférer que des zones de mémoire physiquement continues. Si l'on désire faire fonctionner une application MPI quelconque, on se heurte au cas où l'application transfère des données alloués par un `malloc()` standard, allouées statiquement (variables globales du programme), ou bien alloués dynamiquement sur la pile (variables locales d'une fonction). Dans les trois cas, aucune garantie n'existe sur le fait que les données seront contiguës en mémoire.

De plus, un autre problème existe : PUT n'est pas fait pour faire des passages de messages : Si une machine expédie des données par un `send()` alors que la machine destination ne les a pas encore demandé par un `recieve()`, où les données doivent elles être placées? D'une manière générale, le choix de l'adresse de destination est problématique. Ce second problème est assez complexe et pour le moment non résolu.

Pour le problème de la continuité des blocs de mémoires à émettre, plusieurs solutions ont été imaginées :

1) Utiliser les couches SLR/V du driver HSL, comme cela a été fait pour PVM. Les couches SLR/V sont capables de prendre une zone de mémoire virtuelle, de la découper en zones physiquement continues, et de demander à PCIDDC d'envoyer chaque zone contiguë.
• Avantage : C'est facile, une partie du travail délicat est déjà fait
• Inconvénients : D'une part, les performances seront faibles, les couches SLR/V introduisant beaucoup de latence (essentiellement pour fournir un système de cannaux virtuels, qui ne nous interesse pas ici). D'autre part, SLR/P et SLR/V n'ont pas été portés sous Linux, donc cette implémentation de MPI ne fonctionnera pas sous Linux

2) Allouer au processus entier de la mémoire contiguë, en modifiant le chargeur de processus du noyau.
• Avantage : Là encore, moins de travail délicat, il suffit de demander à PCIDDC de transférer, toute zone étant contiguë.
• Inconvénient : On perd l'allocation dynamique du tas d'un processus : si le processus remplit tout son espace de mémoire, on ne peut pas lui donner plus de mémoire sans physiquement fragmenter son tas.

3) Créer un MPI_Malloc_MPC(), permettant à un processus MPI d'allouer un bloc physiquement continu.
• Avantage : c'est simple, et là encore il suffira de demander un transfert de bloc contigu à PCIDDC.
• Inconvénient : cela ne respecte pas la sémantique de MPI. Il est prévu que les allocations mémoires se fassent avec `malloc()`, pas avec un appel de librairie non standard. Ce premier problème peut se résoudre en remplaçant le `malloc()` standard par notre `MPI_Malloc_MPC()`. Il reste un second problème : les variables globales et locales ne sont pas assurées d'être dans des blocs contigus. Cette méthode ne suffit pas pour pouvoir transférer n'importe quelle donnée.

4) Reprendre dans SLR/V le code qui nous intéresse, pour envoyer une page de mémoire virtuelle par morceaux contigus.
• Avantage : cette méthode est universelle, c'est à dire qu'elle permettra de transmettre n'importe quoi, y compris les données allouées sur la pile.
• Inconvénient : On doit faire des choses complexes qui se programment de façon différente sous

FreeBSD et Linux. (transaltion d'adresses, bloquage des données en mémoire, gestion d'un cache pour ne pas avoir à refaire ces opérations coûteuses)

Ce sont les deux dernières solutions qui ont été retenues : fournir un `MPI_Malloc_MPC()`, qui remplacera le `malloc()` de la librairie C standard, et par ailleurs une méthode d'envoi "à la SLR/V" pour toutes les données qui n'ont pas été allouées par `MPI_Malloc_MPC()`.

Dans la méthode "à la SLR/V", pour envoyer un bloc de mémoire virtuelle, nous devons assurer les tâches suivantes : d'abord faire un mlock() sur le bloc, pour être sur qu'il reste en mémoire centrale, sans changer d'adresse physique. Ensuite, il faut déterminer les adresses physiques et les longueurs des fragments de mémoire physiquement continues composant le bloc de mémoire virtuelle. La Fig. 12 illustre ce problème de fragmentation des blocs de mémoire virtuelle en mémoire physique.

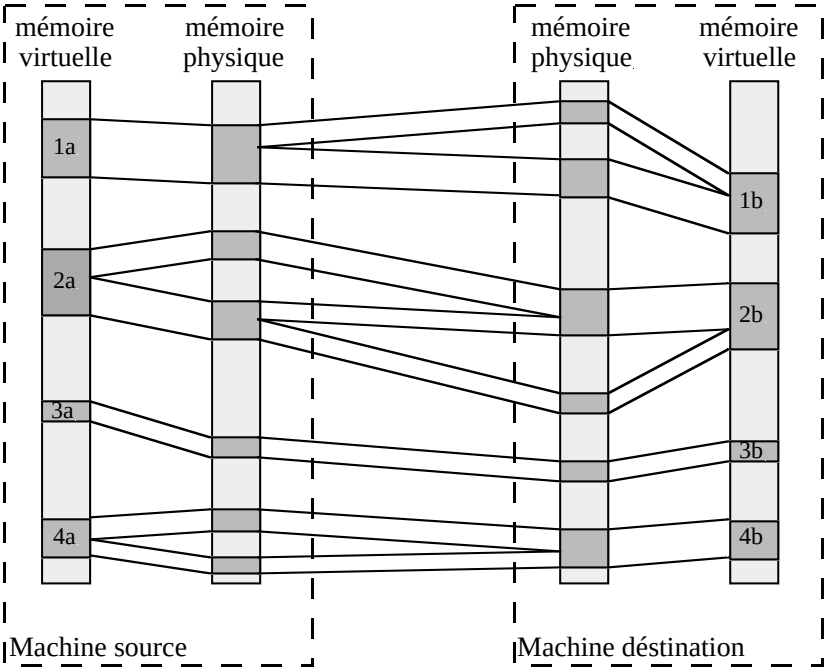

Fig. 12 : Différents types de transferts de blocs de mémoire virtuelle dans la machine MPC. Quatres cas sont possibles. Avec le bloc 1a, on a un bloc continu en mémoire physique à transférer dans un bloc non continu 1b. Dans le cas du bloc 2a, un bloc non continu à transférer dans un bloc non continu. Le bloc 3a est continu et se transfert dans un bloc continu 3b. Enfin, le bloc non continu 4a se transfert dans un bloc continu 4b.

Dans les couches SLR/V, après expédition du message, on fait un `munlock()` sur le bloc, et à chaque re-expédition du même bloc, les mêmes opérations sont faites : `mlock()`, découpage du bloc de mémoire virtuelle en fragments physiquement continus et envoi. Cette approche est économe en mémoire bloquée par `mlock()`, mais elle introduit une forte latence, car les traitements à faire avant l'envoi de chaque bloc de mémoire virtuelle sont assez lourds.

On a donc choisi de ne pas faire de `munlock()`, et de conserver toutes les données concernant le bloc de mémoire d'un envoi à l'autre. Pour cela, on utilise une table, nommée `mpi_vtophys_table`, qui contient des entrées de type `mpi_contigmem_t`, définies ainsi :

```
typedef struct _mpi_contigmem {
  int       next_page;              /* next page in vtophys_table */
  size_t    size;                   /* required / obtained size */
  caddr_t   vptr;                   /* processus virtual address */
  caddr_t   kptr;                   /* kernel virtual address */
  caddr_t   pptr;                   /* physical address */
  char      name[CMEM_NAMELEN];     /* name for the slot */
  int       slot;                   /* slot number */
} mpi_contigmem_t;
```

Chaque entrée décrit un bloc de mémoire physiquement continue. Les champs `vptr`, `kptr`, `pptr` contiennent respectivement les adresses virtuelles utilisateur, virtuelle noyau, et physique du bloc. Le champ `size` indique la longueur du bloc, et le champ `name` permet de donner au bloc un identifiant plus compréhensible pour l'être humain que ses adresses.

Le champ next_page sert à chaîner des fragments physiquement continus de bloc de mémoire virtuelle. Chaque fragment a dans son champ next_page l'index dans la table de l'entrée décrivant le fragment suivant. Le dernier fragment a par convention un champ next_page égal à -1. Les entrées de table libre sont repérées par un champ next_page égal à l'index de l'entrée.

Pour envoyer un bloc de mémoire virtuelle, on va donc parcourir la table à la recherche d'une entrée ayant un champ `vptr` correspondant au bloc à envoyer. On va ensuite demander l'envoi de chaque fragment de mémoire physiquement continue du bloc. Si le premier fragment n'a pas été trouvé dans la table, le bloc n'a encore jamais été envoyé, et il faut donc faire le `mlock()`, parcourir les tables de MMU, et ajouter les entrées correspondant au bloc dans la table.

Par ailleurs, on voudrait que les blocs alloués par `MPI_Malloc_MPC()` soient envoyés de façon identique. On va donc ajouter les blocs CMEM alloués par `MPI_Malloc_MPC()` à la table. Le champ `slot` sert alors à stocker le Numéro de slot CMEM, et il est par convention égal à -1 pour les entrées ne correspondant pas à des blocs CMEM. Ainsi, les blocs CMEM pourront être traités de la même façon que les blocs traités par la méthode "à la SLR/V", simplement, ce sont des blocs de mémoire virtuelle n'ayant qu'un fragment de mémoire physique.

Actuellement, MPI sur MPC se restreint à un processus MPI par nœud. Le but de cette restriction est de commencer le développement sur des bases d'une complexité raisonnable. Ainsi, les structures utilisées par MPI sur MPC sont allouées dans l'espace virtuel utilisateur du processus lors de son lancement, et libérées lors de sa terminaison.

On s'est posé la question de savoir si la table `mpi_vtophys_table` devait être aussi placée dans l'espace utilisateur, ou si l'on devait la placer dans l'espace noyau. On a finalement décidé de la placer dans l'espace noyau, pour la raison suivante : si on doit envoyer un bloc de mémoire virtuelle comportant N fragments, et si la table n'est pas accessible en mode noyau, on doit parcourir la table en mode utilisateur, et faire N appels systèmes pour envoyer chaque fragment. Par contre, si la table est disponible dans l'espace noyau, il suffit d'un seul appel système : le parcours de la table et les envois sont fait en mode noyau.

Comme il est nettement plus facile de rendre accessible en mode utilisateur une structure présente dans l'espace du noyau que le contraire, on a donc choisi de placer la table en mode noyau. Ceci présente de plus l'avantage de faire un pas dans la direction du MPI muli-tâche sur MPC, puisque la table peut être partagée entre plusieurs processus. Il reste toutefois un certain nombre de lacunes empêchant ce système de fonctionner en multi-tâche. En particulier, lorsqu'un processus MPI se termine, on doit libérer la mémoire CMEM, la mémoire bloquée par `mlock()`, et les entrées de tables utilisées par le processus. Actuellement, on traite ainsi toutes les entrées de table. Pour un fonctionnement multi-tâche, il faudrait ajouter un champ `pid` dans la structure `mpi_contigmem`. Ce champ contiendrait le PID du processus propriétaire de la mémoire, et à la libération des ressources, on saurait quelles entrées appartiennent à quel processus, et donc quelles entrées il faut libérer.

VII.3 Implémentation

Intéressons-nous maintenant à l'implémentation proprement dite. MPI sur MPC est divisé en deux composants. D'une part, MPICH doté des couches basses adaptées à MPC. Cette partie se compile en la librairie `libmpich.a`, avec laquelle on doit lier les programmes MPI. D'autre part, le driver MPI, chargé d'effectuer toutes les tâches qui doivent être faites en mode noyau.

Le driver MPI a donc la charge de la table `mpi_vtophys_table`, de l'allocation des blocs CMEM pour `MPI_Malloc_MPC()`, et des messages de contrôle. La méthode "à la SLR/V" n'est pas encore implémentée. Il fournit un certain nombre de fonction accessibles uniquement en mode noyau. Voici celles qui concernent la table et l'allocation de blocs CMEM :

`int mpi_vtophys_table_init(void)`
Allocation et initialisation de la table `mpi_vtophys_table` (utilisé au chargement du driver MPI)

`int mpi_vtophys_table_add_entry(mpi_contigmem_t,int)`
Ajout d'une entrée dans la table. Cette fonction prend en argument une entrée de table, et l'index du fragment précédent. Cet index est utilisé pour chaîner facilement les fragments appartenant à un même bloc de mémoire virtuel : `mpi_vtophys_table_add_entry()` renvoie l'index auquel à été ajoutée l'entrée. Lorsque l'on ajoute une série de fragment, il suffit de donner comme index de fragment précédent la valeur renvoyée par `mpi_vtophys_table_add_entry()` lors de l'appel précédent. On passe un index de -1 pour le premier fragment d'un bloc (ou pour un bloc non fragmenté)

`int mpi_vtophys_table_rm_entry(caddr_t vptr)`
Suppression d'une entrée de la table, retrouvée par son adresse virtuelle utilisateur. Dans le cadre d'une version multi-tâche, il faudra aussi donner le PID, car deux blocs distincts dans deux processus différents peuvent avoir la même adresse virtuelle utilisateur.

`caddr_t mpi_get_vtophys_table(void)`
Renvoie l'adresse virtuelle noyau de la table.

`void mpi_release_vtophys_table(void)`
Détruit la table (utilisé au déchargement du driver MPI).

`void mpi_vtophys_table_cleanup(void)`
Libère toutes les ressources occupées dans la table : libération des blocs CMEM, déverrouillage des

blocs bloqués par `mlock()`, et libération des entrées de table. Dans une version multi-tâche, il faudrait passer ici le PID correspondant au processus dont on veut libérer les ressources.

Pour rendre ces fonctions accessibles en mode utilisateur, le driver implémente les `ioctl()` suivants :

`#define MPI_GETUSERMEM _IOWR('O', 33, struct _mpi_contigmem)`
Alloue un bloc de mémoire CMEM, en appelant en mode noyau la fonction de CMEM `cmem_getmem()`, et ajoute ce bloc dans la table `mpi_vtophys_table` en appelant `mpi_vtophys_table_add_entry()`.

`#define MPI_RELEASEUSERMEM _IOW('O', 35, struct _mpi_contigmem)`
Libère un bloc de mémoire CMEM, et supprime l'entrée correspondante dans la table `mpi_vtophys_table`.

`#define MPI_FREEUSERSLOT _IO('O', 36)`
Appelle `mpi_vtophys_table_cleanup()` pour libérer les ressources. Dans une version multi-tâche, elle devrait prendre le PID du processus.

`#define MPI_GETVTOPHYSTABLE _IOR('O', 34, caddr_t)`
Renvoie l'adresse virtuelle noyau de la table `mpi_vtophys_table`.

Ces `ioctl()` sont ensuite utilisés dans la librairie `libmpich.a`, dans les fonctions suivantes:

`mpi_contigmem_t* mpi_map_vtophys_table(void)`
Mappe la table mpi_vtophys_table dans l'espace de mémoire virtuelle du processus. Cette fonction est utilisée à l'initialisation du processus. Elle utilise l'IOCTL `MPI_GETVTOPHYSTABLE`, et fait un `mmap()` pour mapper la table. Cette fonction est utilisée lors de l'initialisation du processus.

`int mpi_vtophys_table_lookup_byvptr (mpi_contigmem_t*, caddr_t)`
Prend en argument un pointeur sur la table `mpi_vtophys_table` mappée dans l'espace utilisateur et une adresse virtuelle utilisateur. Elle retourne l'index correspondant au bloc ayant cette adresse virtuelle processus.

`int mpi_vtophys_table_lookup_bykptr (mpi_contigmem_t*, caddr_t)`
De même que `mpi_vtophys_table_lookup_byvptr()` renvoie un index dans la table en cherchant une adresse virtuelle processus, cette fonction renvoie un index dans la table correspondant a une adresse virtuelle noyau.

`mpi_contigmem_t mpi_get_usermem(size_t)`
Appelle l'IOCTL `MPI_GETUSERMEM` pour allouer un slot de mémoire CMEM. Le slot est ensuite mappé en mémoire par `mmap()`, et le champ `vptr` de l'entrée de table `mpi_vtophys_table` est mise à jour

`int mpi_release_usermem(mpi_contigmem_t)`
Appelle l'IOCTL `MPI_RELEASEUSERMEM` pour libérer un slot CMEM. Le mappage du slot est sup-

primé par `munmap()`, et l'entrée correspondante est supprimée de la table `mpi_vtophys_table`.

`int mpi_vtophys_table_erase(void)`
Appelle l'IOCTL `MPI_FREEUSERSLOT` pour libérer les ressources.

`int mpi_unmap_vtophys_table(mpi_contigmem_t*)`
Supprime le remappage de la table `mpi_vtophys_table` dans l'espace de mémoire du processus. Elle est utilisée lors de la terminaison du processus.

Enfin, ces fonctions sont appelées par d'autres fonctions de la librairie libmpich.a, qui, elles, sont utilisées par les programmes MPI :

`MPI_Initialize()` appelle `gmpi_subinit()`, qui elle même appelle `mpi_map_vtophys_table()` et `mpi_vtophys_table_erase()`. La Fig. 13 résume les interactions entre ces fonctions. Actuellement, cette dernière fonction libère tout ce qui est répertorié dans la table `mpi_vtophys_table`. Dans une version multi-tâche, il faudrait se contenter de libérer les ressources appartenant à des processus morts.

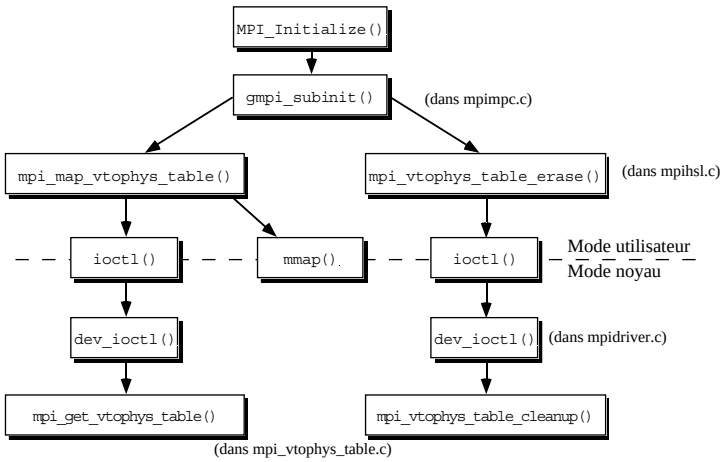

Fig. 13 : `MPI_Initialize` *et les fonctions concernant* `mpi_vtophys_table`.

`MPI_Finalize()` appelle `gmpi_subfinish()`, qui elle même appelle `mpi_unmap_vtophys_table()` et `mpi_vtophys_table_erase()`. La Fig. 14 résume les interractions entre ces fonctions. Dans une version multi-tâche, cette dernière fonction devrait ne libérer que des ressources appartenant au processus courant. On voit apparaître une certaine contradiction entre son rôle dans `gmpi_subinit()` et celui dans `gmpi_subfinish()`. Il faudra peut être introduire deux fonctions : une pour nettoyer à l'initialisation, et une autre pour nettoyer à la terminaison. Ou alors lui faire nettoyer les ressources associées aux processus morts et au processus courant.

`MPI_Malloc_MPC()` appelle `mpi_get_usermem()`, si on lui demande un bloc de mémoire assez gros. En effet, pour des petits blocs, les messages de contrôle seront utilisés, et donc on appelle le `malloc()` standard de la librairie C. La Fig. 15 résume le fonctionnement de `MPI_Malloc_MPC()`.

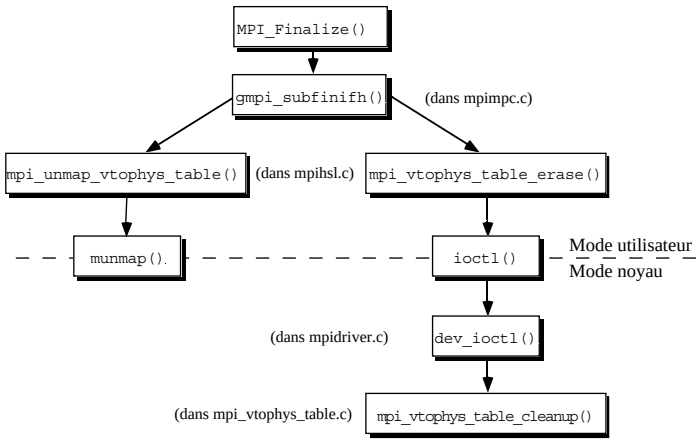

Fig. 14 : `MPI_Finalize()` *et les fonctions concernant* `mpi_vtophys_table`.

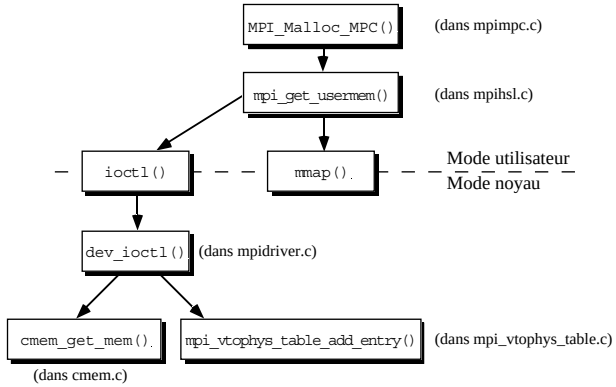

Fig. 15 : `MPI_Malloc_MPC()` *et les fonctions concernant* `mpi_vtophys_table`. *Le cas où* `malloc()` *de la libraire C est appellé n'est pas représenté.*

`MPI_Free_MPC()` appelle `mpi_release_usermem()` si le bloc peut être trouvé dans la table `mpi_vtophys_table`, ou bien le `free()` de la librairie C, s'il ne peut être trouvé. La Fig. 16 résume le fonctionnement de `MPI_Free_MPC()`.

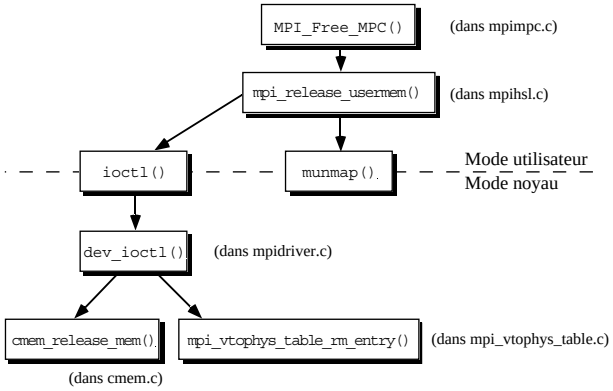

Fig. 16 : `MPI_Free_MPC()` *et les fonctions concernant* `mpi_vtophys_table`. *Le cas où* `free()` *de la libraire C est appellé n'est pas représenté.*

VIII Conclusions

VIII.1 Déroulement du stage

Le stage s'est déroulé ensentiellement en 5 étapes :

D'abord, il m'a fallu me familiariser avec la machine MPC, et réussir à faire fonctionner PUT/Linux sur des cartes FastHSL *first run* (c'était là qu'en était resté le portage).

Je me suis ensuite attelé à un travail de documentation des couches logicielles de MPC. Cette tâche n'était pas prévue dans mon sujet de stage, mais pour pouvoir finaliser le portage de PUT/Linux, il était indispensable que je comprenne bien le fonctionnement de PUT. La rédaction d'une documentation oblige à un effort de synthèse qui permet d'éviter de laisser des choses incomprises. Au cours de cette etape, j'ai écrit 31 pages man décrivant les interfaces de programmation de PUT, la configuration de la machine MPC, et les outils de tests.

Une fois PUT bien pris en main, j'ai commencé la fusion des sources de PUT/FreeBSD et PUT/Linux. Ce travail a été long et fastidieux, mais le jeu en valait la chandelle : Cela assurait que le code exécuté pour PUT Linux et PUT/FreeBSD était exactement le même. La recherche de bogues et l'intégration de modifications venant de PUT/FreeBSD s'en trouvait donc simplifié.

Une fois PUT fonctionnant sur les cartes équipées de PCIDDC *second run*, j'ai pu me livrer aux tests de performances, et aux corrections de problèmes décrits dans la partie VI de ce rapport.

Enfin, je me suis consacré à MPI sur MPC/Linux. Cette tâche s'est faite en collaboration avec Olivier Glück, qui encadrait mon stage. Elle a consisté en un travail de conception et d'implémentation, décrite dans la partie VII de ce présent rapport, ainsi que dans le portage sous Linux des éléments existants précédemment sous FreeBSD.

VIII.2 Apports du stage

Ce stage m'a permis de pratiquer la programmation en mode noyau, et ainsi d'approfondir un domaine que je souhaitais découvrir plus avant, à savoir le fonctionnement du système d'exploitation UNIX, et tout particulièrement ses aspects de niveau le plus bas. J'ai pu en particulier étudier de près l'implémentation des drivers, et la façon dont ils accèdent au matériel.

VIII.3 Le futur de MPC/Linux

Les performances de MPC/Linux vis à vis de MPC/FreeBSD sont très attrayantes. L'idée a été lancée de passer la machine MPC du LIP6 sous Linux, mais elle se heurte à deux lacunes importantes : A ce jour, les couches SLR/P et SLR/V sur lesquelles se base PVM n'ont pas été portées sous Linux. Le MPC-JMS, utilisé par PVM et MPI, n'est pas disponible non plus.

Le MPC-JMS serait portable sous Linux sans trop de peine, car il tourne complètement en mode utilisateur. De plus, moyennant de se donner un peu de peine, on peut utiliser MPI sans le MPC-JMS (c'est ce qui était fait pour développer MPI sur MPC/Linux). Par contre, les couches SLR/P et SLR/V fonctionnent en mode noyau, et seront donc beaucoup plus difficile à porter sous Linux.

Dans l'état actuel des choses, MPC/Linux n'est réellement utilisable qu'avec PUT, et ceci réduit fortement son intérêt. Son avenir dépend donc de la volonté de poursuivre son développement. En particulier, la disponibilité de MPI pour MPC/Linux sera sans doute un point déterminant.

VIII.4 Le futur du projet MPC

Dans sa forme actuelle, la machine MPC bute sur un certain nombre de problèmes difficilement solubles. Les années de développement autour du projet ne sont pas stériles pour autant, puisque plusieurs industriels se sont intéressés aux composants RCube et PCIDDC. Ceux ci sont aujourd'hui utilisés dans plusieurs machines industrielles. La technologie d'interconnexion de MPC a connu suffisamment de succès pour devenir le standard IEEE1355.

Le futur de la machine MPC se dessine aujourd'hui autour de nouveaux circuits reprogrammables, permettant une co-conception logicielle/matérielle. Ceci devrait éviter certaines difficultés telles qu'on a pu en rencontrer avec la solution RCube/PCIDDC.

Bibliographie

Sites Internet

Site de l'Université de Paris VI	http://www.admp6.jussieu.fr
Site du LIP6	http://www.lip6.fr
Site du département ASIM	http://www-asim.lip6.fr
Site du projet MPC	http://mpc.lip6.fr
Site du projet FreeBSD	http://www.freebsd.org
Site du projet de documentation Linux	http://www.linuxdoc.org

Documentation et ouvrages

PCIDDC specification
http://mpc.lip6.fr/pciddc/spec_pciddc.ps.gz

Emulateur Logiciel MPC/1 Manuel d'Installation
http://mpc.lip6.Fr/docemu/docemu.ps.gz

Interface Logicielle MPC/1 Manuel du Programmeur
http://mpc.lip6.fr/docutil/docutil.ps.gz

The Linux Kernel
http://metalab.unc.edu/pub/Linux/docs/linux-doc-project/linux-kernel/tlk-0.8-3.ps.gz

Design and implementation of the 4.4BSD operating system
McKusick Bostic, Karels, et Quarterman, Addison-Wesley, Reading, MA, 1996.

The FreeBSD Handbook
ftp://frp.freebsd.org/pub/FreeBSD/doc/handbook.txt.gz

Pages man de FreeBSD
http://www.FreeBSD.org/cgi/man.cgi?manpath=FreeBSD+3.4-RELEASE

Pages man de Linux
http://linux.wiw.org/doc/man/

Annexe A : le fichier `osdep.h`

```
/* $Id$ */

/*
 * osdep.h - Operating System dependant macros
 *
 * Operating System dependant stuff should be packed in a macro
 * that would be expanded here.
 *
 * Manu 2000/03/17
 *
 *
 */

#ifndef _OSDEP_H_
#define _OSDEP_H_

#ifdef linux
#define boolean_t int
#define vm_offset_t void*
#define vm_size_t size_t
/* #define pcici_t struct pci_dev* */
typedef struct pci_dev* pcici_t;
#define pcireg_t u_char
#define PCI_COMMAND_STATUS_REG (PCI_COMMAND)
#define PCI_COMMAND_MASTER_ENABLE (PCI_COMMAND_MASTER)
#define PCI_STATUS_PARITY_DETECT (PCI_STATUS_DETECTED_PARITY << 16)
#define PCI_STATUS_SPECIAL_ERROR (PCI_STATUS_SIG_SYSTEM_ERROR << 16)
#define PCI_STATUS_MASTER_ABORT (PCI_STATUS_REC_MASTER_ABORT << 16)
#define PCI_STATUS_MASTER_TARGET_ABORT (PCI_STATUS_REC_TARGET_ABORT << 16)
#define PCI_STATUS_TARGET_TARGET_ABORT (PCI_STATUS_SIG_TARGET_ABORT << 16)
#define PCI_STATUS_PARITY_ERROR (PCI_STATUS_PARITY << 16)
#define PCI_COMMAND_SERR_ENABLE (PCI_COMMAND_SERR)
#define PCI_COMMAND_PARITY_ENABLE (PCI_COMMAND_PARITY)
#define PCI_HEADER_MISC (0x0c)
#define WAKEUP wake_up
#define WAKEUP_FLUSH_INTERRUPTS wake_up(&flush_interrupts_queue)
#ifdef __KERNEL__
#define MALLOC(a, b, c, d, e) (a)=(b)kmalloc((c),GFP_KERNEL)
#define FREE(a,b) kfree((a))
#endif __KERNEL__
#define bzero(X,Y) memset((X),0,(Y))
/* #warning "Voir Documentation/spinlocks.txt..." */
#define CRITICAL_VAR    unsigned long  critical_flags
#define CRITICAL_START  { save_flags(critical_flags); cli(); }
```

```
#define CRITICAL_END    restore_flags(critical_flags)
#define VIRT_TO_PHYS(x)   (u_long)((u_long)(x)-
(u_long)opt_contig_space+opt_conti
g_space_phys)
#endif /* linux */

#ifdef __FreeBSD__
#define pcireg_t u_long
#define WAKEUP wakeup
#define WAKEUP_FLUSH_INTERRUPTS wakeup(&flush_interrupts)
#define HSL_LOG_DEBUG(ARGS...) log(LOG_DEBUG, ## ARGS)
#define HSL_LOG_ERR(ARGS...) log(LOG_ERR, ## ARGS)
#define HSL_LOG_WARNING(ARGS...) log(LOG_WARNING, ## ARGS)
#define CRITICAL_VAR int s
#define CRITICAL_START s=splhigh()
#define CRITICAL_END splx(s)
#define VIRT_TO_PHYS(x) vtophys(x)
#define HSL_READ(a,b,c) copyin((b), (a), (c))
#define HSL_WRITE(a,b,c) uiomove((b), (c), (a))
#define HSL_LOG_CONSOLE printf
#endif /* __FreeBSD__ */

#endif /* _OSDEP_H_ */
```

Annexe B : Quelques pages man

Cette annexe regroupe quelques unes des 40 pages man que j'ai écrit pour documenter MPC-OS.

B.1 mpc-intro(8)

```
MPC-INTRO(8)              FreeBSD System Manager's Manual              MPC-INTRO(8)

NAME
     mpc-intro - Quick introduction to the MPC machine installation.

DESCRIPTION
     The MPC machine is a low-cost parallel machine, based on industry stan-
     dard 80x86 PCs. The mecanism used for communication is a remote write us-
     ing DMA in the destination physical memory.

     The hardware components of the MPC machine are:

     A high bandwidth network, used for data transmissions
             The data network is made of RCube routers. Each RCube router fea-
             tures eight 1Gb/s HSL links.

     A low bandwidth network, used for control
             The control network is any IP capable link. Currently, a 10Mb/s
             ethernet is used.

     A set of 80x86 PCs
             Each PC has an ethernet board, to get a connection to the control
             network, and a PCI board containing the RCube router, that is
             used to connect to the data network. Note that even if you do not
             have HSL boards, you can still test your programs on the MPC ma-
             chine, using the emulation mode: data will be sent through the
             control network.

     MPC nodes run the FreeBSD operating system. A Linux port is also in pro-
     gres. Some software component are used to access the HSL board and to
     maintain the control opérations between the nodes:

     The cmem driver
             This driver is used to allocate contiguous blocks of physical
             memory. See cmem(4) man page for more details about what this
             driver is doing.

     The hsl driver
             This driver is used to get access to the HSL board. It also im-
             plements various communication layers. See hsl(4) man page for
             more details.

     hslclient  and hslserver
             Theses two daemons are used to maintain the control link between
             nodes.

     Please note that if you plan to work in emulation mode, without the HSL
     boards, you still need all the software components, including the hsl
```

```
      driver.

SETTING UP THE MPC machine
      Here is the steps that are to be done before applications such as
      testputbench can be used:

   Loading the drivers
     See how to load the drivers in the man pages cmem(4) and hsl(4)

   Building the routing tables
     This is done by the ir command. See ir(8).

   Launching the control daemons
     Next you must launch the hslserver and hslclient control daemons.  See
     the hslserver(8) and hslclient(8) man pages of theses two commands for
     more information

      After this step, the MPC machine is ready for use.

AUTHOR(S)
      Alexandre Fenyo <alex@asim.lip6.fr>
      Linux port by Fred Arnoud <arnoud@ecoledoc.lip6.fr>
      Man page by Emmanuel Dreyfus <p99dreyf@criens.u-psud.fr>

SEE ALSO
      cmem(4),  hsl(4),  ir(8),  hslserver(8),  hslclient(8).

MPC-OS Administrator Guide       March 8th, 2000                          2
```

B.2 cmem(4)

CMEM(4) FreeBSD Kernel Interfaces Manual CMEM(4)

NAME

cmem - Driver for allocating buffers of contiguous physical memory

DESCRIPTION

The MPC machine communications work by writing in the destination node memory using DMA. During such an opération, the source node has no idea of the virtual memory mappings on the destination node. All it does is writing to a given remote physical address.

If we want to correctly receive the data on the destination node, we must allocate a buffer mapped to the physical memory where the source node is writing. This buffer must be made of contiguous memory, and must be locked (ie: It must not be swapped out).

The job of the cmem driver is to hold a pool of contiguous physical memory, and then to maintain a service of contiguous physical memory allocation. The functions exported by the cmem driver are only accessible from kernel space. They are currently used by the hsl driver, which make them accessible from user space using ioctl calls (on the hsl driver), or using the useraccess library. Programmers that are using cmem from user space are expected to use the useraccess library.

The memory hold by cmem is divided into slots. cmem_getmem() is used to get a contiguous memory slot. cmem_releasemem() is used to release the memory allocated using cmem_getmem(). The functions cmem_virtaddr() and cmem_phys_addr() are used to get the virtual and physical start address of a slot.

LOADING THE CMEM DRIVER ON FREEBSD

When starting, the cmem driver needs a pool of contiguous memory for later use. The problem is that as the system runs, physical memory fragments. On FreeBSD, it is possible to get a large unfragmented physical memory pool at early boot stage. Therefore, it is better to load the cmem driver as early as possible during the boot process, in the /etc/rc script.

For instance, doing it just after the / have been remounted in read/write mode (which is done by the mount -u -o rw / command) is fine.

The driver can be loaded by the following command:

/etc/mpc/modload -e cmeminit -o /etc/mpc/cmemdriver.sym /etc/mpc/cmemdriver.o

If you want to modify some parameters such as the size of the zone managed by cmem , on FreeBSD, you currently have to recompile the module.

After the module is loaded, you must make the device /dev/cmem for the cmem driver. This is done by typing

```
mknod /dev/cmem c major-device-number 0

Where major-device-number is the major device number for the cmem driver.
If you use LKM modules, you can get it using the modstat command (it is
in the Off field).  If you are using KLD module, the major-device-number
should be 121.

Anyway, on recent releases of MPC-OS for FreeBSD, you can load the module
by typing a

gmake load


in the MPC-OS source directory.

LOADING THE CMEM DRIVER ON LINUX
     On Linux, it is not possible to get a contiguous block of memory after
     the kernel is loaded. Hence, you must tell the Linux kernel to not use
     some of the memory, so that it remains free for cmem .  This is done by
     using the Ic mem option at boot time.

     Using the mem option, you can tell the Linux kernel how much memory it
     must use. For example, if you have 64MB and want to keep 2MB for the cmem
     driver, add the option mem=62M at boot time. If you are using lilo, check
     the lilo(8) man page for information about how to automate this task.

     Once this is set up, you can load the module at any time. Loading it from
     a script in is a good idea. You load the driver by the following command:

     insmod cmem.o cmem_size=2097152

     You must specify as cmem_size the memory you reserved for cmem , in
     bytes.  You can specify at module load time the following parameters:

     cmem_major
             The cmem major device number.

     cmem_nslots
             The number of slots maintained by the cmem driver.

     cmem_size
             The size of the memory used by cmem , as described above.

     cmem_phys_stat
             The beginning of the memory used by cmem . If you reserved your
             contiguous memory by the technique described above, it is localt-
             ed at the end of the address space, and you do not need to speci-
             fy this parameter: the driver will find it for you.

     Once the driver is loaded, you must make the special device file
     /dev/cmem. This is done by typing

     mknod /dev/cmem c major-device-number 0

     Where major-device-number is the major device number, as returned by
```

```
      cat /proc/devices

      Note that the document doc/Install.txt in the Linux source directory will
      give you more information (in french) about how to load the driver on
      Linux.

USING THE CMEM DRIVER
      As explained earlier, the cmem driver does not provide any function to
      user space. All you can do with it is dumping the driver status by typing

      cat /dev/cmem

      However, cmem must be loaded if you want to load the hsl driver.

BUGS
      None known yet?

AUTHOR(S)
      Alexandre Fenyo <alex@asim.lip6.fr>
      Linux port by Fred Arnoud <arnoud@ecoledoc.lip6.fr>
      Man page by Emmanuel Dreyfus <p99dreyf@criens.u-psud.fr>

SEE ALSO
      hsl(4), mpc-into(8), cmem_releasemem(9), cmem_virtaddr(9),
      cmem_phys_addr(9), cmem_getmem(9).

MPC-OS User Guide                March 7th, 2000                              3
```

B.3 hsl(4)

```
HSL(4)                  FreeBSD Kernel Interfaces Manual                  HSL(4)

NAME
     hsl - HSL board driver for MPC. Also implements PUT,SLR/P,SLR/V.

DESCRIPTION
     The MPC machine uses HSL boards to make the connexion to the data net-
     work. The hsl driver is used to handle theses boards. Additionnally, the
     hsl driver implements three layers in the MPC-OS software: PUT, SLR/P and
     SLR/V.

     The primitive implemented by the PUT layer is a remote write in the phys-
     ical memory of the destination node.

     The SLR/P layer implements send and receive primitives using physical ad-
     dresses.

     The SLR/V layer implements send and reveive primitives using virtual ad-
     dresses.

     SLR/P and SLR/V are not yet documented.

     The primitives implemented at the PUT layer are accessible from user
     space by ioctl() calls, or by the useraccess library. Programmers that
     use PUT features are expected to use the useraccess library.  PUT func-
     tions are also available from kernel space. See put-intro(9) for more in-
     formation about how to work with PUT in kernel space.

     The hsl driver is implemented on both FreeBSD and Linux as a kernel mod-
     ule. Before loading it, you must load the cmem driver, which is used by
     the hsl driver. Note that the functions of the cmem driver are made
     available to user space through ioctl() calls by the hsl driver.

LOADING THE HSL DRIVER ON FREEBSD
     Unlike the cmem driver, which has to be loaded at an early boot stage,
     the hsl driver can be loaded at any time. Loading it from /etc/rc.local
     is fine. The driver is loaded by the following command:

     /sbin/modload -A /etc/mpc/cmemdriver.sym -o /etc/mpc/hsldriver.sym
     /etc/mpc/hsldriver.o

     Once the driver is loaded, you must make the device special file
     /dev/hsl. this is done by the following command:

     mknod /dev/hsl c major-device-number 0

     Where major-device-number is the major device number for the hsl driver.
     If you use LKM modules, you can get it using the modstat command (it is
     in the Off field).  If you are using KLD modules, the major-device-number
     should be 122.

     Anyway, on recent releases of MPC-OS for FreeBSD, you can load the module
     by typing a
```

```
      gmake load

      in the MPC-OS source directory.

LOADING THE HSL DRIVER ON LINUX
      You can load the the hsl driver at any time. Loading it from a script in
      /etc/rc.d is a good idea. You load the driver by the following command:

      insmod hsl.o


      Once the driver is loaded, you must make the special device file
      /dev/hsl. This is done by typing

      mknod /dev/hsl c major-device-number 0

      Where major-device-number is the major device number, as returned by

      cat /proc/devices

      Note that the document doc/Install.txt in the Linux source directory will
      give you more information (in french) about how to load the driver on
      Linux.

USING THE HSL DRIVER
      From user space, hsl driver fonctionnalities are accessed by issuing
      ioctl() calls on the device special file /dev/hsl. All opérations sup-
      ported by the hsl driver are implemented through ioctl(), including read-
      ing and writing. This approach avoids data copy between user and kernel
      space and thus improve performance.

      However, programmers of userland applications using the hsl driver to get
      access to PUT are expected to use the library useraccess instead of
      ioctl() calls. This is why available ioctl() for hsl are not documented.

BUGS
      None known yet?

AUTHOR(S)
      Alexandre Fenyo <alex@asim.lip6.fr>
      Linux port by Fred Arnoud <arnoud@ecoledoc.lip6.fr>
      Man page by Emmanuel Dreyfus <p99dreyf@criens.u-psud.fr>

SEE ALSO
      cmem(4),  put-intro(9),  mpc-into(8).

MPC-OS User Guide                March 7th, 2000                              2
```

B.4 hslserver(8)

HSLSERVER(8) FreeBSD System Manager's Manual HSLSERVER(8)

NAME

hslserver - Configuration daemon for the MPC machine.

SYNOPSIS

hslserver

DESCRIPTION

The MPC machine needs a control connexions between the different nodes. This is achieved by running two daemons: hslserver and hslclient , on every node of the MPC machine.

hslserver and hslclient use Remote Procedure Calls (RPC) over the control network to communicate. When invoked, hslserver waits for incomming connexions from hslclient processes running on remote nodes. Because hslserver does not deteach itself from the controlling terminal, you might like to run it in the background.

OPTIONS

hslserver does not have any options.

EXAMPLES

hslserver &

Launch hslserver in the background.

DIAGNOSTICS

Initialisation de pinguin
Unknown error. Spefific to the Linux port.

open: no such file or directory
This message probably means that hslserver was unable to open the hsl driver /dev/hsl .

atexit
Unknown error.

ioctl HSLWRITEPHYS
An ioctl call to the hsl driver failed.

ioctl HSLSIMINT
An ioctl call to the hsl driver failed.

BUGS

hslserver Should be a real deamon and put itself in the background once initialisation was successful.

Couldn't hslserver and hslclient be invoked by the same command?

Many other user interface enhancement possible in the way errors are reported.

```
AUTHOR(S)
      Alexandre Fenyo <alex@asim.lip6.fr>
      Linux port by Fred Arnoud <arnoud@ecoledoc.lip6.fr>
      Man page by Emmanuel Dreyfus <p99dreyf@criens.u-psud.fr>

SEE ALSO
      hslclient(8),  mpc-intro(8).

MPC-OS administrator Guide       March 7th, 2000                              1
```

B.5 hslclient(8)

```
HSLCLIENT(8)              FreeBSD System Manager's Manual              HSLCLIENT(8)

NAME
     hslclient - Configuration daemon for the MPC machine.

SYNOPSIS
     hslclient -f config-file -d hsl-device [-r route-file] [-e error-rate]

DESCRIPTION
     The MPC machine needs a control connexions between the different nodes.
     This is achieved by running two daemons: hslclient and hslserver , on ev-
     ery node of the MPC machine.

     hslclient and hslserver use Remote Procedure Calls (RPC) over the control
     network to communicate. When invoked, hslclient tries to connect to
     hslserver processes running on remote nodes. Because hslclient does not
     deteach itself from the controlling terminal, you might like to run it in
     the background.

     When running in emulation mode, you do not need to specify a routing
     table file route-file. But if you do not run in emulation mode, you must
     use a routing table file. Please check the ri(8) man page if you need in-
     formations about how to create a routing table file.

OPTIONS
     The following options are available:

     -f config-file
             Uses config-file as the configuration file for the local node.

     -d hsl-device
             Uses hsl-device as the device special file for the hsl driver.
             Usually this is /dev/hsl .

     -r route-file
             Uses route-file to get routing information for the local node.

     -e error-rate
             Undocumented option.

EXAMPLES
     hslclient -f /etc/mpc/conf/hsl01.cfg -d /dev/hsl -r
     /etc/mpc/conf/hsl01.raw &

     Launch hslclient in the background, using configuration file
     /etc/mpc/conf/hsl01.cfg and routing table /etc/mpc/conf/hsl01.raw.

DIAGNOSTICS
     fopen
           The config-file you specified does not exists, or you do not have
           read access to it.

     fgets
```

```
            Your config-file is damaged or invalid.

      open  The route-file or the hsl-device you specified does not exists, or
            you do not have read access to it.

      mmap  mmap() failed.

      init_put
            PUT layer initialisation failed in the HSL driver.

      ioctl HSLPUTEND

            Unknown error.

      ioctl HSLGETLPE
            Unknown error. Cannot get list of pages to be sent.

      malloc
            Cannot allocate memory.

      Many other diagnostics...

BUGS
      hslclient Should be a real deamon and put itself in the background once
      initialisation was successful.

      Couldn't hslclient and hslserver be invoked by the same command?

      There could be default values for the -f, -d, and -r flags.

      Many other user interface enhancement possible in the way errors are re-
      ported.

AUTHOR(S)
      Alexandre Fenyo <alex@asim.lip6.fr>
      Linux port by Fred Arnoud <arnoud@ecoledoc.lip6.fr>
      Man page by Emmanuel Dreyfus <p99dreyf@criens.u-psud.fr>

SEE ALSO
      hslserver(8),  mpc-intro(8),  and ri(8).

MPC-OS administrator Guide       March 7th, 2000                          2
```

B.6 testputbench (1)

```
TESTPUTBENCH(1)          FreeBSD General Commands Manual          TESTPUTBENCH(1)

NAME
     testputbench - Benchmark tool for the MPC machine.

SYNOPSIS
     testputbench -1 -s message-size -a remote-physical-address -n remote-node
     [-d] [-c] [-p page-size] [-i red-zone-value] [-o red-zone-size] [-l
     phase-limit] [-m | -M freq-MHz] [-S]
     testputbench -2 -s message-size -a remote-physical-address -n remote-node
     [-d] [-c] [-p page-size] [-i red-zone-value] [-o red-zone-size] [-l
     phase-limit] [-m | -M freq-MHz] [-S]

DESCRIPTION
     testputbench is a tool to benchmark the PUT layer of the MPC machine. It
     must be used with testputrecv.

     Before running testputbench you must setup the MPC machine: load appro-
     priate drivers, and run hslserver and hslclient. See mpc-intro(8) for
     more information about how to setup the MPC machine. Additionnaly,
     testputrecv must be running on both source and destination nodes, because
     testputbench uses memory allocated by testputrecv.

OPTIONS
     -d      Run in debbug mode. Generate much more messages about what is go-
             ing on.

     -c      Check data for transmission errors.

     -1      Run as source node (This testputbench should be started after the
             testputbench on the destination node).

     -2      Run as destiation node (This testputbench should be started be-
             fore the testputbench on the source node).

     -p page-size
             Uses page-size as the page size. Default is 64KB. When sent, mes-
             sages are split into pages of this size.

     -s message-size
             Uses message-size as the size of sent messages.

     -n remote-node
             Specify remote-node as the remote node.

     -a remote-physical-address
             Specify remote-physical-address as the physical address where the
             data should be sent on the remote node.

     -o red-zone-size
             Specify red-zone-size as the size of the red zone between the be-
             ginning of the buffer and the data to be transmitted. The red
             zone is used to check that no data is sent outside the target ad-
```

```
dresses. 8 bytes is a good value.

-i red-zone-values
        Use red-zone-values to initialize the red zone. Default is 0.

-S      Silent mode. Does not output the phase count. Useful when using
        testputbench in scripts.

-l phase-limit


        Stops testputbench after phase-limit phases have been executed.

-m      Measure how long the testputbench lasts. The CPU frequency will
        be estimated by a one second long test at the beginning. Note
        that only the initiator host  (the machine where testputbench was
        started with the -1 flag) will display a result. This is is due
        to the fact that the receptor first waits for the initator, and
        hence it would display a biased measure that include the waiting
        time at the beginning of the test.

-M freq-MHz
        Same as -m but the user supply the CPU frequency in MHz. This al-
        lows the user to supply a good value for the CPU speed (the value
        estimated by -m is not very accurate).

EXAMPLES
     testputrecv
     testputbench -c -i 12 -2 -o 8 -s 4096 -a 0x3e00000 -n 0

     On the destination node (node 1).

     testputrecv
     testputbench -c -i 12 -1 -o 8 -s 4096 -a 0x3e00000 -n 1

     On the source node (node 0).  The address used for the -a flag is readen
     from testputrecv output.


DIAGNOSTICS
     cannot allocate memory
           You did not launch testputrecv before running testputbench

     Many diagnostics...
           To be finished later.

BUGS
     Many user interface enhancement possible in the way errors are reported.

     testputbench sould be able to allocate its memory so that it would not be
     necessary to invoke testputrecv.

AUTHOR(S)
     Alexandre Fenyo <alex@asim.lip6.fr>
     Linux port by Fred Arnoud <arnoud@ecoledoc.lip6.fr>
```

```
      Man page and time measurement stuff by Emmanuel Dreyfus
      <p99dreyf@criens.u-psud.fr>
      Time measurement uses timer.c from Philippe Lalevee
      <Philippe.Lalevee@int-evry.fr>

SEE ALSO
      hslserver(8),  hslclient(8),  mpc-intro(8),  testputrecv(1).

MPC-OS User Guide                March 7th, 2000                              2
```

B.7 testputsend_loop (1)

```
TESTPUTSEND_LOOP(1)      FreeBSD General Commands Manual     TESTPUTSEND_LOOP(1)

NAME
     testputsend_loop - Benchmark tool for the MPC machine.

SYNOPSIS
     testputsend_loop -s message-size -R remote-physical-address -L local-
     physical-address -n remote-node [-l phase-limit] [-m | -M freq-MHz] [-S]
     [flags]

DESCRIPTION
     testputsend_loop is a tool to benchmark the PUT layer of the MPC machine.
     It is merly a bandwidth measurement tool since it just try to send as
     many data as possible, without requesting replies from the targer host. .
     must be used with testputrecv.

     Before running testputsend_loop you must setup the MPC machine: load ap-
     propriate drivers, and run hslserver and hslclient. See mpc-intro(8) for
     more information about how to setup the MPC machine. Additionnaly,
     testputrecv must be running on both source and destination nodes, because
     testputsend_loop uses memory allocated by testputrecv.

OPTIONS
     -s message-size
             Uses message-size as the size of sent messages.

     -n remote-node
             Specify remote-node as the remote node.

     -R remote-physical-address
             Specify remote-physical-address as the physical address where the
             data should be sent on the remote node.

     -L local-physical-address
             Specify local-physical-address as the physical address of the da-
             ta that should be sent to the remote node.

     -S      Silent mode. Does not output the phase count. Useful when using
             testputsend_loop in scripts.

     -l phase-limit
             Stops testputsend_loop after phase-limit phases have been execut-
             ed.

     -m      Measure how long the testputsend_loop lasts. The CPU frequency
             will be estimated by a one second long test at the beginning.
             Note that only the initiator host  (the machine where testput-
             send_loop was started with the -1 flag) will display a result.
             This is is due to the fact that the receptor first waits for the
             initator, and hence it would display a biased measure that in-
             clude the waiting time at the beginning of the test.

     -M freq-MHz
```

```
            Same as -m but the user supply the CPU frequency in MHz. This al-
            lows the user to supply a good value for the CPU speed (the value
            estimated by -m is not very accurate).

      Additionnnally, you can supply one or more of the following flags:

      NOR     trigger an interruption on remote node when data is transfered.

      LMP     the message is considered as the end of a packet.


      LRM     not yet functionnal.

      CM      message from N nodes to one node. Not yet functionnal.

      LMI     update the list of Message Identifers (MI) on remote node upon
              reception of data

      SM      short message: data are embedded in the header.

      NOS     trigger an interruption on local node when data is transfered.

      RESERVED
              reserved for future use.

      NONE    no option.

      Note that the flags must be separated by a ``|'', and that you must quote
      them, in order to prevent interpretation by the shell.

EXAMPLES
      testputrecv
      testputsend_loop -s 4096 -R 0x3e00000 -L 0x3200000 -n 0 -M 166 -l 100000
      'LMP|LMI'

      Send 100000 packets of 4096 bytes to node 0. Local and remote physical
      addresses as given in -L and -R are readen from testputrecv output. Addi-
      tionnally, by the -M -flag, the user request the measurement of the time
      taken by the opération. For that measure, a CPU frequency of 166MHz is
      given. The flags LMP and LMI are set for the opération.

DIAGNOSTICS
      cannot allocate memory
            You did not launch testputrecv before running testputsend_loop

      Many diagnostics...
            To be finished later.

BUGS
      Many user interface enhancement possible in the way errors are reported.

      testputsend_loop sould be able to allocate its memory so that it would
      not be necessary to invoke testputrecv.

      Specifying the NOR flag without the LMI flag, or the NOS flag without the
```

```
     LMP flag will cause you an unpleasant kernel crash.

AUTHOR(S)
     Alexandre Fenyo <alex@asim.lip6.fr>
     Man page and time measurement stuff by Emmanuel Dreyfus
     <p99dreyf@criens.u-psud.fr>
     Time measurement uses timer.c from Philippe Lalevee
     <Philippe.Lalevee@int-evry.fr>

SEE ALSO
     hslserver(8), hslclient(8), mpc-intro(8), testputrecv(1).

MPC-OS User Guide              Apris 20th, 2000                          2
```

B.8 testputrecv (1)

```
TESTPUTRECV(1)          FreeBSD General Commands Manual          TESTPUTRECV(1)

NAME
     testputrecv - Test tool for the MPC machine.

SYNOPSIS
     testputrecv

DESCRIPTION
     testputrecv is a tool to test the PUT layer of the MPC machine. It can be
     used with testputsend or testputbench.

     Before running testputrecv you must setup the MPC machine: load appropri-
     ate drivers, and run hslserver and hslclient. See mpc-intro(8) for more
     information about how to setup the MPC machine.

     When started, testputrecv allocate some memory, display various informa-
     tion, including the physical address of the beginning of the allocated
     buffer. After this, testputrecv loops, displaying the content of the
     first bytes of the allocated buffer. You can use that display to check
     that data sent by testputsend on a remote node successfully reach their
     destination.

     If you use testputrecv with testputbench, the display might be useless to
     you, so you might like to suspend testputrecv using a ^Z once it is
     started. You can also do this if you use testputsend on the remote node.
     Data will arrive, but you will not be able to see it.

     Please note that you need to run testputrecv on both source and destina-
     tion node where you run a testputbench . If you do not run testputrecv ,
     testputbench will refuse to work because they need to use the memory al-
     located by testputrecv .

OPTIONS
     No options are available.

EXAMPLES
     testputrecv

     Launch testputrecv . After reading the messages, you might prefer to sus-
     pend it.

DIAGNOSTICS
     Many diagnostics...
           To be finished later.

BUGS
     Many user interface enhancement possible in the way errors are reported.

AUTHOR(S)
     Alexandre Fenyo <alex@asim.lip6.fr>
     Linux port by Fred Arnoud <arnoud@ecoledoc.lip6.fr>
```

```
     Man page by Emmanuel Dreyfus <p99dreyf@criens.u-psud.fr>

SEE ALSO
     hslserver(8), hslclient(8), mpc-intro(8), testputsend(1), and
     testputbench(1).

MPC-OS User Guide                March 7th, 2000                              1
```

B.9 testmpibench (1)

```
TESTMPIBENCH(1)          FreeBSD General Commands Manual          TESTMPIBENCH(1)

NAME
     testmpibench - Benchmark tool for MPI.

SYNOPSIS
     testmpibench -1 -n target_mpi_task [-v] [-c] [-l samples] [-m | -M
     freq-MHz] [-s size]
     testmpibench -2 -n target_mpi_task [-v] [-c] [-l samples] [-m | -M
     freq-MHz] [-s size]

     testmpibench -1 -n target_mpi_task [-v] [-c] [-l samples] [-m | -M
     freq-MHz] [-p max_power]
     testmpibench -2 -n target_mpi_task [-v] [-c] [-l samples] [-m | -M
     freq-MHz] [-p max_power]

DESCRIPTION
     testmpibench is a benchmark tool for MPI. Its purpose is to compute
     throughput and lattency of the underlying MPI implementation.
     testmpibench have been developped to benchmark MPI over the MPC machine,
     but it should also work with any other MPI implementation.

     Before running testmpibench you must setup the MPC machine, and run
     mpinit(8).  See mpc-intro(8) for more information about how to setup the
     MPC machine.

     You must run testmpibench on two nodes of your MPC cluster. The first
     testmpibench process you launch will be the receptor and must have the
     flag -2, the second process will be the initator and must have the flag
     -1. There is another mandatory flag, -n, which tells testmpibench who is
     the target MPI task.

     The default behavior of testmpibench is to exchange a large amount of
     packets of a given size. You can choose the amount of packet with the -l
     flag, and the size of the packets with the -s flag .

     There is also a batch mode, that you trigger with the -p flag. In batch
     mode, testmpibench will exchange packets of 1 byte, then packets of 2
     bytes, then 4 bytes, 8 bytes, 16 bytes, and so on. With the -p flag, you
     control the last power of two that testmpibench will use. For instance,
     with -p 8 testmpibench will exchange packets of 1, 2, 4, 8, 16, 32, 64
     and 128 bytes. You can control how many packets are sent for each packet
     size with the -l flag.

OPTIONS
     -1      Run as source node (This testmpibench should be started after the
             testmpibench on the destination node).

     -2      Run as destiation node (This testmpibench should be started be-
             fore the testmpibench

     -n target_mpi_task
             Specify target_mpi_task as the target MPI task. Currently, there
```

```
            is one MPI task per MPC node, and the MPI task number is equal to
            the MPC node number.

      -v    Run in verbose mode. This generates a lot of output.

      -c    Check data for transmission errors.

      -l samples
            for each packet size, send samples packets.

      -m    Measure how long the testmpibench lasts. The CPU frequency will
            be estimated by a one second long test at the beginning. Note
            that only the initiator host (the machine where testmpibench was
            started with the -1 flag) will display a result. This is is due
            to the fact that the receptor first waits for the initator, and
            hence it would display a biased measure that include the waiting
            time at the beginning of the test.

      -M freq-MHz
            Same as -m but the user supply the CPU frequency in MHz. This al-
            lows the user to supply a good value for the CPU speed (the value
            estimated by -m is not very accurate).

      -s size
            Send packets of size size

      -p max-power
            Send packets of size equal to two power n, with n going from 0 to
            max-power - 1.

EXAMPLES
      testmpibench -2 -n 0 -l100000 -p 3

      On the destination node (node 1).

      testmpibench -1 -n 1 -l100000 -p 3

      On the source node (node 0). This will send 100000 packets of 1 byte,
      then 100000 packets of 2 bytes, and then 100000 packets of 4 bytes.


DIAGNOSTICS
      data error
            A data transfer error occured.  testmpibench exits immediatly
            when such an error is discovered.

      various problems from libmpich.a
            Error reporting in libmpich.a is not always helpful, and some er-
            rors there can lead testmpibench to dump core.  If you get odd
            problems, it might be caused by configuration files that cannot
            be red or that contain invalid data (See section FILE for more
            info), or by /dev/kmem (or /dev/cmem for Linux) that cannot be
            open.

FILES
```

```
       /usr/local/mpich/mpid/ch_mpc/mpc.conf
              Contains "NUMBER_OF_NODES nodes ", where nodes is the number of
              nodes in the MPC machine.

       /usr/local/MPC-JMS/admin-bin/nodes-bin/mpi-cmem-addr.n
              Contains the physical address of the cmem memory used by MPI for
              MPC node n. This file is created by mpinit.

BUGS
       We assume that the two nodes CPUs are running at the same frequency.

       The only way to run testmpibench as a non root user is to have /dev/kmem
       (or /dev/cmem for the Linux port) world readable.  testmpibench should be
       made setgid kmem to fix this.

       There is nothing to prevent strange behavior if the user provide differ-
       ent -l, -s, or -p arguments to the initator and to the receptor.

       Path to configuration files are hardcoded in libmpich.a. This is not a
       testmpibench bug, but testmpibench will panic if one of this files cannot
       be red, or if it contains a wrong information (eg: wrong number of
       nodes). We could find a better way of getting the MPI configuration.

AUTHOR(S)
       Olivier Gluck <Olivier.Gluck@lip6.fr>
       Man page and user interface improvments by Emmanuel Dreyfus
       <p99dreyf@criens.u-psud.fr>
       Time measurement uses timer.c from Philippe Lalevee
       <Philippe.Lalevee@int-evry.fr>

SEE ALSO
       mpc-intro(8),  mpinit(8).

MPICH over MPC-OS User Guide     March 7th, 2000                                3
```

B.10 put-intro (9)

```
PUT-INTRO(9)            FreeBSD Kernel Developer's Manual            PUT-INTRO(9)

NAME
     put-intro - Developer introduction to the PUT layer.

DESCRIPTION
     The hsl driver is used to handle HSL boards. Additionnally, it also im-
     plements three layers in the MPC-OS software: PUT, SLR/P and SLR/V.

     The primitive implemented by the PUT layer is a remote write in the phys-
     ical memory of the destination node. This page is a quick introduction to
     PUT programming in kernel space.

PROGRAMMING WITH THE PUT LAYER
     The PUT layer maintain a tables to keep track of pages that are to be
     sent: it is referred as the List of Page to Emit (LPE) table.

     A LPE entry is defined as:
     typedef struct _lpe_entry {
     u_short page_length;
     u_short routing_part;
     u_long control;
     caddr_t PRSA,PLSA;
     } lpe_entry_t

     The page_length is the length of the page to be send, in bytes.  The
     routing_part is the destination node. The PRSA and PLSA are the remote
     and local physical starting addresses for the transfert.

     The control field is divided in two part: The 23 lower bits are the Mes-
     sage Identifier (MI), which is an unique identifier accross the whole
     network for one message, and the 8 higher bits are control flags:

     bit 31: NOR
           Trigger an interruption on remote node when data is transfered.

     bit 30: LMP
           The message is considered as the end of a packet.

     bit 29: LRM
           Not yet functionnal.

     bit 28: CM
           Message from N nodes to one node. Not yet functionnal.

     bit 27: LMI
           Update the list of Message Identifiers (MI) on remote node upon re-
           ception of data.

     bit 26: SM
           Short message: data are embedded in the header.

     bit 25: NOS
```

```
            Trigger an interruption on local node when data is transfered.

      bit 24: RESERVED
            Reserved for future use.

      bit 0 to 23:
            Message identifier (MI)

      To send a message, a PUT user must create LPE entries corresponding to
      the message. If the message is short enough, it can be send in only one
      page, and only one LPE is requiered. If the message is longer, it has to
      be split into several pages, and several LPE entries must be built.

      For a given message, all LPE must have the same MI. Because MI must be
      unique accross the whole network, they cannot be chosen randomly, and PUT
      provides functions to get unique MI.

      The first function a PUT user should call is put_register_SAP().  It
      gives the caller a SAP numer, that is to be used for various other PUT
      functions. Additionnaly, it allows the user to specify callback functions
      that will be asynchronously called upon send and receive completion. When
      the user is done with PUT, put_unregister_SAP() should be called to re-
      lease allocated resources.

      put_get_node() is used to get the node number of an HSL board. This node
      is a unique identifier for an HSL board accross the whole network.

      put_attach_mi_range(), telling how much MI he will need. After
      put_attach_mi_range() have been called once, put_get_mi_start() Can be
      called to get valid a valid MI.  put_get_mi_start() canbe called several
      times to get multiples MIs.

      The put_add_entry() function is used to add a LPE to the LPE table. It is
      possible to get a pointer to the LPE entry a HSL board is working on by
      using put_get_lpe_high().  Finnally, put_get_lpe_free() can be used to
      get the number of free entries in the LPE table.

      This is just a quick introduction. Please check the man page of each
      function for more information.

BUGS
      None known yet?

AUTHOR(S)
      Alexandre Fenyo <alex@asim.lip6.fr>
      Linux port by Fred Arnoud <arnoud@ecoledoc.lip6.fr>
      Man page by Emmanuel Dreyfus <p99dreyf@criens.u-psud.fr>

SEE ALSO
      cmem(4),  put-intro(9),  put_get_node(9),  put_register_SAP(9),
      put_unregister_SAP(9),  put_attach_mi_range(9),  put_get_mi_start(9),
      put_get_lpe_high(9),  put_get_lpe_free(9),  put_add_entry(9),  mpc-
      into(8).

MPC-OS développer Guide          March 7th, 2000                          2
```