



# MÉMOIRE DE STAGE MASTER 2

Stage du 1<sup>er</sup> février 2023 au 31 juillet 2023

Intitulé : Interruptions en espace utilisateur pour le réseau  
BXI

Charles GOEDEFROIT

Encadré par Alexandre DENIS (Inria), Grégoire PICHON (Atos) et  
Mathieu BARBE (Atos)

28 juillet 2023

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Présentation Inria . . . . .	2
1.2	Présentation Atos (Eviden) . . . . .	2
1.3	Environnement de travail . . . . .	2
1.4	Le cadre . . . . .	2
1.5	Présentation du plan . . . . .	3
<b>2</b>	<b>Context</b>	<b>3</b>
2.1	HPC . . . . .	3
2.2	OS bypass . . . . .	4
2.2.1	Les interruptions matérielle . . . . .	5
2.2.2	TODO : trouvé un titre . . . . .	5
2.3	Polling . . . . .	5
2.3.1	Inconvénients du polling . . . . .	6
2.4	BXI . . . . .	6
2.5	MPI . . . . .	6
2.5.1	Les communications point à point . . . . .	7
2.5.2	Les communications collective . . . . .	7
2.5.3	La progression en tâche de fond . . . . .	7
2.6	NewMadeleine . . . . .	8
2.7	Les signaux . . . . .	8
2.8	Travaux antérieur . . . . .	9
<b>3</b>	<b>Problématiques / Objectifs</b>	<b>9</b>
3.1	Sujet . . . . .	9
3.1.1	Nouveau mécanisme d'interruption en espace utilisateur . . . . .	9
3.1.2	contentions... . . . .	9
3.2	Projet global . . . . .	10
3.2.1	Les objectifs . . . . .	10
3.3	Objectifs du stage . . . . .	10
3.4	La suite . . . . .	11
<b>4</b>	<b>Exploration de uintr</b>	<b>11</b>
4.1	Prérequis est accès . . . . .	11
4.2	Fonctionnement des uintr . . . . .	12
4.2.1	Les interruptions . . . . .	12
4.2.2	Les uintr . . . . .	14
4.2.3	Capacités présente et futur . . . . .	15
4.2.4	Exemple de fonctionnement . . . . .	17
4.2.5	Partage du descripteur de fichier . . . . .	19
4.3	Testes du mécanisme . . . . .	19
4.4	Correction d'un bug dans le patch noyau . . . . .	21

4.5	Mesure de la latence . . . . .	22
4.6	Performances . . . . .	24
<b>5</b>	<b>Intégration dans NewMadeleine</b>	<b>28</b>
5.1	Présenté NewMadeleine details . . . . .	28
5.2	Modification . . . . .	29
5.3	Suite de testes . . . . .	29
5.4	Performances . . . . .	30
5.4.1	Résultats avec attente active . . . . .	30
5.4.2	Résultats recouvrement communication par du calcule	30
<b>6</b>	<b>Bilan</b>	<b>30</b>
<b>7</b>	<b>Remerciements</b>	<b>30</b>
<b>8</b>	<b>Annexes</b>	<b>31</b>

# 1 Introduction

J'ai effectué mon stage de 6 mois dans l'équipe-projet TADaaM du Centre Inria de l'université de Bordeaux. Le sujet du stage à été proposé par Alexandre Denis (Inria) et Grégoire Pichon (Atos). Ils ont aussi encadré le stage avec Mathieu BARBE (Atos).

## 1.1 Présentation Inria

Inria est l'institut national français de recherche en sciences et technologies du numérique. Il compte plus de 3 900 chercheurs et ingénieurs au sein de 215 équipes-projets. La plupart des centres sont communs avec une grande université.

## 1.2 Présentation Atos (Eviden)

Atos est un leader international de la transformation numérique. Elle couvre un champ large d'activité : le cloud, la cybersécurité, les services transactionnels, le conseil, l'infogérance, le Big Data, les supercalculateurs... Atos compte 112 000 collaborateurs. Atos est en restructuration pour ce séparé en 2 entités. L'entité qui nous intéresse pour ce stage est Eviden qui englobe notamment les activités autour des supercalculateurs et du HPC. Cette restructuration est récente donc quand Atos est évoqué dans ce document nous parlons de la partie Eviden.

## 1.3 Environnement de travail

Mon environnement de travail c'est les locaux du Centre Inria de l'université de Bordeaux. On a mis à disposition un bureau dans l'open space de l'équipe-projet TADaaM. Je peux participer et assister à des activités scientifiques intéressantes (séminaire, soutenance de thèse, soutenance HDR, activités diverses...). Nous avons accès aux salles de visio conférence, notamment pour les réunions de suivi de stage hebdomadaire. Il y a aussi un Baby-foot, une cafeteria, une petite Médiathèque...

## 1.4 Le cadre

Du côté de l'équipe TADaaM de Inria l'objectif de l'équipe est de faire de la recherche sur les sujets suivants :

- Gestion des I/O (ordonnancement, bande passante...)
- Placement de processus
- Partitionnement de maillage (i.e. SCOTCH)
- Localité matérielle (i.e. hwloc)

- Optimisation des communications pour les réseaux haute performance, MPI (i.e. NewMadeleine)

Elle est composée de chercheurs, ingénieurs de recherche, post-doc, doctorant et stagiaire. La culture informatique de l'équipe est l'utilisation des environnements linux, des logiciels open source, des cluster de calcul HPC, le traitement des données et le système. L'équipe a mis à ma disposition un ordinateur portable avec une station de travail brancher à un écran, à internet, à un clavier et une souris. Inria donne aussi accès à un ensemble de services, boîte email, service de communication Mattermost, service de visio Webex, intranet avec le menu de la cafeteria... Nous sommes libre d'installer le système d'exploitation et d'utiliser les outils informatiques que l'on veut.

Du côté d'Atos l'équipe BXI-LL (BXI Low Level) est une équipe dont l'objectif est de fournir le support logiciels bas niveau pour les cartes réseaux BXI. Cela consiste à développer les drivers de la carte, le support Lustre – un système de fichier parallèle et distribué qui est utilisé pour l'I/O – ne fait pas partie de ce stage – ainsi que l'interface logiciels Portal4. Cette interface permet aux implémentations MPI d'utiliser les réseaux BXI. Le personnel est composé d'ingénieurs, de doctorant et de stagiaire. /\* TODO : vérifier \*/ La culture informatique de l'équipe est l'utilisation des clusters HPC, la programmation système et l'utilisation d'un système de sécurité PKI pour accéder aux ordinateurs et aux services internes. Atos nous a donné accès à une machine avec deux CPU Intel® Sapphire Rapids. Pour accéder à cette machine ils nous ont fourni un accès ssh et un compte VPN. Nous avons eu tous les droits d'accès sur cette machine pour changer le noyau du système.

## 1.5 Présentation du plan

Dans ce mémoire nous allons commencer par une présentation du contexte dans lequel le stage se place. Puis nous continuerons avec les objectifs à long terme et les objectifs du stage. Nous verrons ensuite le nouveau mécanisme d'interruption et ce que nous avons fait avec. Nous continuerons par la présentation de l'intégration de ce mécanisme dans la bibliothèque de communication NewMadeleine et les tests que l'on a effectués. Pour finir un bilan du stage ainsi que les travaux qui vont suivre.

## 2 Contexte

### 2.1 HPC

Le stage s'est déroulé dans le contexte du Calcul Haute Performance (HPC, High-Performance Computing en anglais). Le HPC utilise des supercalculateurs pour la simulation numérique et le pré-apprentissage d'in-

telligences artificielles. Ces simulation simule la dynamique des fluides, la résistance structurelle, les interaction moléculaire, les flux d'aire...

Elles couvrent différent domaines :

- L'industrie : le médicales, l'automobiles, l'aviations, la constructions, l'aérospatiales...
- La défense : simulation atomique
- La recherche scientifiques : la création des galaxie, la fusion nucléaire, le climat...
- La météo

De nos jour les supercalculateurs sont composé de plusieurs ordinateur que l'on appelle noeud de calculs. Ceci sont regroupé en grappe (cluster), il sont tous vue est utiliser comme une seul grande machine. Ce fonctionnement pose des questions sur la répartition du calcul, la distribution de la mémoire, la communication entre les différent noeuds... Dans le contexte du stage, nous nous sommes concentrés sur les communications entre noeuds. Les noeuds sont connecté entre eux par un réseau haute performance. Ce réseau est dédiés au communications est n'est pas forcément de type Ethernet. La gestion des noeuds est généralement effectuée par un second réseau plus traditionnel (Ethernet ; TCP/IP). Les réseaux haute performance ont une latence autour de quelque microseconde. /\* TODO : insisté sur la différence entre les réseaux pour le HPC et les réseaux classiques \*/ Chaque noeuds possède une ou plusieurs carte réseau et il faut les programmé.

## 2.2 OS bypass

Le stage ce passe donc aussi dans un context système. Chaque noeuds à sont propre système d'exploitation qui permet la gestion des ressources, des processus, des fichiers, des périphériques. Pour cela le système à 2 espaces :

- un espace noyau où seul le code du système peut s'exécuter. Le code du système peut donc modifié n'importe quelle endrois de la mémoire, exécuté n'importe quelle instructions...
- un espace utilisateur où le code de l'utilisateur est exécuter. Cette espace est limité par le noyau qui controle ce que fait l'utilisateur.

En temps normale les périphériques sont programmé directement depuis le noyau du système d'exploitation ceci pour des questions de sécurité, de standardisation des accès... Lors que l'on passe par le noyau (kernel en anglais) on à un surcoût qui n'est pas négligeable dans notre cas. Pour passer par le noyau on utilise généralement des appels système qui ce présente sous la forme d'une fonction. Un appel système vas effectué un changement de contexte (context switch) pour passé de l'espace utilisateur à l'espace noyau.

Ce changement de context est coûteux car il sauvegarde les états du code de l'utilisateur avant d'exécuté celui du noyau. Une fois le context switch effectué le code noyau de l'appel système s'exécute avant de refaire un context switch pour cette fois passé de l'espace noyau à l'espace utilisateur, et donc restoré l'états du code de l'utilisateur. Donc le fait de passé par un appel système coûte plusieurs microseconde. On vois donc qu'il n'est pas préférable d'utiliser les appels système car un appel système est déjà du même ordre de temps qu'une communication. En HPC on programme donc directement la carte réseau à partir de l'espace utilisateur (OS bypass en anglais). Pour cela on initialise toujours la carte à partir du noyau mais on fait une projection de la mémoire et des registre de la carte réseau dans l'espace d'adressage virtuelle du processus utilisateur.

### **2.2.1 Les interruptions matérielle**

Les interruptions matérielle, aussi nommé IRQ pour Interrupt ReQuest, existe depuis long temps est serve notamment à remonté des exceptions d'un processeur au noyau. Par exemple elle sont utilisé quand il y a une eu une division par zero. Elle on ensuite été utilisé pour signaler des événements en provenance de périphériques et aussi pour de la communications entre processus. Dans ce document nous utiliseront le terme interruption ordinaire ou IRQ pour les désigner.

### **2.2.2 TODO : trouvé un titre**

Pour transmettre des événement à l'utilisateur les périphériques utilise généralement les interruptions ordinaire, qui passe par le système donc on ne les utilise très peu en HPC. En HPC on fait donc du polling.

## **2.3 Polling**

Le polling consiste à scruté (poll) régulièrement si on à reçus un événement. Concrètement cela consiste à lire une zone mémoire modifier par la carte réseau et voire si un bit est passé à 1. Pour cela il es possible de dédié un thread qui vas faire de l'attente active, scruté sans cesse si un événement à été reçus. Mais on perd une unité de calcule lors ce que le thread est ordonnancer donc de la puissance de calcul, donc cette technique est peu utilisé. Il est aussi possible d'entrelacés le code de l'utilisateur avec des scrutation, c'est fréquemment utiliser mais ça oblige à l'utilisateur de prendre en compte la progression. Une autre solution qui est utiliser par Pioman dans NewMadeleine (Nous en parlerons plus tard) consiste à faire ces scrutation de manière opportuniste dans les threads qui on fini leur calcul, mais pour cela il faut déjà utilisé plusieurs thread et avoir une application qui à des calculs d'une durée hétérogène.

Il faut donc, peut importe la technique, régulièrement scruté pour faire progressé les communications.

### 2.3.1 Inconvénients du polling

Dans le cas d'un thread dédié qui fait de l'attente active la réactivité est excellente hormis quand il y a plus de threads que d'unité de calcul est que le thread n'est pas ordonnancer. /\* TODO : cité ce papier <https://inria.hal.science/hal-03695835v1> et modifier \*/ Quand on entrelaces le calcul est les scrutations la réactivité est moins bonne car il faut attendre que le calcul soit fini pour faire un poll. C'est ce qui est fait habituellement.

Quand on utilise les thread de façon opportunist pour faire des scrutations la réactivité est moins bonne car il faut qu'il y est un thread disponible.

Donc un choix doit être fait entre perdre de la capacité de calcul ou perdre en réactivité. En plus on perd un peu de temps de calcul à effectuer du polling.

## 2.4 BXI

BXI pour Bull eXascale Interconnect est un type de réseau d'interconnection (réseau haute performance) développé par Atos. Historiquement développé par Bull qui a été racheté par Atos. Ce réseau est dédié aux communications entre noeuds. Il est composé de carte réseau BXI et de switch BXI. La carte BXI est capable de faire progressé la communication réseau sans aucune intervention du CPU (offload des communications réseau). Le CPU à juste à soumettre une commande dans une file sur la carte et elle s'occupe de tous. Le CPU peut ensuite récupérer une file d'événements pour savoir ce qu'il s'est passé, en somme faire un poll. La carte est également capable de déclencher des interruptions. L'utilisation de la carte passe donc par une implémentation du protocol Portal4. L'utilisateur utilise donc le protocol pour envoyer et recevoir des paquets réseau.

## 2.5 MPI

MPI pour Message Passing Interface est un standard pour fournir une interface pour effectuer des communications entre plusieurs processus, qui sont souvent sur des noeuds différents et qu'on appelle *processus MPI*. Cette interface fournit une abstraction pour transmettre des données entre plusieurs processus. L'abstraction masque la complexité des communications. L'interface permet d'envoyer et de recevoir des messages. Pour cela il y a deux modes de communications :



### 2.5.1 Les communications point à point

C'est à dire entre deux processus MPI, aussi appelé One-to-One. Pour ce faire le processus MPI récepteur vas appelé la fonction `MPI_Recv` qui est bloquante et vas attendre la reception d'un message. L'émetteur vas lui faire un appel à la fonction `MPI_Send` qui est aussi bloquante et vas envoyer un message est attendre que la communication sois fini. L'utilisation des fonctions `MPI_Send` / `MPI_Recv` bloque le code ce qui nous fait perdre du temps à attendre. La nome propose aussi une version non bloquante de ces fonctions qui sont `MPI_Isend` et `MPI_Irecv`. Cette version ce contente de posté la communication et rend immédiatement la main. Pour la progression et verifier la terminaison il faut donc d'autre fonctions `MPI_Test` qui vérifie la progression, et la fait si nécessaire, et la fonction `MPI_Wait` qui attend activement la terminaison et s'occupe de la progression si nécessaire. Il est important de noté que le standard ne précise pas si la progression ce fait en tâche de fond ou non, c'est aux implémentations de la nome MPI de choisir. C'est pour cela que la progression peut ce faire au niveau des fonctions `MPI_Wait` et `MPI_Test` ou être fait avant est donc l'appel aux fonctions s'occupe juste de la terminaison. L'envoi des messages peut donc être asynchrone. /\* gardé cette phrase ? \*/

### 2.5.2 Les communications collective

Les communications collective ce fond entre plusieurs processus. Il en existe de différent type :

- un processus vers plusieurs (One-to-All) par exemple un broadcast d'un message
- de plusieurs processus vers un seul (All-to-One) par exemple une reduction (e.g. un processus reçoit la somme des valeurs des autre processus)
- de plusieurs processus vers plusieurs (All-to-All) par exemple quand tous les processus on un message pour les autre

Pour les collective il existe également deux versions, bloquante et non bloquante, qui fonctionne de la même façon que les communications point à point.

### 2.5.3 La progression en tâche de fond

Pour faire progressé les communications en tâche de fond il est possible de :

- faire des appels à `MPI_test` régulièrement et faire du calcule entre chaque appel. Cela nous permet de recouvrir la latence des communications par du calcule. La progression ce faire également dans d'autre

appel aux fonctions MPI. Quand il n'y a plus de calcul à faire on repasse à une progression bloquante par un appel à `MPI_Wait` sauf si la communication est déjà fini.

- utilisé un thread dédié aux progressions. Dans ce cas c'est la bibliothèque MPI qui s'occupe de la progression des communications en tâche de fond grâce à un thread dédié. Il faut donc faire attention au placement des threads et à prendre en compte qu'un thread est déjà utilisé par la bibliothèque de communication. Il faut aussi éviter d'appeler trop souvent `MPI_Test` car ça crée de la contention.
- utilisation des threads de façon opportuniste c'est à dire quand un des thread à fini son calcul il va faire progresser les communications. C'est ce qui est fait par Pioman dans NewMadeleine.

## 2.6 NewMadeleine

*NewMadeleine* est une bibliothèque de communications qui supporte le RPC (Remote procedure call) et implémente aussi une interface MPI. On peut donc la considérer comme une implémentation du standard MPI avec des fonctionnalités supplémentaires. Elle est développée au Centre Inria de l'université de Bordeaux. Elle est basée sur un système de progression événementielle qui lui permet d'être asynchrone. Elle est composée en modules ce qui lui permet de charger dynamiquement des stratégies d'optimisation sur les paquets. Les stratégies sont l'agrégation de paquets, l'utilisation de priorités et le multi-rail /\* TODO : précision multi-rail/split\_balance \*/. Elle possède également un système de drivers pour supporter différents types de communications comme des réseaux (e.g. portals4 pour BXI, ibverbs pour InfiniBand, psm2 pour OmniPath, ofi (Open Fabrics Interfaces), TCP) et localement (shm (mémoire partagée), socket, self).

## 2.7 Les signaux

Les signaux c'est un des mécanismes qui permet de faire l'interface entre un processus et le système. Pour cela le processus utilisateur fait des appels système pour définir pour quels signaux il veut être notifié ou non. Pour être notifié l'utilisateur peut définir au près du système un handler par signal. Ces handlers sont déclenchés par le noyau qui s'occupe de tout (sauvegarde de l'état du processus, changement de contexte...). Les signaux que l'utilisateur peut recevoir correspondent à des exceptions système (e.g. *SIGFPE* pour une division par zéro ou *SIGSEGV* pour un segmentation fault), à des informations (e.g. *SIGTERM* le système demande au processus de se fermer) ou à des notifications de d'autres processus (i.e. *SIGUSR1* et *SIGUSR2*). Il est possible de masquer la réception de signaux grâce à des appels système de masquage.

## 2.8 Travaux antérieur

TODO :

- système de progression dans NewMadeleine
- progression avec pioman
- overlap
- Travaux de Mathieu Barbe durant un stage en 2019 sur l'utilisation d'interruption pour transmettre des événements réseau.
- Ces travaux vis à diminué la latence du au fait de passer par un driver noyaux.
- Il parle dans les perspective de directement traitement des interruptions depuis l'espace utilisateur ce qui mène à ce stage.

TODO : Il est important d'avoir les définitions suivante en tête :

- *CPU* défini la puce en ça totalité.
- *core* ou *processeur* défini un des coeur du *CPU*.
- *core logique*, *processeur logique* ou *unité de calcul* défini une unité de calcul au sein d'un coeur, il y en a deux par core dans les CPUs qui sont à notre disposition.

## 3 Problématiques / Objectifs

### 3.1 Sujet

Le sujet est *Interruptions en espace utilisateur pour le réseau BXI*.

#### 3.1.1 Nouveau mécanisme d'interruption en espace utilisateur

Ce nouveau mécanisme qui permet de dérouler une interruption à partir de l'espace utilisateur et très récent. Il est seulement disponible sur les CPU Intel® Sapphire Rapids qui sont officiellement sortis le 10 janvier 2023. La plupart des CPU ont été disponibles à la vente le 14 mars. AMD n'a pas encore annoncé de support pour les interruption en espace utilisateur.

#### 3.1.2 contentions...

Le fonctionnement actuelle de la progression des communications amène à des problèmes de contention mémoire car plusieurs threads peuvent chercher à lire / modifier la même zone mémoire. Utiliser les interruptions permettrai de ne plus avoir ce problèmes car seul le thread concerné par une zone mémoire serai prévenu.

TODO :

## 3.2 Projet global

Dans la plupart des autres domaines les périphériques envoient une interruption ordinaire à l'application, par le biais des signaux ou d'un appel système bloquant, pour avertir d'un changement. L'idée sera de faire la même chose en HPC grâce à l'interruption en espace utilisateur. Le projet global vise donc à faire progresser les communications entre plusieurs nœuds du réseau BXT sans faire de polling et en utilisant les interruptions en espace utilisateur. Cela permettra de réduire globalement le temps de calcul d'une application. Pour ce faire la carte réseau BXT devra être capable de lever des interruptions en espace utilisateur. Le fait de supprimer le polling et de réduire le temps de calcul permettra de diminuer la consommation électrique.

### 3.2.1 Les objectifs

Il y a plusieurs objectifs, les principaux sont :

- La réduction du temps de calcul.
- utiliser des interruptions en espace utilisateur pour remplacer le polling.
- Simplifier pour l'utilisateur le recouvrement des communications par du calcul pour qu'il n'est plus à ajouter des `MPI_Test` en plein milieu de ces calculs.
- Améliorer la réactivité des communications sans qu'il y ait besoin d'une unité de calcul dédiée à l'attente active.

Ce stage est donc une première étape de ce projet global.

## 3.3 Objectifs du stage

Le premier objectif est de défricher le fonctionnement des interruptions en espace utilisateur à partir des éléments suivants :

- Le manuel Intel® de l'architecture 64 et IA-32 pour les développeurs logiciels
- La présentation du mécanisme de Sohil Mehta, ingénieur Intel® qui a développé le patch noyau, qui est une diapositive associée à une discussion sur LWN.net /\* TODO : cité \*/<sup>1</sup>
- le patch du noyau Linux avec ces manuels. /\* TODO : cité \*/

Le second objectif est de connaître les propriétés du mécanisme et de mesurer sa performance. Le troisième objectif est de ne plus avoir à faire du polling que se soit dédié un thread ou utilisé les threads de façon opportuniste, ne plus perdre du temps de calcul à poller et résoudre les problèmes de réactivité. Pour cela on envisage l'intégration de ces interruptions dans le driver de mémoire

---

1. <https://lwn.net/Articles/869140/> /\* TODO : déplacé \*/

partagé (shm) de NewMadeleine. Pour testé dans un premier temps le fonctionnement avec des communications entre processus (IPC<sup>2</sup>). Pour intégrer les interruption dans les drivers NewMadeleine il faut également permettre la progression des communications à partir d'un handler d'interruption. Le dernier objectif est de montré que l'utilisation d'interruption permet bien d'amélioré le recouvrement des communications par du calcul.

### 3.4 La suite

Les objets suivant du projet global seront traité dans une thèse qui fait suite au stage.

## 4 Exploration de uintr

Pour cette partir j'ai utiliser des connaissance personnel autour du système, du développement noyau, de Linux... et aussi ce que j'ai appris en cours de *programmation système*, de *système d'exploitation* et *architecture des ordinateurs*.

### 4.1 Prérequis est accès

Pour utilisé le mécanisme d'interruption en espace utilisateur il faut donc avoir accès à un CPU Intel® Sapphire Rapids. Comme ils sont sortie très récemment ils sont assez difficile d'accès. Atos nous a, non sans difficulté donné accès, à une machine qui possède 2 CPU Intel® Sapphire Rapids qui sont des Intel® Xeon® Platinum 8470. Ils y a eu plusieurs difficulté pour avoir une machine, pour quelle sois installer et pour quelle sois dans un réseau au quelle on puisse accédé depuis l'Inria. On à donc eu un accès VPN qui utilisé l'ancien système de VPN car le nouveau ne fonctionné pas. Nous avons donc eu accès à la machine environ deux mois et demi après le début du stage. La machine est déjà configuré avec le système d'exploitation Red Hat Enterprise Linux (RHEL) 9.1. Elle possède deux carte BIX v2 que nous n'avons pas utilisé lors du stage.

Il faut aussi avoir une version patché du noyaux linux avec le support du nouveau mécanisme. Cette version patché n'est pas encore disponible dans la branche principal du noyau. Elle est disponible sur le GitHub d'Intel®. Nous avons donc télécharger cette version patché. Nous l'avons compiler et installer sur la machine. Lors de la compilation il faut activé le support des uintr (Voir figure en annexes13). Il est aussi possible d'activé le support qu'un thread bloqué, c'est à dire pas ordonnancer ou dans une appel système interruptible, puisse recevoir une uintr.

---

2. Inter Process Communication en anglais/\* TODO : on garde pour certain acronyme ? \*/

Le mécanisme utilise de nouvelles instructions, il faut donc une version récente du compilateur GCC pour compiler les programmes utilisateurs qui utiliseront les `uintr`. Il faut donc la version **11.3.0** ou plus récente de GCC, sur RHEL il faut la version **12.1.1** ou supérieur. Le support n'est pas encore disponible dans d'autres compilateurs comme LLVM-Clang. Pour compiler un programme utilisateur il faut préciser le flag de compilation `-muintr` pour les fichiers qui définissent un handler d'interruption ou qui utilisent les nouvelles instructions.

## 4.2 Fonctionnement des `uintr`

### 4.2.1 Les interruptions

Pour commencer nous allons voir comment les interruptions matérielles fonctionnent. Nous allons nous concentrer sur l'envoi d'interruption entre deux processus qui sont fixés sur deux unités de calcul. Pour fonctionner les CPU ont une unité dédiée au traitement des interruptions, l'APIC pour Advanced Programmable Interrupt Controller. Cette APIC permet au système d'enregistrer un handler pour chaque interruption. Pour ce faire le système définit un tableau *IDT* (Interrupt Descriptor Table) qui contient 256 entrées. Donc 256 interruptions possibles, les valeurs entre 0 et 255 sont aussi appelées vecteurs. Les vecteurs compris entre 0 et 31 sont réservés pour les exceptions et interruptions système, ceux entre 32 et 127 sont réservés pour les interruptions pour les périphériques, 128 est réservé pour les appels système et entre 129 et 255 pour des utilisations variées.

Il est important de savoir que chaque unité de calcul (processeur logique) possède une *APIC ID* physique. Petit fun-fact le *core ID* est un sous-ensemble de l'*APIC ID*.

Pour déclencher une interruption il y a quatre possibilités :

1. Une exception déclenchée par un processeur (e.g. division par zéro, défaut de segmentation...).
2. Une instruction comme `INT80 numSysCall` pour déclencher un appel système ou bien `INT3` pour définir un point d'arrêt, ou encore `INTO`, `BOUND` et `INT n`.
3. Des pins du CPU dédiés à la réception d'interruption lancés à partir d'un périphérique.
4. Demander à l'APIC en lui-même grâce à un registre *ICR* pour Interrupt Command Register. Il existe un *ICR* par vecteur il faut donc écrire l'*APIC ID* du destinataire dans le *ICR* du vecteur que l'on veut déclencher. Seul le CPU et le noyau peuvent modifier les *ICR*.

On voit bien que les IRQ fonctionnent au niveau du noyau et du CPU.

Nous allons voir un exemple d'envoi d'IRQ entre 2 processus en cours d'exécution. Tout d'abord l'initialisation est faite au démarrage du système et

consiste principalement à définir les handler noyau dans l'*IDT*. Au préalable il faut définir le vecteur utiliser, le handler noyau, la technique pour contacté le système et l'identification du récepteur (e.g. par un patch du noyau, par un module noyau...).

Nous allons maintenant voir les états de l'envoi d'une IRQ montré sur la figure suivante 1 :

- ① Le récepteur fait un appel système pour indiquer au noyau comment il veut être avertie d'une interruption (e.g. un handler utilisateur, un descripteur de fichiers qu'il vas lire, un appel système bloquante, une zone mémoire où écrire...).
- ② l'émetteur peut donc avertir le noyau qu'il faut envoyer une interruption pour cela il peut utiliser un appel système ou écrire dans un descripteur de fichiers...
- ③ Le noyau détermine l'unité de calcul où se trouve le récepteur. Pour cela il peut utiliser par exemple un *PID* (Processus ID) donné par l'émetteur ou autre. Il va donc pouvoir déterminer le *APIC ID* de l'unité de calcul à interrompre.
- ④ Le noyau écrit donc l'*APIC ID* dans le *ICR* d'un vecteur déterminé à l'avance. L'émetteur va reprendre la main après un autre changement de contexte.
- ⑤ L'APIC va donc interrompre le récepteur qui va donc stoppé son exécution et passer dans le noyau. Une fois dans le noyau le handler va se déclencher et exécuté le code prévu au préalable (déclenchement d'un handler utilisateur, écrire dans un descripteur de fichiers, écrire dans une zone mémoire...).

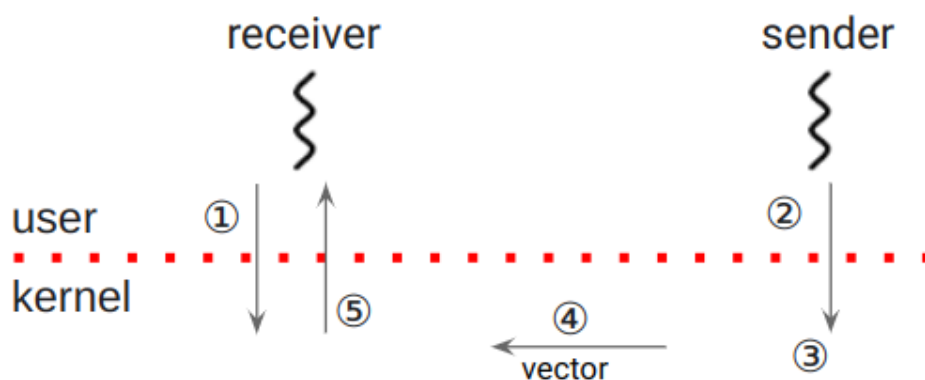


FIGURE 1 – L'envoi d'une interruption

Lors du déclenchement du handler noyau certains registres actuels sont sauvegardés comme le pointeur de pile `RSP`, le registre d'états `RFLAGS`, le

registre **CS** et le registre de pointeur d'instruction **RIP**. Cette sauvegarde ce fait en les empilant dans une nouvelle pile. Le vecteur de l'interruption est aussi empilé en temps que code erreur *errorCode*. Une fois que le handler noyau a fini de s'exécuter il doit faire l'instruction **iret** qui a pour effet de dépiler les registre sauvegardé et de les restorer.

Il est possible de masquer les interruptions grâce à deux instructions utilisable seulement par le noyau qui sont **clui** et **stui**. Ces interruption modifie un flag (*IF*) qui se trouve dans le registre d'états de l'unité de calcul **RFLAGS** (aussi nommé **EFLAGS** sur les architecture 32bits). La liste des instructions pour les IRQ se trouve ici<sup>2</sup>.

Comme on la vue ce mécanisme fonctionne totalement dans le noyau du système. Dans notre exemple il faut au minimum deux changement de contexte (context switch) pour le récepteur et l'émetteur et même peut être plus si le récepteur doit déclencher un handler coté utilisateur.

#### 4.2.2 Les uintr

Le mécanisme d'interruption en espace utilisateur est utilise cinq nouvelle instructions qui sont listé dans le tableau suivant 2. Les deux premiers, **clui** et **stui**, permet le masquage des interruption, en effet comme pour les interruptions ordinaire qui on un flag *IF* pour Interrupt Flag les uintr on un flag *UIF* pour User Interrupt Flag. l'instruction suivant **testui** permet à l'utilisateur de savoir si les interruption sont masquer ou non. Cette instruction existe car l'utilisateur n'a pas accès directement au *UIF* contrairement au interruption ordinaire ou le noyau a lui accès directement au *IF*. L'instruction suivant **uiret** fonctionne comme celle des interruption ordinaire (**iret**) mais pas avec les même registre et surtout elle est utilisable en espace utilisateur. La dernier instruction permet d'envoyer une uintr grâce à un indice que nous allons voir dans cette section 4.2.4.

Interruption	Interruption utilisateur
cli ( <b>CL</b> ear <b>IF</b> )	clui ( <b>CL</b> ear <b>UIF</b> )
sti ( <b>SeT</b> <b>IF</b> )	stui ( <b>SeT</b> <b>UIF</b> )
	testui (Read <b>UIF</b> )
iret (Interrupt <b>RETurn</b> )	uiret (User Interrupt <b>RETurn</b> )
APIC pins, APIC ICR, <b>INT n</b> , <b>INT3</b> , <b>INT0</b> , <b>BOUND</b> et <b>INT80 n</b>	sendipi <uipi_index>

FIGURE 2 – Instructions des interruption et interruption en espace utilisateur

Le mécanisme arrive aussi avec six nouveaux *registres d'états*, des registres *MSR* pour Model-specific registers. Ces registres sont modifier par



le noyau grâce à des appels système que l'utilisateur fait pour initialiser les uintr et utiliser par le CPU. Ils sont décrit dans le tableau suivant<sup>3</sup> et nous expliciterons leur utilité par la suite.

Nom du registre	Description
IA32_UINTR_STACKADJUST	Utilisé par le récepteur pour définir l'adresse de la pile alternative
IA32_UINTR_HANDLER	Utilisé par le récepteur pour définir l'adresse du handler uintr
IA32_UINTR_MISC	Utilisé par l'émetteur pour définir la taille de la <i>UITT</i> et par le récepteur pour que l'APIC connaisse le vecteur d'interruption ordinaire qu'il doit reconnaître pour déclencher le handler uintr et le dernier bit pour le flag de masquage <i>UIF</i>
IA32_UINTR_PD	Utilisé par le récepteur pour définir l'adresse du <i>UPID</i>
IA32_UINTR_RR	Utilisé par l'APIC pour <i>lister</i> les vecteurs uintr qu'il doit envoyer au récepteur <i>UPID</i>
IA32_UINTR_TT	Utilisé par l'émetteur pour définir l'adresse de la <i>UITT</i>

FIGURE 3 – Liste des six registre d'états des uintr

#### 4.2.3 Capacités présente et futur

Le mécanisme a une interface pour l'utilisateur similaire aux signaux 2.7. Nous allons donc voir les capacités des uintr par comparaison à celle des signaux.

Tous d'abord le fonction des uintr ce fait au niveau des threads que les signaux le fonctionnement ce fait au niveau du processus. Il est possible d'avoir un fonctionnement qui ce rapproche d'un fonctionnement par threads avec plusieurs options.

Avec les uintr il est possible de définir une handler différent par threads d'un processus que pour les signaux il est possible de définir un seul handler pour tous les threads d'un processus. Par-contre les signaux permet de définir un handler différent par signal se qui n'est pas possible avec les uintr, il faut le faire nous même en appelant la fonction qui correspond au vecteur reçus.

Il exit 64 signaux possible avec les 32 premiers qui on on signification, pour les uintr il existe aussi 64 vecteurs possible entre 0 et 63 qui n'ont aucune signification par défaut.

Pour les uintr le masquage ce fait via une instruction que pour les signaux il faut faire un appel système.

L'envoi d'un signal ce fait par le noyau suite à une exceptions, une décision du noyau ou la demande d'un processus grâce à un appel système (`kill(signum)` ou `tgkill(signum)`). Pour les uintr l'envoi peut ce faire depuis un autre processus ou depuis le noyau et dans le future pourra ce faire depuis un périphériques.

Avec les signaux le handler peut être déclenché que le processus cible soit endormi ou non pour les uintr c'est différant. Il faut que le thread sois en espace utilisateur pour recevoir une interruption sinon l'interruption sera reçus quand le thread revient en espace utilisateur. On a vue plutôt, dans les prérequis4.1, qu'une fonctionnalité existe à la compilation du noyau pour autorisé l'interruption d'un thread bloqué. Si la fonctionnalité est activé il est donc possible d'interrompre un thread qui n'est pas ordonnancer ou qui est entrain de faire un appel système interruptible et donc le passé en espace utilisateur pour qu'il puisse recevoir l'interruption. Pour utilisé cette fonctionnalité l'utilisateur dois renseigné un flag au moment de définir le handler. Il existe donc trois flags, le premier `UINTR_HANDLER_FLAG_WAITING_ANY` qui active la fonctionnalité et les deux autre qui s'ajoute au précédent et précise si c'est l'émetteur ou le récepteur qui vas prendre le surcoût du passage dans le noyau. Les flags sont `UINTR_HANDLER_FLAG_WAITING_RECEIVER` et `UINTR_HANDLER_FLAG_WAITING_SENDER`.

Pour les signaux le déclenchement du handler est géré par le noyau qui vas sauvegarde de l'états du processus, définir une pile alternative si besoin, changer de context et appelé le handler utilisateur. Pour les uintr le déclenchement du handler est fait par le CPU, il est donc très sommaire :

- changer la pile si une pile alternative est disponible dans le registre `IA32_UINTR_STACKADJUST`
- empiler l'ancien pointeur de pile, le registre d'états de l'unité de calcul, le registre de pointeur d'instruction `RIP` et le vecteur uintr
- aller à l'adresse du handler utilisateur disponible dans le registre `IA32_UINTR_HANDLER`

C'est donc à l'utilisateur qu'appartient la responsabilité de la sauvegarde des registres généraux, des registres vectoriel (SIMD)... et des les restorer à la sortie du handler. Le compilateur permet déjà de faire la sauvegarde des registres généraux avec le flag `general-regs-only` par-contre pour les registres vectoriel il faut les sauvegardé avant de les utilisé. Il faut faire attention aux operations sur les chaîne de caractère de la *libc* car `memcpy`, `memmove`, `memset` et `memcpy` utilise des registres vectoriel par défaut. Le compilateur fournis le flag `-minline-all-stringops` qui permet d'inline<sup>3</sup> ces opérations pour ne plus utiliser de registre vectoriel.

---

3. besoin d'expliqué l'inline ?

Une fois que le handler à fini son execution il faut s'occuper du retour. Pour les signaux c'est le noyau qui s'en occupe et pour les uintr il faut que l'utilisateur utilise l'instruction `uiret`. Donc l'unité de calcul va dépiler le vecteur, les registers qui suivent pour les restaurer ce qui va permettre au code de continuer là où il en était.

Que ce soit dans un handler de signal ou dans un handler d'interruption utilisateur on a la même contrainte, on ne peut pas faire d'attente donc on peut seulement appeler la fonction et appel système dit *async safe*.

#### 4.2.4 Exemple de fonctionnement

Nous allons maintenant voir un exemple d'initialisation des uintr montré sur la figure suivante 4 :

- ① Le récepteur enregistre au près du noyau un handler d'interruption qu'il a défini avec l'appel système `uintr_register_handler(ui_handler)`. Le noyau va enregistrer ce handler dans le registre `IA32_UINTR_HANDLER` et va initialiser une zone mémoire nommée *UPID* pour User Posted Interrupt Descriptor. Ce *UPID* permet au mécanisme des uintr de manipuler des informations propres à ce thread qui sont essentielles pour l'envoi d'uintr. Il est enregistré dans le registre `IA32_UINTR_PD`.
- ② Le récepteur donne au noyau un vecteur uintr entre 0 et 63 qu'il veut recevoir, 8 dans la figure, avec l'appel système `uvec_fd <- uintr_vector_fd(8)`. Le noyau lui retourne un descripteur de fichier qui pointe vers une structure qui possède à la fois le vecteur et l'adresse du *UPID*.
- ③ Le récepteur envoie ce descripteur de fichier aux émetteurs potentiels, un seul dans notre cas. Nous verrons comment partager ce descripteur de fichier dans une section dédiée 4.2.5.
- ④ L'émetteur va s'enregistrer au près du noyau grâce au descripteur de fichier, avec l'appel système `uipi_index <- uintr_register_sender(uvec_fd)`. Pour ce faire le noyau possède un tableau *UITT* pour User Interrupt Target Table qui fait une taille de 256 entrées par défaut. L'adresse de ce tableau doit être enregistrée dans le registre `IA32_UINTR_TT` avec cette taille dans les quatre premiers octets du registre `IA32_UINTR_MISC`, on peut donc faire varier la taille de ce tableau. Chaque entrée de ce tableau consiste en une zone mémoire nommée *UITTE* pour User Interrupt Target Table Entry. Le noyau va donc trouver une entrée libre dans le *UITT* et renseigner l'*UITTE* avec le vecteur uintr et l'adresse du *UPID* obtenu grâce au descripteur de fichier. Il va aussi mettre un vecteur d'interruption ordinaire dans le cinquième octet du registre `IA32_UINTR_MISC`, ce vecteur "ordinaire" est une nouvelle *IRQ* dédiée pour les uintr et a pour valeur 236. Pour finir il retourne l'indice de l'entrée à l'émetteur.

- ⑤ L'utilisateur peut maintenant envoyer autant d'interruption totalement depuis l'espace utilisateur.

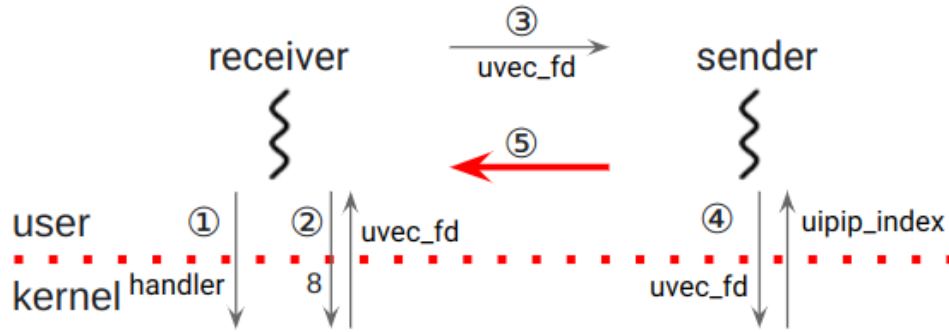


FIGURE 4 – Phase d'initialisation des uintr

Maintenant que nous avons vu l'initialisation nous allons pouvoir voir comment l'envoi d'interruption se fait avec la figure suivante 5 :

- ① L'émetteur va utiliser l'instruction `senduipi` avec l'indice récupéré au moment de l'initialisation. L'utilité de calcul va donc pouvoir récupérer l'adresse de l'*UPID* et le vecteur uintr à envoyer du récepteur grâce au tableau *UITT* qui se trouve dans le registre `IA32_UINTR_TT` et grâce à l'indice donné à l'instruction. Il va aussi pouvoir vérifier que l'indice se trouve bien dans le tableau, avec `IA32_UINTR_MISC`, et que l'entrée est valide. Dans la zone mémoire de l'*UPID* il va pouvoir écrire le vecteur uintr à envoyer et il va déterminer si il faut envoyer une interruption ordinaire. En effet les interruptions en espace utilisateur utilisent les interruptions ordinaires. L'envoi de celle-ci dépend de si une interruption ordinaire n'a pas déjà été envoyée. Dans notre cas on va considérer que c'est le premier envoi.
- ② Donc l'émetteur va récupérer le vecteur d'interruption ordinaire dans le registre `IA32_UINTR_MISC` et sélectionner le *ICR* correspondant au vecteur ordinaire. Puis il va récupérer l'*APIC ID* dans l'*UPID* et l'écrire dans le registre *ICR*.
- ③ L'*APIC* va réceptionner l'interruption, elle va comparer le vecteur d'interruption ordinaire reçu avec celui qui se trouve dans l'*UPID* qu'il connaît grâce au registre `IA32_UINTR_PD`. Si ils sont identiques l'*APIC* va pouvoir déclencher le système de traitement des uintr sinon elle va utiliser le mécanisme habituel des *IRQ*.
- ④ Le système de traitement des uintr va donc indiquer dans l'*UPID* que l'interruption ordinaire a déjà été envoyée puis va commencer le déclenchement des handlers pour tous les vecteurs uintr reçus en partant du plus grand, 63, au plus petit, 0. Dans notre exemple on en a un seul qui

est 8. L'*APIC* vas donc ordonné à l'unité de calcul de changer de pile si alternative est disponible dans le registre `IA32_UINTR_STACKADJUST`, empiler les registres nécessaire, empiler le vecteur uintr (donc 8 dans l'exemple) et aller à l'adresse du handler qui est disponible dans le registre `IA32_UINTR_HANDLER`.

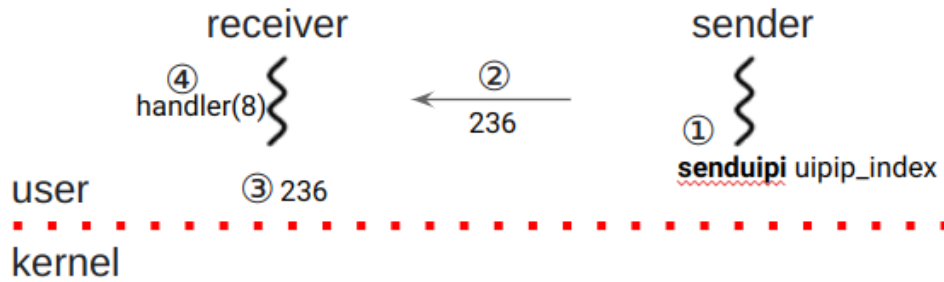


FIGURE 5 – L'envoi d'une uintr

Le fait d'utiliser le mécanisme habituelle des *IRQ* peut être dans le cas ou l'uintr et déclencher depuis le noyau ou en destination du thread non ordonnancer ou dans un appel système interruptible.

#### 4.2.5 Partage du descripteur de fichier

Pour partager le descripteur de fichier il y a plusieurs façon possible.

Dans nos testes du mécanisme entre processus nous avons utilisé l'héritage des descripteurs de fichier. Dans nos testes du mécanisme entre threads nous avons utiliser un appel système qui permet d'enregistré tous les threads du processus comme émetteur sans utiliser le descripteur de fichier, mais il est possible d'utiliser une variable global pour y mettre le descripteur de fichier.

Dans le cas ou les deux processus sont independent on peut utiliser l'appel système `pidfd_getfd` qui permet de dupliqué le descripteur de fichier d'un autre processus si il a le même propriétaire et si on connais sont *PID* et le numéro du descripteur de fichier. Nous avons donc testé en partageant le numéro de descripteur de fichier avec un pipe. On aurai aussi pu passer par un fichier ou par sockets.

Par la suite, dans *NewMadeleine* nous avons utiliser le système d'URL à la connection qu'on verra dans la section `/* TODO : ref */`.

### 4.3 Testes du mécanisme

Nous avons donc fait des testes du mécanisme avec des exemple minimaux de communications entre processus, nous allons voir les plus pertinent dans cette section.

Tout d'abord des tests pour mesurer le temps d'envoi d'interruption en espace utilisateur entre deux threads et deux processus.

Pour le teste d'envoi entre deux threads il commence par créer un nouveau thread qui vas commencer par se "bind" à une unité de calcul, nous le verrons en détail dans la section 4.5. Il vas ensuite enregistrer un handler, démasquer les interruption, enregistrer tous les threads du processus comme émetteur avec l'appel système `uipi_index <- uintr_register_self(vector)` (`uipi_index` est global au processus) et attendre grâce à une boucle. Pendant ce temps là le thread principal attend une second, ce temps est arbitraire est laisse le temps au thread d'enregistrer un handler. Après sont attente il vas se "bind" à une unité de calcul puis envoyer une interruption. Pour cela nous commençons par enregistre le temps processeur actuelle puis on utilise l'instruction `senduipi` avec variable global `uipi_index`. Une fois que le handler ce déclenche on enregistre le temps actuelle du processeur pour ensuite calculer la différence. Ce teste est capable de faire plusieurs fois cette mesure. Il fini par imprimer les différence de temps dans la console et par désallouer et de-enregistré les uintr et les autre structure.

Pour le teste d'envoi entre deux processus il commence par créer un pipe pour l'envoi du descripteur de fichier et une zone de mémoire partagé pour stocker les mesures de temps. Puis le processus ce fork en deux, le premier devient l'émetteur et le second de récepteur. Comme pour la version thread ils se "bind" à une unité de calcul. Le récepteur enregistre un handler et récupère un descripteur de fichier avec l'appel système `uvec_fd <- uintr_vector_fd(vector)` qu'il envois dans le pipe avant d'attendre grâce à une boucle. L'émetteur reçoit le numéro du descripteur de fichier, il connais déjà le pid du processus grâce au fork, il peut donc utilisé `pidfd_getfd` pour dupliqué le descripteur de fichier. Avec ce descripteur il s'enregistre en temps qu'émetteur d'uintr. La mesure du temps et l'envoi d'interruption fonctionne comment pour la version thread. On peut aussi faire plusieurs fois la mesure et on imprime et effectue la terminaison proprement.

On verra le résultats de c'est mesure dans la section 4.6.

Un teste ou le thread s'auto interromps tous simple en faisant un `uipi_index <- uintr_register` puis un `senduipi uipi_index` et on fait la mesure de la même façon que pour les autre testes.

Pour les testes avec la pile alternative on à un test très simple qui est basé sur celui qui s'auto interromps et on modifie les tests d'envois entre deux processus ou threads pour mesurer les performances. Il est bien sur possible d'activé ou non l'utilisation de la pile alternative. Pour définir cette nouvelle pille on le faut juste après avoir enregistrer le handler et avant le démasquage.

Un teste de démasquer les uintr dans un handler, c'est tous à fait possible et ça pose des problématique similaire que pour les signaux.

Un teste est l'envoi de plusieurs interruptions d'affilée. Il y à une différence de comportement par rapport au signaux, quand on reçoit une inter-

ruption et que le handler est déclenché le comportement est le même c'est à dire que les interruptions vont s'écrouler et le handler se déclenchera à nouveau une fois. Là où se trouve la différence c'est si on fait plusieurs interruptions avant que le handler se déclenche, les interruptions s'écroulent à ce moment là aussi donc le handler se déclenche qu'une seule fois quand on fait deux interruptions successives qu'avec les signaux le handler de signal se déclenchera deux fois pour deux émissions de signal successive car le handler est déjà déclenché dès le premier signal.

Grâce à un test nous avons vu qu'actuellement on peut enregistrer plusieurs fois le même descripteur de fichier ce qui mène à des doublons dans le tableau *UITT* avec plusieurs *UITTE* pour le même couple vecteur / *UPID*. Cette limitation de l'implémentation noyau est documentée dans le patch et accompagné d'un commentaire *TODO*.

#### 4.4 Correction d'un bug dans le patch noyau

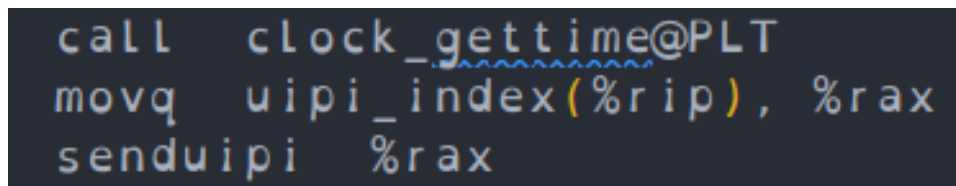
En manipulant le mécanisme nous sommes tombés sur un bug qui concerne l'utilisation d'une pile alternative. Comme pour les signaux il est possible de définir une pile alternative, cette nouvelle pile est utilisée au moment où le handler est déclenché. L'interface pour définir la pile alternative est la même pour les signaux et les uintr. Dans les manuels il est bien indiqué que l'utilisateur doit lui-même allouer une zone mémoire consacrée à la nouvelle pile, il a également la responsabilité de libérer la mémoire une fois le handler dé-enregistré. Il est bien indiqué que l'utilisateur doit donner l'adresse de début (*base address*) de la zone mémoire en plus la taille à l'appel système qui définit la pile alternative. Pour les uintr l'appel système est `uintr_alt_stack(spAddress, size)`. Il faut noter qu'une pile empile les éléments vers le haut c'est à dire que l'adresse du pointeur de pile décroît à l'ajout d'un élément. Donc pour utiliser la zone mémoire dédiée à la pile à partir de la fin. Du côté du noyau le mécanisme des signaux garde l'adresse et la taille en mémoire pour le moment où le handler doit être déclenché. Le calcul du pointeur de pile de la nouvelle pile se fait donc juste avant de déclencher le handler. Pour les uintr le noyau se contente juste d'enregistrer l'adresse dans le registre dédié `IA32_UINTR_STACKADJUST` et le processeur utilise l'adresse telle que comme pointeur de pile. On est donc confronté à un bug de débordement mémoire car on part du début de la zone mémoire. Il y a bien un test dans le patch du noyau qui vérifie ce cas mais mal. Nous sommes tombés sur ce bug au moment d'utiliser le uintr dans *NewMadeleine* qui manipule bien plus la mémoire qu'un simple test, on se retrouve avec des problèmes de corruption mémoire, des défauts de segmentation et des "double free detected". Nous avons donc corrigé le test du patch noyau ainsi que l'appel système. Pour ce fait on ajoute la taille de la zone mémoire à l'adresse avant de l'écrire dans le registre. Nous avons donc fait une *Pull request* sur le dépôt GitHub du patch, à l'heure où nous écrivons ce document

il n'a toujours pas été appliqué par Intel®.

## 4.5 Mesure de la latence

Pour mesurer la latence entre le moment où on envoie une interruption et où l'interruption est reçue par le handler on fait 2 mesures de temps. Pour faire les mesures de temps on utilise `clock_gettime` qui utilise l'instruction `rdtsc` et retourne le temps actuel du processeur. On fait une première mesure juste avant d'envoyer une interruption et une seconde au tout début du handler. Pour obtenir la latence on a juste à soustraire la première mesure à la seconde.

Nous avons regardé le code assembleur pour s'assurer de la mesure. Pour l'envoi, que l'on peut voir sur la figure 6, on peut voir que entre la mesure est l'instruction d'envoi il y a seulement une Lecture mémoire qui n'est pas très coûteuse. Du côté de la réception, que l'on peut voir sur la figure 7, on voit la sauvegarde des registres généraux au début du handler. Cette sauvegarde ajoute un petit surcoût mais il est obligatoire. Le compilateur *GCC* nous force à mettre le flag `general-regs-only` pour compiler un handler d'interruption et donc sauvegarder les registres généraux. On peut voir la déclaration d'un handler avec la mesure du temps sur la figure 8.

The image shows three lines of assembly code on a dark background with syntax highlighting. The first line is 'call clock\_gettime@PLT', the second is 'movq uipi\_index(%rip), %rax', and the third is 'senduipi %rax'.

```
call    clock_gettime@PLT
movq    uipi_index(%rip), %rax
senduipi %rax
```

FIGURE 6 – Code assembleur de l'envoi d'untr



```

handler:
.LFB203:
    .cfi_startproc
    .cfi_def_cfa_offset 16
    pushq %r11
    .cfi_def_cfa_offset 24
    .cfi_offset 11, -24
    pushq %r10
    .cfi_def_cfa_offset 32
    .cfi_offset 10, -32
    pushq %r9
    .cfi_def_cfa_offset 40
    .cfi_offset 9, -40
    pushq %r8
    .cfi_def_cfa_offset 48
    .cfi_offset 8, -48
    pushq %rdi
    .cfi_def_cfa_offset 56
    .cfi_offset 5, -56
    movl $1, %edi
    pushq %rsi
    .cfi_def_cfa_offset 64
    .cfi_offset 4, -64
    leaq ts2(%rip), %rsi
    pushq %rcx
    .cfi_def_cfa_offset 72
    .cfi_offset 2, -72
    pushq %rdx
    .cfi_def_cfa_offset 80
    .cfi_offset 1, -80
    pushq %rax
    .cfi_def_cfa_offset 88
    .cfi_offset 0, -88
    subq $8, %rsp
    .cfi_def_cfa_offset 96
    cld
    call clock_gettime@PLT

```

FIGURE 7 – Code assembleur de l'handler uintr

```

__attribute__((target("general-regs-only")))
__attribute__((interrupt))
void handler(struct __uintr_frame* ui_frame, u64 vector) {
    clock_gettime(CLOCK_MONOTONIC, &ts2);
}

```

FIGURE 8 – Déclaration du handler uintr

Nous faisons la mesure de la même façon pour les signaux.

Pour ne pas perturber la mesure il faut "bind" les threads à une unité de calcul. "bind" consiste à demander au noyau de toujours ordonnancer le thread sur la même unité de calcul. Pour ce faire nous utilisons la bibliothèque *hwloc*. Il est donc important de "bind" les threads pendant les mesures car dans le cas contraire le noyau vas changer le thread d'unité de calcul ce qui vas amené à un surcoût non négligeable. En effet le changement d'unité et un peut coûteux et invalide certain cache.

"bind" les threads nous permet aussi de contrôler le placement de ceci, c'est à dire si on met le thread émetteur et le thread récepteur sur deux unité de calcul proche ou distante. Comme nous le verrons dans la section 4.6 le placement a un impact sur la latence des uintr.

Il est aussi important de fixer la fréquence de tous les core du CPU pour avoir des mesure courante.

## 4.6 Performances

Dans cette section nous allons voir des mesure de la latence des uintr dans différent contexts. La fréquence des unités de calcul est fixer à 2GHz lors des mesure. Les mesure faite avec le turbo boost activé monte à 3.8GHz. Les mesures de latence sont en nanoseconde. Il est intéressant de noter que sur une unité de calcul cadencer à une fréquence de 2GHz elle execute environ deux instructions par nanoseconde. Certaine mesure, que se sois avec les uintr ou les signaux, sont très élevé mais sont très peut nombreuse, elle sont certainement lié au système. Dans les graphiques que nous allons voir nous avons donc coupé les valeurs qui dépasse les 8000 nanoseconde. Nous avons défini trois placements à partir le la topologie de la machine fourni par Atos (vous pouvez trouver la topologie sur cette figure14). Les trois placements sont les suivant :

- le placements "proche" qui consiste à placer les threads sur deux unité de calcul proche mais pas dans le même core.
- le placements "éloigné" qui consiste à placer les threads sur deux unité de calcul qui sont dans le même CPU et qui sont éloigné.
- le placements "très éloigné" qui consiste à placer les threads sur deux unité de calcul qui ce trouve sur deux CPU différent.

Les mesures que nous allons présenter on étai faite sans que la pile alternative ne sois activé. Nous avons bien fait des mesures avec la pile alternative des uintr activé est nous n'avons vue aucune différence car l'utilisation de cette pile amené seulement à une copie de la mémoire d'un registre à un autre si le registre contiens une adresse ce qui est très peut coûteux.

Dans nos graphiques nous appelons une mesure une interaction. Nous faisons donc un millions d'itérations et nous n'affichons pas la premier car cette itération est énormément perturber notamment par le coût de chargement des caches.

Dans les graphiques nous avons représenté la mesure de la latence par des points bleu pour les signaux et par des points rouge pour les uintr. Nous ne cherchons pas à expliquer les mesure des signaux, elle sont juste là pour comparer les uintr avec un mécanisme qui passe par le noyau.

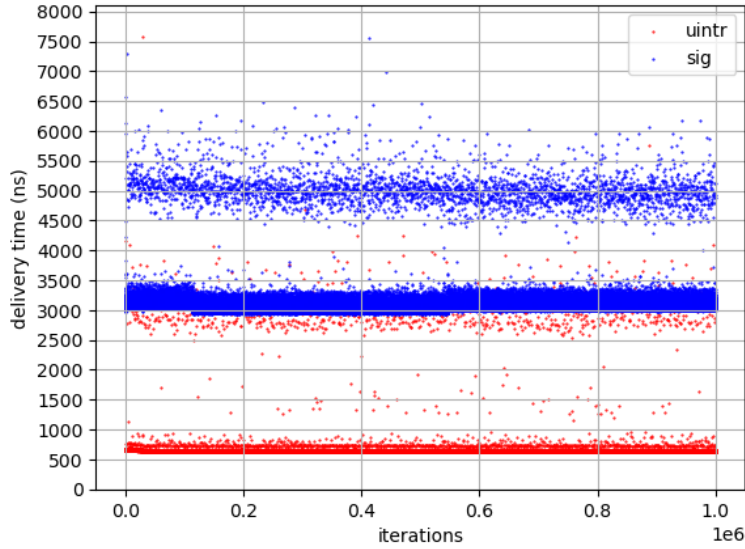
Sur la graphiques 9a, les itérations sont représenté sur l'axe des abscisses et la latence sur l'axe des ordonnés. Bien sur plus la latence est basse mieux

c'est. Nous observons que le mécanisme a une latence de environ 642 nanoseconde ce qui est environ 4.75 fois plus rapide que les signaux. On voit qu'il y a deux groupe de mesure :

- un au alentours des 642 nanoseconde qui comporte la majorité des mesures. On le voit bien dans le tableau 9b qui se trouve juste en dessous du graphique. En effet entre la mesure minimum et la mesure à 95% il y a une différence de 19 nanoseconde. On peut voir cette distribution aussi dans l'histogramme sur cette figure 10. Cette histogramme porte sur des plage de 300 nanoseconde. On voit bien pour les uintr, en orange, que la majorité se trouve sur la bande entre 300 et 600 nanoseconde.
- un autre vers 2700 nanoseconde qui correspond certainement au moment où l'interruptions n'as pas pu être reçus car le thread été dans le noyau.

Pour un mécanisme qui fonctionne au niveau des instructions on pourrait s'attendre à une latence moins grande mais le mécanisme est bien plus rapide que le fait de passer par le système.

999999 iterations 2 threads with core binding and without the first iteration (Next to)



(a)

	mean	std	min	10%	50%	95%	max
<b>sig</b>	3065	160	2920	3001	3054	3157	68532
<b>uintr</b>	643	90	629	638	642	648	65973

(b)

FIGURE 9 – Mesures de latence entre deux threads avec un placement proche

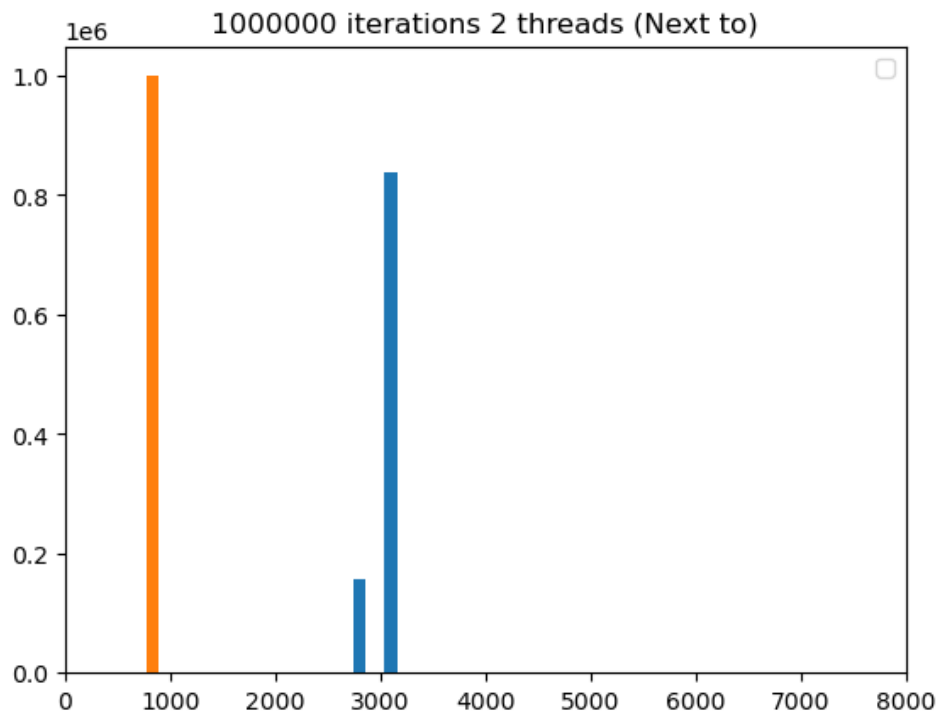


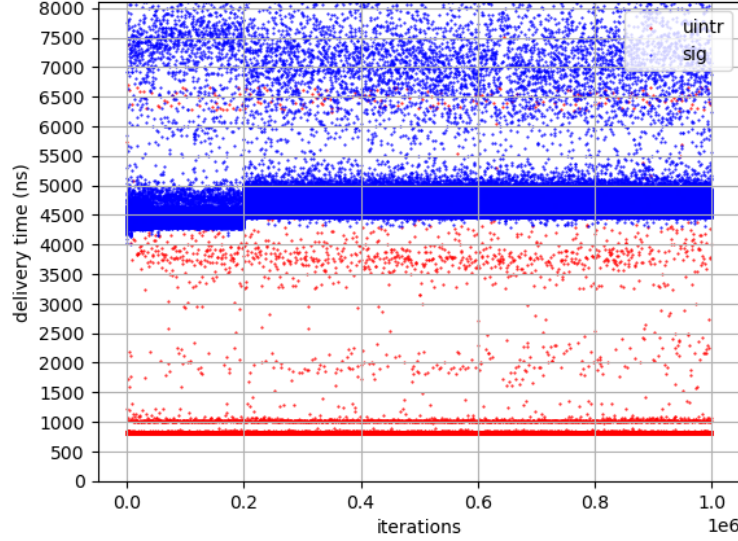
FIGURE 10 – Histogramme de distribution des mesures de la latence entre deux threads proche

Nous avons fait les même mesure entre deux processus est les valeur sont très similaire peut importe le placement car le mécanisme fonctionne au niveau des threads. On peut retrouvé le ces mesure en annexe sur la figure 15.

En comparant les mesures de latences entre le placement proche et éloigné on vois une petite différence de 10 nanoseconde de plus, on peut voir ça sur la figure 16.

Quand on compare entre le placement proche et très éloigné on vois une grand différence qui est du notamment au fait de passer d'un noeud mémoire NUMA à un autre. On à donc une différence d'environ 172 nanoseconde de plus. Le graphiques nous montre également une plus grande dispersion de la latence avec la majorité qui est toujours en dessous de 1000 nanoseconde. On retrouve ces mesures sur la figure 11 juste en dessous.

999999 iterations 2 threads with core binding and without the first iteration (Very Far)



(a)

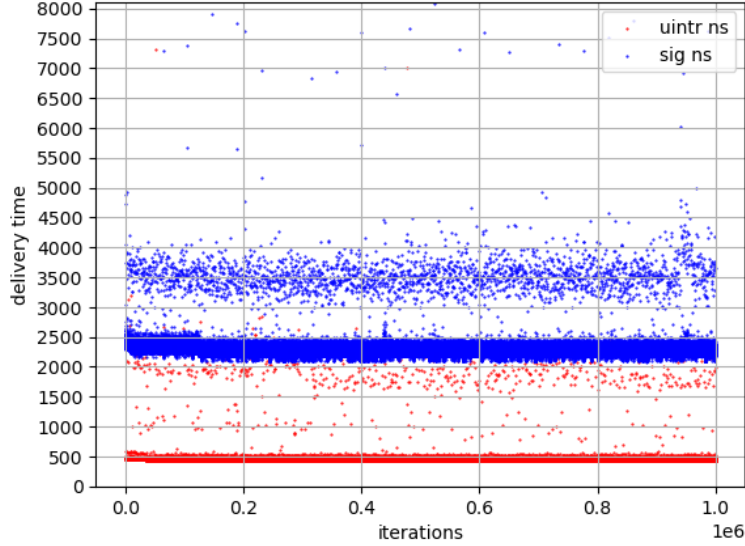
	mean	std	min	10%	50%	95%	max
<b>sig</b>	4601	403	4009	4396	4561	4796	58200
<b>uintr</b>	818	136	801	810	814	820	58072

(b)

FIGURE 11 – Mesures de latence entre deux threads avec un placement très éloigné

Quand on augmente la fréquence des unités de calcul la latence diminue. Pour ce faire on active le turbo boost du CPU. On le voit sur les mesure de la figure 12 pour le placement proche mais c'est également le cas pour l'éloigné et le très éloigné en figure 17 et figure 18.

999999 iterations 2 threads with core binding and without the first iteration (Next to)



(a)

	mean	std	min	10%	50%	95%	max
<b>sig</b>	2262	294	2081	2216	2251	2357	283171
<b>uintr</b>	442	313	426	437	440	450	311486

(b)

FIGURE 12 – Mesures de latence entre deux threads avec un placement proche et le turbo boost activé

Nous n'avons fait aucune mesures de latence pour les threads bloqué et interruptible.

## 5 Intégration dans NewMadeleine

### 5.1 Présenté NewMadeleine details

- liste de progression recv
- liste de progression send
- p\_pw
- post
- poll
- driver
- nm\_schedule (appelé par nm\_wait, ...)
- pioman
- existe : progression du nm\_schedule ou de pioman vers le core\_task

## 5.2 Modification

TODO : on désactive le poll avec les driver qui on des handler. Mais on fait toujours un poll qui fonctionne du premier coup.

- ajout d'un driver sig\_shm
- ajout d'un driver uintr\_shm
- progression à partir du handler du driver vers les core\_task
- problématiques de gestion des interruptions
  - on ne peut pas faire d'attente dans les handler Les fonction doivent être async safe
  - on ne peut pas utiliser d'allocateur donc il faut utiliser des p\_pw déjà alloué
  - pour avoir un p\_pw disponible il faut faire progresser le communications
  - la progresser à une partie critique qui naissaisite un verrou
  - si on ne peut pas récupéré le core\_lock (try\_lock) il faut mettre à plus tard le trétement de l'interruption
  - on utilise donc une file lock-free
  - problématique des liste lock-free qui doivent être wait-free
    - le principe d'une lfqueue est d'attente si quelqu'un d'autre modifie la file
    - il faut donc une file wait-free
    - j'ai fait de la biblio
    - décrire les différante solutions
    - solution d'Alexandre
- gestion des multiple interruptions qui s'écrase, (prob\_any / pour les large si la progression à traiter le pipeline courant)
- quand est ce qu'on envois des interruption ?
  - au moment ou on poste le premier paquet d'envois, pour indiquer au recepteur que des données sont disponible (le recepteur à toujours un paquet de reception posté)
  - au moment ou une progression est fait du coté du recepteur, pour indiquer une reception a l'émetteur
  - l'émetteur reçoit une iterruption est détermine si l'émission est fini, petit paquet, ou si il faut envoyer la suite, paquet large.

## 5.3 Suite de testes

Tous les testes ne passe pas ?

## **5.4 Performances**

### **5.4.1 Résultats avec attente active**

TODO : voire le sur coup des interruption

### **5.4.2 Résultats recouvrement communication par du calcule**

TODO : courbe overlap reception

## **6 Bilan**

## **7 Remerciements**



## 8 Annexes

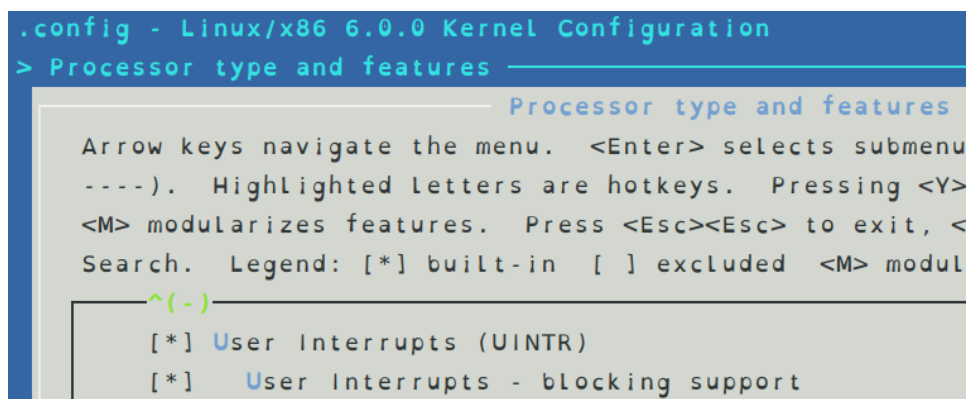


FIGURE 13 – Activé le support des uintr à la compilation du noyau

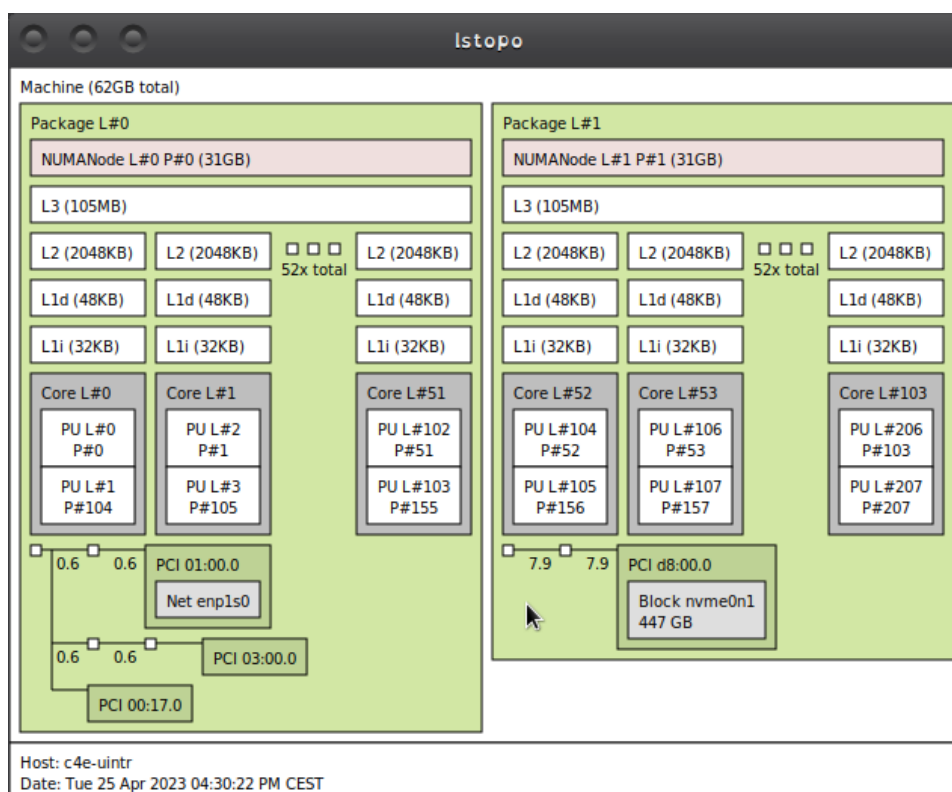
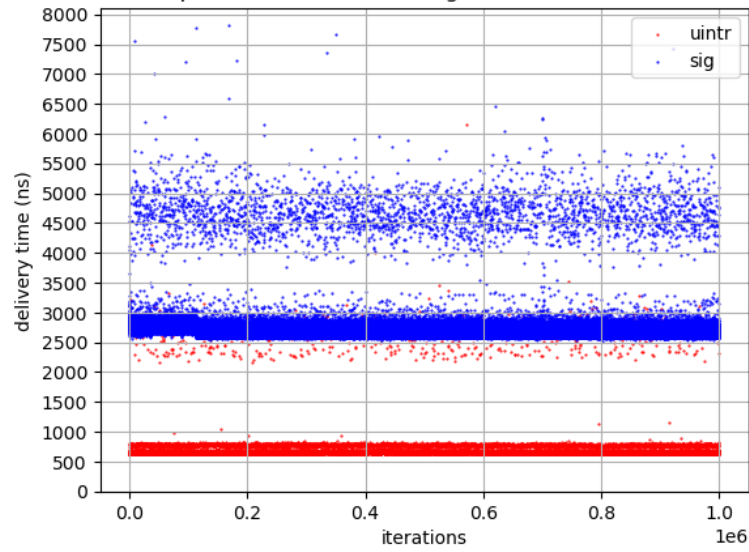


FIGURE 14 – Topologie de la machine fourni par *Atos* obtenus grâce à la command `lstopo`

999999 iterations 2 process with core binding and without the first iteration (Next to)



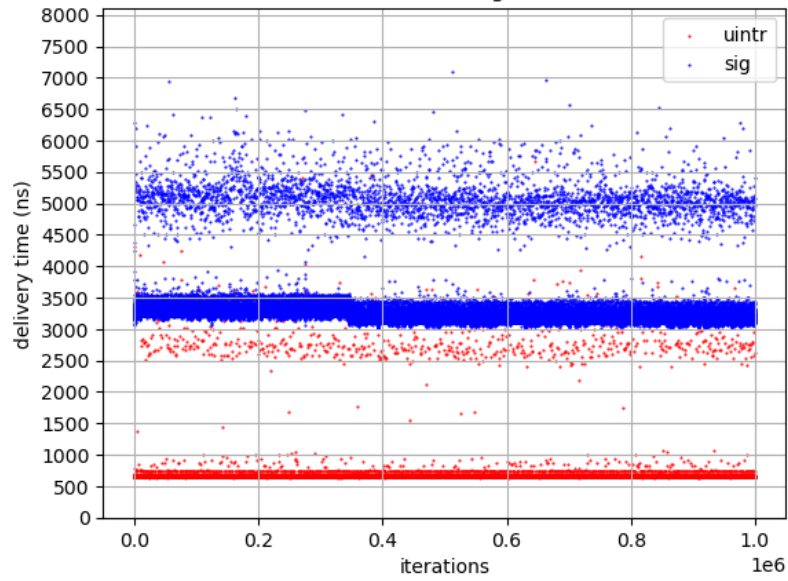
(a)

	mean	std	min	10%	50%	95%	max
<b>sig</b>	2675	147	2520	2620	2658	2781	44311
<b>uintr</b>	651	72	631	642	649	659	46053

(b)

FIGURE 15 – Mesures de latence entre deux process avec un placement proche

999999 iterations 2 threads with core binding and without the first iteration (Far)



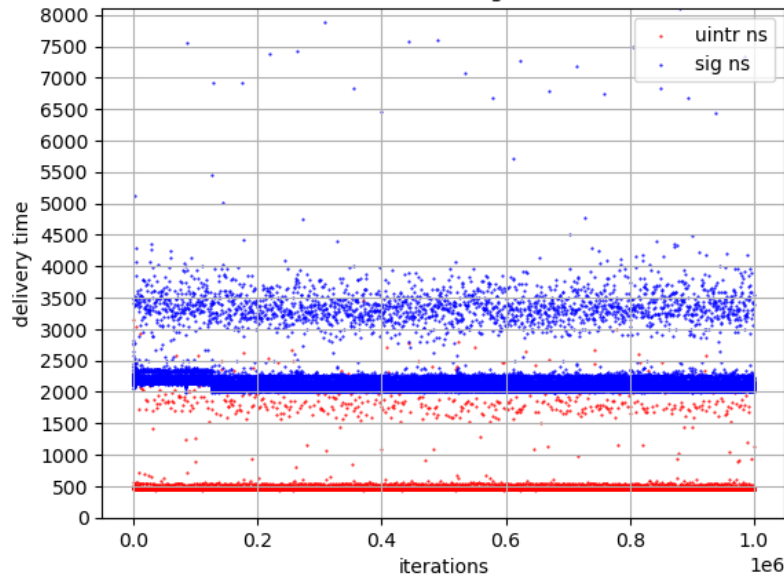
(a)

	mean	std	min	10%	50%	95%	max
<b>sig</b>	3210	156	3014	3141	3177	3302	63690
<b>uintr</b>	654	329	638	648	653	659	325408

(b)

FIGURE 16 – Mesures de latence entre deux threads avec un placement éloigné

999999 iterations 2 threads with core binding and without the first iteration (Far)



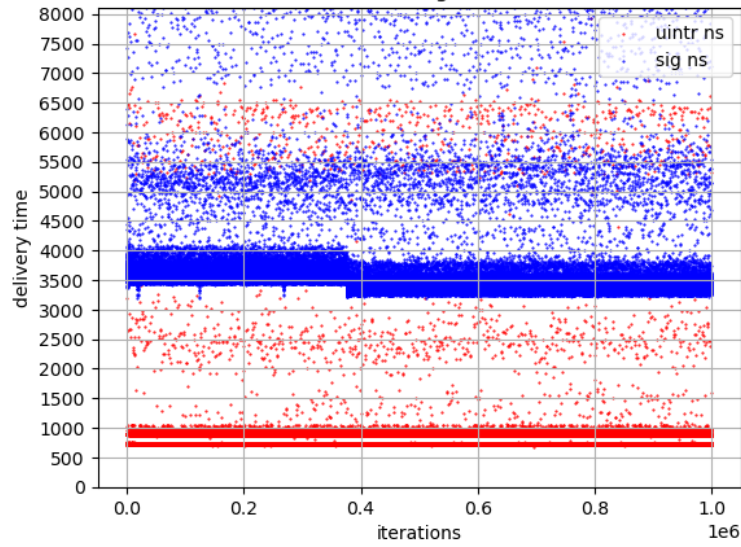
(a)

	mean	std	min	10%	50%	95%	max
<b>sig</b>	2097	282	1981	2064	2082	2178	270625
<b>uintr</b>	455	389	440	450	454	460	388815

(b)

FIGURE 17 – Mesures de latence entre deux threads avec un placement éloigné et le turbo boost activé

999999 iterations 2 threads with core binding and without the first iteration (Very Far)



(a)

	mean	std	min	10%	50%	95%	max
<b>sig</b>	3520	327	3189	3383	3481	3678	51798
<b>uintr</b>	736	159	683	721	725	735	46787

(b)

FIGURE 18 – Mesures de latence entre deux threads avec un placement très éloigné et le turbo boost activé