



Mémoire de stage Master 2

Stage du 1^{er} 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)

31 juillet 2023

Table des matières

1	Introduction	2
1.1	Présentation Inria	2
1.2	Présentation Atos (Eviden)	2
1.3	Environnement de travail	3
1.4	Le cadre	3
1.5	Présentation du plan	4
2	Contexte	4
2.1	HPC	4
2.2	OS bypass	6
2.3	Les interruptions matérielles	7
2.4	Polling	7
2.4.1	Inconvénients du polling	8
2.5	BXI	8
2.6	MPI	9
2.6.1	Les communications point à point	9
2.6.2	Les communications collective	10
2.6.3	La progression en tâche de fond	10
2.7	NewMadeleine	11
2.8	Les signaux	11
2.9	Travaux antérieur	12
2.10	Définitions	12
3	Problématiques / Objectifs	12
3.1	Sujet	12
3.1.1	Nouveau mécanisme d'interruption en espace utilisateur	13
3.1.2	contentions...	13
3.2	Projet global	13
3.2.1	Les objectifs	14
3.3	Objectifs du stage	14
3.4	La suite	15
4	Exploration de uintr	15
4.1	Prérequis est accès	15
4.2	Fonctionnement des uintr	16
4.2.1	Les interruptions	16
4.2.2	Les uintr	19
4.2.3	Capacités présente et futur	20
4.2.4	Exemple de fonctionnement	22
4.2.5	Partage du descripteur de fichier	25

4.3	Testes du mécanisme	26
4.4	Correction d'un bug dans le patch noyau	27
4.5	Mesure de la latence	28
4.6	Performances	31
5	Intégration dans NewMadeleine	36
5.1	Présenté NewMadeleine details	36
5.2	Modification	37
5.3	Suite de testes	38
5.4	Performances	38
5.4.1	Résultats avec attente active	38
5.4.2	Résultats recouvrement communication par du calcule	38
6	Bilan	39
7	Remerciements	39
8	Annexes	40

1 Introduction

J'ai effectué mon stage de fin de master de six mois dans l'équipe-projet TADaaM du Centre Inria de l'université de Bordeaux. Le sujet du stage a 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 commun avec de grandes universités.

1.2 Présentation Atos (Eviden)

Atos est l'un des leaders internationaux de la transformation numérique. Elle couvre un large éventail d'activités, notamment le cloud, la cybersécurité, les services transactionnels, le conseil, l'infogérance, le Big Data, les supercalculateurs, etc. Atos compte 112 000 collaborateurs. Actuellement, Atos est en restructuration afin de se séparer en deux entités. L'entité qui nous intéresse pour ce stage est Eviden, qui englobe notamment les activités liées aux supercalculateurs et au HPC. Cette restructuration est récente, donc lorsque nous évoquons Atos dans ce document, nous parlons de la partie Eviden.

Atos est le seul grand constructeur européen de supercalculateurs. Elle est donc le leader européen et est bien positionnée mondialement. Elle est présente dans le secteur du HPC depuis l'acquisition de la société Bull en 2014. Bull était déjà impliquée dans le HPC depuis les années 2000 et en avait fait son cœur de métier.

La machine Leonardo en Italie, construite par Atos, se classe quatrième au dernier Top500. Ce classement évalue les machines les plus puissantes pour le HPC et est publié deux fois par an.

1.3 Environnement de travail

Mon environnement de travail se trouve dans les locaux du Centre Inria de l'université de Bordeaux. Un bureau dans l'open space de l'équipe-projet TADaaM a été mis à ma disposition. Je peux participer et assister à des activités scientifiques intéressantes, telles que des séminaires, des soutenances de thèse, une soutenance HDR et d'autres activités diverses. Nous avons également accès aux salles de visioconférence, notamment pour les réunions de suivi de stage hebdomadaires. En plus, il y a un Baby-foot, une cafétéria, une petite médiathèque, etc.

1.4 Le cadre

Du côté de l'équipe TADaaM d'Inria, l'objectif de l'équipe est de mener des recherches 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, d'ingénieurs de recherche, de post-doc, de doctorants et de stagiaires. Sa culture informatique est l'utilisation d'environnements Linux, de logiciels open source, de clusters de calcul HPC, le traitement des données et les systèmes. L'équipe a mis à ma disposition un ordinateur portable avec une station de travail reliée à un écran, à Internet, à un clavier et une souris. Inria donne également l'accès à un ensemble de services, notamment une boîte e-mail, un service de communication (Mattermost), un service de visio (Webex) et un intranet avec le menu de la cafétéria, entre autres. Nous sommes libres d'installer le système d'exploitation et d'utiliser les outils informatiques de notre choix.

Du côté d'Atos l'équipe BXI-LL (BXI Low Level) a pour objectif de fournir le support logiciel bas niveau pour les cartes réseaux BXI. Cela comprend le développement des pilotes de

la carte, le support Lustre (un système de fichiers parallèle et distribué utilisé pour l'I/O, qui ne fait pas partie de ce stage), ainsi qu'une implémentation de l'interface logicielle *Portal4*. *Portal4* est une interface logicielle qui permet une abstraction des opérations possibles sur les réseaux HPC. Son but est de faciliter l'utilisation de ces opérations réseau avec les meilleures performances possibles pour les implémentations MPI. /* je suis pas sur de ma présentation de Portal4 */ L'implémentation d'Atos permet aux implémentations MPI d'utiliser le réseau BXI. Le personnel de l'équipe est composé d'ingénieurs, de doctorants et de stagiaires. La culture informatique de l'équipe est l'utilisation de 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 commencerons par une présentation du contexte dans lequel le stage se place. Ensuite, nous aborderons les objectifs à long terme ainsi que les objectifs du stage. Nous continuerons en décrivant le nouveau mécanisme d'interruption en espace utilisateur et ce que nous en avons fait. Ensuite, nous verrons l'intégration de ce mécanisme dans la bibliothèque de communication NewMadeleine, ainsi que les tests que nous avons effectués. Pour finir, nous ferons un bilan du stage et évoquerons les travaux qui suivront.

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'intelligences artificielles. Ces simulations simulent la dynamique des fluides, la résistance structurelle, les interactions moléculaires, les flux d'air...

Elles couvrent différents domaines :

- L'industrie : médical, automobile, aviation, construction, aérospatiale...
- La défense : simulation atomique...
- La recherche scientifique : création de galaxies, fusion nucléaire, climat...
- La météo

De nos jours les supercalculateurs sont composés de plusieurs ordinateurs que l'on appelle nœuds de calculs. Ceci est regroupé en grappe (cluster), il est tout vu et utilisé comme une seule grande machine. Ce fonctionnement pose des questions sur la répartition du calcul, la distribution de la mémoire, la communication entre les différents nœuds... Dans le contexte du stage, nous nous sommes concentrés sur les communications entre nœuds. Les nœuds sont connectés entre eux par un réseau haute performance. Ce réseau est dédié aux communications et n'est pas forcément de type Ethernet. La gestion des nœuds est généralement effectuée par un second réseau plus traditionnel (Ethernet ; TCP/IP). Les réseaux haute performance ont une latence autour de quelques microsecondes. /* TODO : insister sur la différence entre les réseaux pour le HPC et les réseaux classiques */ Chaque nœud possède une ou plusieurs cartes réseau et il faut les programmer.

De nos jours, les supercalculateurs sont composés de plusieurs ordinateurs que l'on appelle nœuds de calcul. Ceux-ci sont regroupés en grappes (clusters) et sont tous vus et utilisés comme une seule grande machine. Ce fonctionnement soulève des questions concernant la répartition du calcul, la distribution de la mémoire et la communication entre les différents nœuds. Dans le contexte du stage, nous nous sommes concentrés sur les communications entre nœuds. Les nœuds sont connectés entre eux par un réseau haute performance dédié aux communications, qui n'est pas forcément de type Ethernet. La gestion des nœuds est généralement effectuée par un second réseau plus traditionnel (Ethernet ; TCP/IP). Les réseaux

haute performance ont une latence de l'ordre de quelques microsecondes ce qui n'est pas le cas des réseaux classiques qui ont une plus grande latence. Avec ce type de réseaux, le débit est aussi bien plus élevé. Chaque noeud possède une ou plusieurs cartes réseau et il faut les programmer.

2.2 OS bypass

Le stage se passe donc aussi dans un contexte système. Chaque noeud a son propre système d'exploitation qui permet la gestion des ressources, des processus, des fichiers, des périphériques. Pour cela, le système a 2 espaces :

- Un espace noyau (kernel en anglais) où seul le code du système peut s'exécuter. Le code du système peut donc modifier n'importe quel endroit de la mémoire, exécuter n'importe quelle instruction...
- Un espace utilisateur où le code de l'utilisateur est exécuté. Cet espace est limité par le noyau qui contrôle ce que fait l'utilisateur.

En temps normal, les périphériques sont programmés directement depuis le noyau du système d'exploitation, cela est fait pour des raisons de sécurité et de standardisation des accès. Cependant, lorsque l'on passe par le noyau, il y a un surcoût qui n'est pas négligeable dans notre cas. En effet, pour passer par le noyau, on utilise généralement des appels système qui prennent la forme d'une fonction. Un appel système effectue un changement de contexte (context switch) pour passer de l'espace utilisateur à l'espace noyau. Ce changement de contexte est coûteux car il sauvegarde les états du code de l'utilisateur avant d'exécuter celui du noyau. Une fois le changement de contexte effectué, le code noyau de l'appel système s'exécute avant de refaire un changement de contexte pour, cette fois, passer de l'espace noyau à l'espace utilisateur et ainsi restaurer l'état du code de l'utilisateur. Donc, le fait de passer par un appel système coûte plusieurs microsecondes. On voit donc qu'il n'est pas préférable d'utiliser les appels système, car leur durée est du même ordre de grandeur qu'une communication entre noeuds. 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 registres de la carte réseau dans l'espace d'adressage virtuel du processus utilisateur.

Pour transmettre des événements à l'utilisateur, les périphériques utilisent généralement les interruptions ordinaires, qui passent par le système donc on ne les utilise très peu en HPC. En HPC on privilégie donc le polling que nous verrons en se 2.4.

2.3 Les interruptions matérielles

Les interruptions matérielles, également appelées IRQ pour "Interrupt ReQuest", existent depuis longtemps et servent à remonter des événements extérieurs, par exemple d'un clavier, d'un port de communication, d'un port IDE, etc. Elles ont ensuite été utilisées pour signaler des événements en provenance de périphériques plus variés, comme des cartes réseau. Dans ce document, nous utiliserons le terme "interruption ordinaire" ou "IRQ" pour les désigner.

2.4 Polling

Il faut donc, peu importe la technique, régulièrement scruter pour faire progresser les communications.

Le *polling* consiste à scruter (poll) régulièrement si un événement a été reçu. Concrètement, cela consiste à lire une zone mémoire modifiée par la carte réseau et vérifier si un bit est passé à 1. Pour cela, il est possible de dédier un thread qui effectuera une attente active, scrutant sans cesse si un événement a été reçu. Cependant, cette technique entraîne la perte d'une unité de calcul quand le thread est ordonnancé, ce qui réduit la puissance de calcul, donc elle est utilisée dans certains cas comme expliqué dans cet article [?]. Une autre approche consiste à entrelacer le code de l'utilisateur avec les scrutations, ce qui est fréquemment utilisé, mais cela oblige l'utilisateur à prendre en compte la progression. Une troisième solution, utilisée par *Pioman* dans *NewMadeleine* (nous en parlerons plus tard), consiste à effectuer ces scrutations de ma-

nière opportuniste dans les threads qui ont fini leur calcul, mais pour cela, il faut déjà utiliser plusieurs threads et avoir une application avec des calculs de durée hétérogène.

2.4.1 Inconvénients du polling

Dans le cas d'un thread dédié qui effectue une attente active, la réactivité est excellente, sauf lorsque le nombre de threads dépasse le nombre d'unités de calcul et lorsque le thread n'est pas ordonnancé.

Dans le cas où l'on entrelace le calcul et les scrutations, la réactivité est moins bonne car il faut attendre que le calcul soit terminé pour effectuer un poll. C'est ce qui est habituellement fait.

Dans le cas où l'on utilise les threads de façon opportuniste pour effectuer des scrutations, la réactivité est moins bonne car il faut qu'il y ait un thread disponible.

Un choix doit donc être fait entre perdre de la capacité de calcul ou perdre en réactivité. De plus, un peu de temps de calcul est perdu à effectuer du polling.

2.5 BXI

BXI pour "Bull eXascale Interconnect" est un type de réseau d'interconnexion (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 cartes réseau BXI et de switches BXI. Les cartes BXI sont capables de faire progresser les communications réseau sans aucune intervention du CPU (offload des communications réseau). Le CPU a juste à soumettre une commande dans une file sur la carte, et elle s'occupe de tout. Le CPU peut ensuite récupérer une file d'événements pour savoir ce qu'il s'est passé, en somme faire un poll. Les cartes sont également capables de déclencher des interruptions. Les cartes implémentent donc l'API de communication *Portal4*. Les utilisateurs utilisent donc cette API pour envoyer et recevoir des paquets réseau.

2.6 MPI

MPI, pour "Message Passing Interface", est un standard permettant de fournir une interface pour effectuer des communications entre plusieurs processus, souvent situés sur des noeuds différents, que l'on appelle *processus MPI*. Cette interface fournit une abstraction pour transmettre des données entre ces différents processus, masquant ainsi la complexité des communications. L'interface permet d'envoyer et de recevoir des messages. Pour cela, il existe deux modes de communications :

2.6.1 Les communications point à point

C'est-à-dire entre deux processus MPI, également appelé One-to-One. Pour ce faire, le processus MPI récepteur va appeler la fonction `MPI_Recv` qui est bloquante et va attendre la réception d'un message. L'émetteur va lui faire un appel à la fonction `MPI_Send` qui est également bloquante et va envoyer un message, puis attendre que la communication soit terminée. L'utilisation des fonctions `MPI_Send` / `MPI_Recv` bloque le code, ce qui nous fait perdre du temps à attendre. Le standard MPI propose également une version non bloquante de ces fonctions, qui sont `MPI_Isend` et `MPI_Irecv`. Cette version se contente de poster la communication et rend immédiatement la main. Pour la progression et vérifier la terminaison, il faut donc utiliser d'autres fonctions comme `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 noter que le standard MPI ne précise pas si la progression se fait en tâche de fond ou non, c'est aux implémentations de la norme MPI de choisir. C'est pour cela que la progression peut se faire au niveau des fonctions `MPI_Wait` et `MPI_Test`, ou être faite avant, et donc l'appel aux fonctions s'occupe juste de la terminaison. L'envoi des messages peut donc être asynchrone. /* gardé cette phrase ? */

2.6.2 Les communications collective

Les communications collectives se font entre plusieurs processus. Il en existe différents types :

- 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 réduction (e.g. un processus reçoit la somme des valeurs des autres processus).
- De plusieurs processus vers plusieurs (All-to-All), par exemple lorsque tous les processus ont un message pour les autres.

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

2.6.3 La progression en tâche de fond

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

- Faire régulièrement des appels à `MPI_test` et effectuer des calculs entre chaque appel. Cela permet de recouvrir la latence des communications par du calcul. La progression peut également se faire dans d'autres appels aux fonctions MPI. Lorsqu'il n'y a plus de calcul à effectuer, on repasse à une progression bloquante par un appel à `MPI_wait`, sauf si la communication est déjà terminée.
- Utiliser 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 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 cela peut créer de la contention.
- Utilisation des threads de façon opportuniste, c'est-à-dire qu'un des threads, une fois son calcul terminé, fera progresser les communications. C'est ce qui est fait par *Pio-man* dans *NewMadeleine*.

2.7 NewMadeleine

NewMadeleine est une bibliothèque de communications qui prend en charge le RPC pour "Remote Procedure Call" et implémente également 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, ce qui lui permet d'être asynchrone. Elle est composée de modules, ce qui lui permet de charger dynamiquement des stratégies d'optimisation sur les paquets. Ces 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, tels que des réseaux (e.g. portals4 pour BXL, ibverbs pour InfiniBand, psm2 pour OmniPath, ofi pour "Open Fabrics Interfaces", TCP) et des communications locales (shm pour "mémoire partagée", socket, self).

2.8 Les signaux

Les signaux sont l'un des mécanismes permettant l'interface entre un processus et le système. Pour cela, le processus utilisateur effectue des appels système pour définir les signaux pour lesquels il souhaite être notifié. Pour être notifié, l'utilisateur peut définir un handler pour chaque signal auprès du système. Ces handler sont déclenchés par le noyau qui prend en charge toutes les opérations nécessaires (sauvegarde de l'état du processus, changement de contexte, etc.).

Les signaux que l'utilisateur peut recevoir correspondent à des exceptions système (e.g. *SIGFPE* pour une division par zéro ou *SIGSEGV* pour une erreur de segmentation), à des informations (e.g. *SIGTERM* pour demander au processus de se fermer) ou à des notifications d'autres processus (i.e. *SIGUSR1* et *SIGUSR2*). Il est possible de masquer la réception de certains signaux en utilisant des appels système de masquage.

2.9 Travaux antérieur

TODO : Amélioré ?

Les origines de *NewMadeleine* sont décrites dans cet article [?]. Le système de progression dans *NewMadeleine* fait l'objet de plusieurs travaux. Le fonctionnement du système de progression opportuniste *Pioman* est décrit dans cet article[?]. Pour l'impact de l'overlap et la manière de le mesurer, l'article [?] explique tout en détail. Concernant l'utilisation des interruptions pour transmettre des événements réseau, les travaux de Mathieu Barbe pendant son stage en 2019 sont disponibles ici [?]. Ces travaux visent à réduire la latence due au passage par un driver noyau. Ils abordent également les perspectives de traitement direct des interruptions depuis l'espace utilisateur, ce qui a conduit à ce stage.

2.10 Définitions

Il est important d'avoir les définitions suivantes en tête :

- *CPU* désigne la puce dans sa totalité.
- *core* ou *processeur* désigne l'un des coeur du *CPU*.
- *core logique*, *processeur logique* ou *unité de calcul* désigne une unité de calcul au sein d'un coeur. Il y en a deux par core dans les *CPUs Intel® Sapphire Rapids* fourni par Atos.

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 permettra de ne plus avoir ce problèmes car seul le thread concerné par une zone mémoire sera prévenu.

TODO :

3.2 Projet global

Dans la plupart des autre domain les périphériques envoi une interruption ordinaire à l'application, par le bais des signaux ou d'un appel système bloquante, pour avertir d'un changement. L'idée serai de faire la même chose en HPC grâce au interruption en espace utilisateur. Le projet globale vise donc à faire progresser les communications entre plusieurs noeuds du réseau BXI 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 BXI devra être capable de levé des interruptions en espace utilisateur. Le fait de supprimer le polling et de réduire le temps de calcul permettra de diminué 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 pleins milieu de ces calcul.
- Améliorer la réactivité des communications sans qu'il y est besoin d'une unité de calcul dédié 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 des discussions sur LWN.net /* TODO : cité */¹
- 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 poll et résoudre les problèmes de réactivité. Pour cela on envisage l'intégration de ces interruptions dans le driver de mémoire partagée (shm) de NewMadeleine. Pour tester dans un premier temps le fonctionnement avec des communications entre processus (IPC²). Pour intégrer les inter-

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

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

ruption dans les drivers NewMadeleine il faut également permettre la progression des communications à partir d'un handler d'interruption. Le dernier objectif est de montrer que l'utilisation d'interruption permet bien d'améliorer 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 soit installer et pour quelle soit dans un réseau au quelle on puisse accéder 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 téléchargé cette version patchée. Nous l'avons compilé et installé 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.

Le mécanisme utilise de nouvelle instructions, il faut donc une version récente du compilateur GCC pour compiler les programme utilisateur 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'autre compilateur comme LLVM-Clang. Pour compiler un programme utilisateur il faut précisé le flag de compilation `-muintr` pour les fichiers qui définisse un handler d'interruption ou qui utilise les nouvelles instructions.

4.2 Fonctionnement des uintr

4.2.1 Les interruptions

Pour commencer nous allons voir comment les interruption matérielle fonctionnent. Nous allons nous concentrer sur l'envoi d'interruption entre deux processus qui sont fixer sur deux unité de calcul. Pour fonctionné les CPU on une unité dédié au traitement des interruptions, l'APIC pour Advanced Programmable Interrupt Controller. Cette APIC permet au système d'enregistré un handler pour chaque interruptions. Pour ce faire le système définit un tableau *IDT* (Interrupt Descriptor Table) qui contiens 256 entrées. Donc 256 interruptions possible, les valeurs entre 0 et 255 sont aussi appelé vecteur. Les vecteur compris entre 0 et 31 sont réservé pour les exceptions et interruption système, ce entre 32 et 127 sont réservé pour les interruptions pour les périphériques, 128 est réservé pour les appel système et entre 129 et 255 pour des utilisation varier.

Il est important de savoir que chaque unité de calcul (processeur logique) possède un *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 exceptions déclencher par un processeur (e.g. division par zero, 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 `INT0`, `BOUND` et `INT n`.
3. Des pins du CPU dédié à la réception d'interruption lancer 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 vois bien que les IRQ fonctionne au niveau du noyau et du CPU.

Nous allons voir un exemple d'envoi d'IRQ entre 2 processus en cours d'exécution. Tous d'abord l'initialisation ce fait 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'envois 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 ou ce trouve le récepteur. Pour cela il peut utiliser par exemple un `PID` (Processus ID) donné par l'émetteur ou autre. Il vas donc pouvoir déterminer le `APIC ID` de 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...).

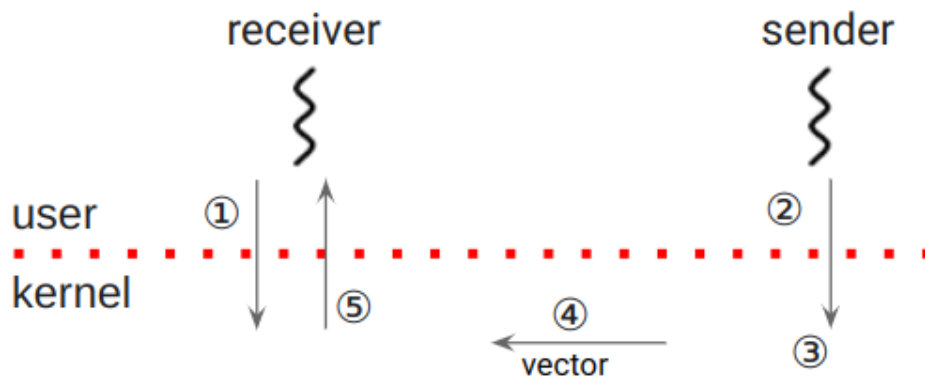


Figure 1 – L'envoi d'une interruption

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 se 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 registres sauvegardés et de les restaurer.

Il est possible de masquer les interruptions grâce à deux instructions utilisables seulement par le noyau qui sont `cli` et `sti`. Ces interruptions modifient un flag (`IF`) qui se trouve dans le registre d'états de l'unité de calcul `RFLAGS` (auss appelé `EFLAGS` sur les architectures 32bits). La liste des instructions pour les IRQ se trouve ici².

Comme on le voit 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 déclenché 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
<code>cli (CLear IF)</code>	<code>clui (CLear UIF)</code>
<code>sti (SeT IF)</code>	<code>stui (SeT UIF)</code>
	<code>testui (Read UIF)</code>
<code>iret (Interrupt RETurn)</code>	<code>uiret (User Interrupt RETurn)</code>
APIC pins, APIC ICR, <code>INT n</code> , <code>INT3</code> , <code>INT0</code> , <code>BOUND</code> et <code>INT80 n</code>	<code>sendipi <uipi_index></code>

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 suivant3 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
IA32_UINTR_HANDLER	Utilisé par le récepteur pour définir l'adresse du handler
IA32_UINTR_MISC	Utilisé par l'émetteur pour définir la taille de la UITT et par le récepteur pour que l'APIC connaisse le vecteur d'interruption ordinaire qu'il doit reconnaître pour déclencher le handler uintr et
IA32_UINTR_PD	Utilisé par le récepteur pour définir le couplet pour le tag de destination de l'UPID
IA32_UINTR_RR	Utilisé par l'APIC pour lister les vecteurs uintr qu'il doit envoyer au récepteur UPID
IA32_UINTR_TT	Utilisé par l'émetteur pour définir l'adresse de la UITT

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 2.8. 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 différant 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 différant 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 exception, 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 différent. 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_WA`

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 contexte 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 opérations sur les chaîne de caractère de la `libc` car `memcpy`, `memmove`, `memset` et `memcmp` utilise des registres vectoriel par défaut. Le compilateur fournis le flag `-minline-all-stringops` qui permet d'inline³ ces opérations pour ne plus utiliser de registre vectoriel.

Une fois que le handler à fini sont 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 vas dépiler le vecteur, les registers qui suivent pour les restorer ce qui vas permettre au code de continuer là où il en été.

Que ce sois dans un handler de signal ou dans un handler d'interruption utilisateur on a la même contrainte, on ne peu pas faire d'attente donc on peut seulement appeler le 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 prés du noyau un handler d'interruption qu'il a défini avec l'appel système `uintr_register_handler(u)`. Le noyau vas enregistré ce handler dans le registre `IA32_UINTR_HANDLER` et vas initialiser une zone mémoire nommé `UPID` pour User Posted Interrupt Descriptor. Ce `UPID` permet au mécanisme des uintr de manipuler des informations propre à ce thread qui sont essentielle 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

3. besoin d'expliqué l'inline ?

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 potentielle, un seul dans notre cas. Nous verrons comment partagé 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 `uip_index <- uintr_register`. Pour ce faire le noyau possède un tableau *UITT* pour User Interrupt Target Table qui fait une taille de 256 entré par défaut. L'adresse de ce tableau doit être enregistré dans le registre `IA32_UINTR_TT` avec ça taille dans les quatre premier octets du registre `IA32_UINTR_MISC`, on peut donc faire varier la taille de ce tableau. Chaque entrées de ce tableau consiste en une zone mémoire nommé *UITTE* pour User Interrupt Target Table Entry. Le noyau va donc trouvé une entré libre dans le *UITT* et renseigné 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* dédié pour les `uintr` et à pour valeur 236. Pour finir il retourne l'indice de l'entré à l'émetteur.
- ⑤ L'utilisateur peut maintenant envoyer autant d'interruption totalement depuis l'espace utilisateur.

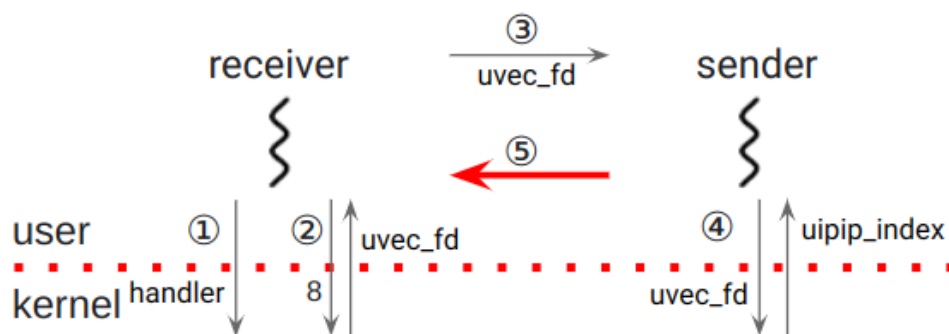


Figure 4 – Phase d'initialisation des `uintr`

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 `sendupi` 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 va donc ordonner à l'unité de calcul de changer de pile si alternative est disponible dans le registre `IA32_UINTR_STACKADJUST`, empiler les registres nécessaires, 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`.

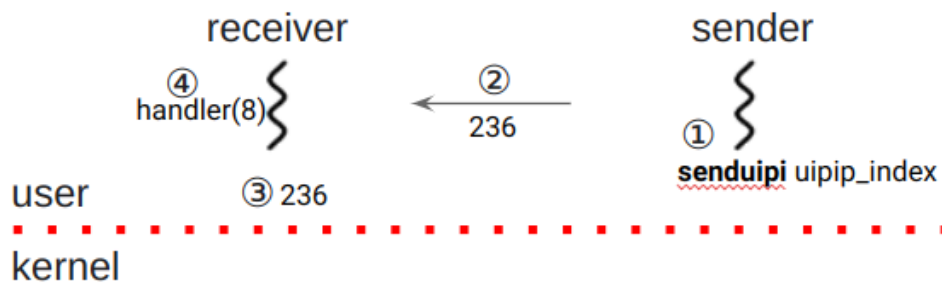


Figure 5 – L’envoi d’une uintr

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 seconde, 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 connaît 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 me-

sure 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 interrompt tous simple en faisant un `uipi_index <- uintr_register_self(vector)` 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 interrompt et on modifie les tests d'envois entre deux processus ou threads pour mesurer les performances. Il est bien sur possible d'activer ou non l'utilisation de la pile alternative. Pour définir cette nouvelle pile on le fait 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 interruption 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 ou 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 successive qu'avec les signaux le handler de signal se déclenchera deux fois pour deux émission de signal successive car le handler est déjà déclenché dès le premier signal.

Grâce à un teste 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 documenté 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 tombé sur un bug qui concerne l'utilisation d'une pile alternative. Comme pour les signaux il est possible de définir une pile alternative, cette

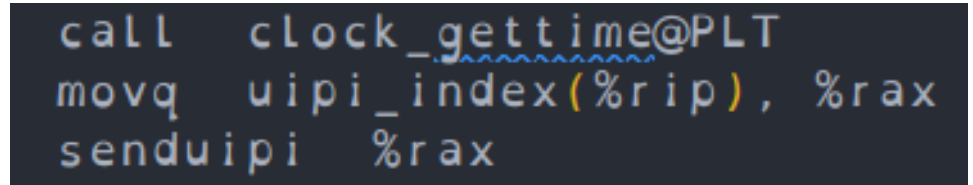
nouvelle pile est utiliser au moment où le handler et déclencher. L'interface pour définir la pile alternative est la même pour les signaux et les uintr. Dans les manuels il est bien indiquer que l'utilisateur doit lui même allouer une zone mémoire consacré à la nouvelle pile, il a également la responsabilité de libérer la mémoire une fois le handler dé-enregistrer. Il est bien indiquer que l'utilisateur doit donné l'adresse de début (base address) de la zone mémoire en plus la taille à l'appel système qui défini la pile alternative. Pour les uintr l'appel système est `uintr_alt_stack(spAddress, size)`. Il faut noté qu'une pile empile les éléments vers le haut c'est à dire que l'adresse du pointeur de pile décrois à l'ajout d'un éléments. Donc pour utiliser la zone mémoire dédié à la pile à partir de la fin. Du coté du noyau le mécanisme des signaux garde l'adresse et la taille en mémoire pour le moment ou le handler doit être déclencher. Le calcul du pointeur de pile de la nouvelle pile ce 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 tel quel 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 somme tombé sur ce bug au moment d'utiliser le uintr dans *NewMadeleine* qui manipulent bien plus la mémoire qu'un simple teste, on ce retrouvé avec des problèmes de corruption mémoire, des défauts de segmentation et des "double free detected". Nous avons donc corriger le teste 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'heurs ou nous écrivons ce document il n'a toujours pas étai appliqué par Intel®.

4.5 Mesure de la latence

Pour mesurer la latence entre le moment où on envois une interruption et où l'interruption est reçus par le handler on fait 2 mesure de temps. Pour faire les mesure de temps on utilise `clock_gettime` qui utilise l'introduction `rdtsc` et retourne le temps actuelle du processeur. On fait une premier

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 a snippet of assembly code on a dark background with syntax highlighting. The code consists of three lines: 'call clock_gettime@PLT', 'movq uipi_index(%rip), %rax', and 'senduipi %rax'. The first line has a blue squiggly underline under 'clock_gettime@PLT'.

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

Figure 6 – Code assembleur de l'envoi d'unintr

```

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 va changer le thread d'unité de cal-

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

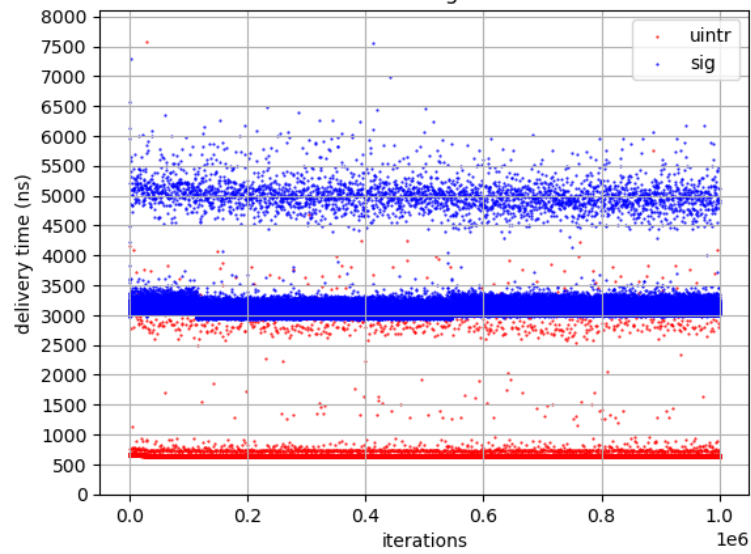
Dans les graphiques nous avons représenté la mesure de la latence par des points bleu pour les signaux et par des points rouge pour les uintr. Nous ne cherchons pas à expliquer les mesure des signaux, elle sont juste là pour comparer les uintr avec un mécanisme qui passe par le noyau.

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 vois qu'il y a deux groupe de mesure :

- un au alentours des 642 nanoseconde qui comporte la majorité des mesures. On le vois bien dans le tableau 9b qui ce trouve juste en dessous du graphiques. 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 vois bien pour les uintr, en orange, que la majorité ce trouve sur la bande entre 300 et 600 nanoseconde.
- un autre vers 2700 nanoseconde qui correspond certainement au moment ou 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 pourrais s'attendre à une latence moins grande mais le mécanisme est bien plus rapide que le fait de passé 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
sig	3065	160	2920	3001	3054	3157	68532
uintr	643	90	629	638	642	648	65973

(b)

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

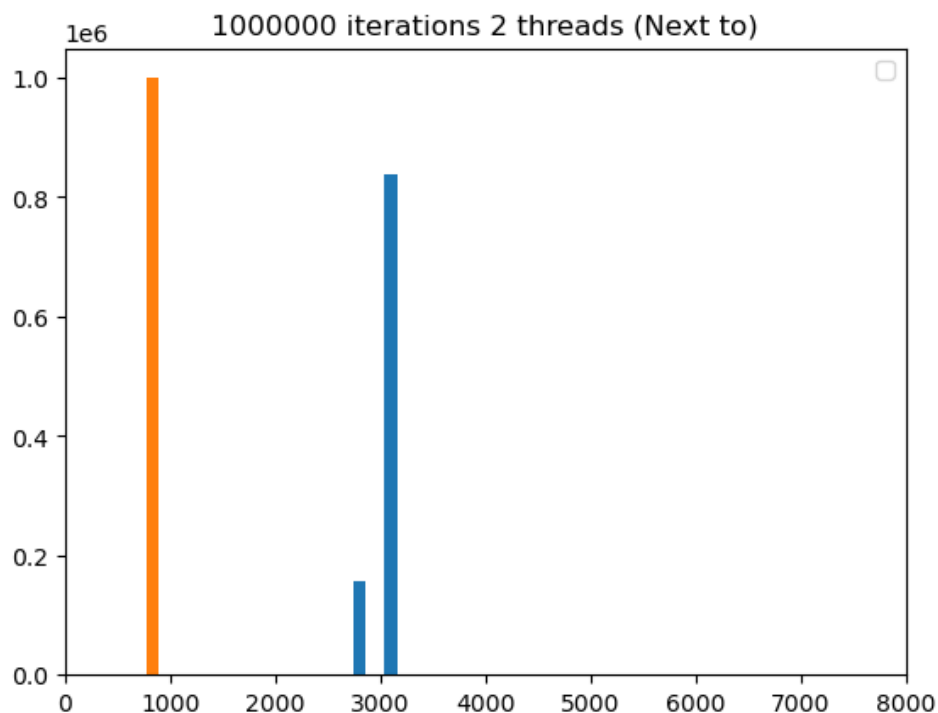


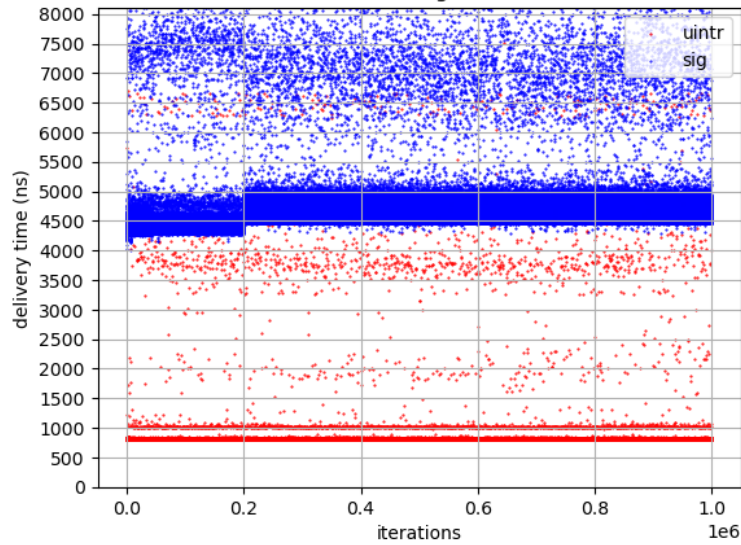
Figure 10 – Histogramme de distribution des mesures de la latence entre deux threads proche

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 retrouver 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)

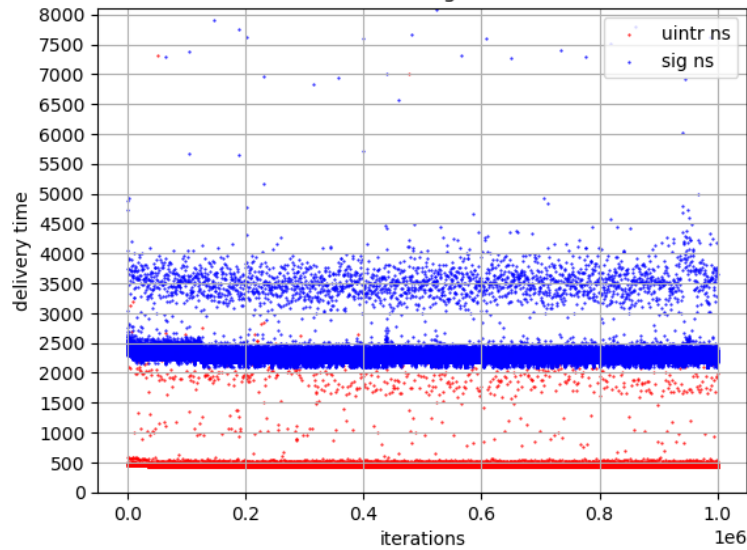
	mean	std	min	10%	50%	95%	max
sig	4601	403	4009	4396	4561	4796	58200
uintr	818	136	801	810	814	820	58072

(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
sig	2262	294	2081	2216	2251	2357	283171
uintr	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 différentes solutions
- solution d'Alexandre
- gestion des multiples interruptions qui s'écrasent, (prob_any / pour les larges si la progression à traiter le pipeline courant)
- quand est-ce qu'on envoie des interruptions ?
 - au moment où on poste le premier paquet d'envoi, pour indiquer au récepteur que des données sont disponibles (le récepteur a toujours un paquet de réception posté)
 - au moment où une progression est faite du côté du récepteur, pour indiquer une réception à l'émetteur
 - l'émetteur reçoit une interruption et détermine si l'émission est finie, petit paquet, ou si il faut envoyer la suite, paquet large.

5.3 Suite de tests

Tous les tests ne passent pas ?

5.4 Performances

5.4.1 Résultats avec attente active

TODO : voir le surcoupe des interruptions

5.4.2 Résultats recouvrement communication par du calcul

TODO : courbe overlap reception

6 Bilan

7 Remerciements

8 Annexes

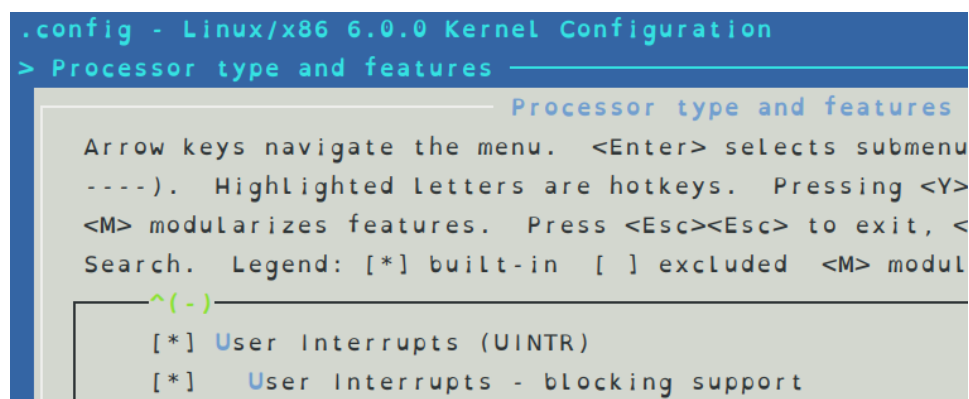


Figure 13 – Activé le support des uintr à la compilation du noyau

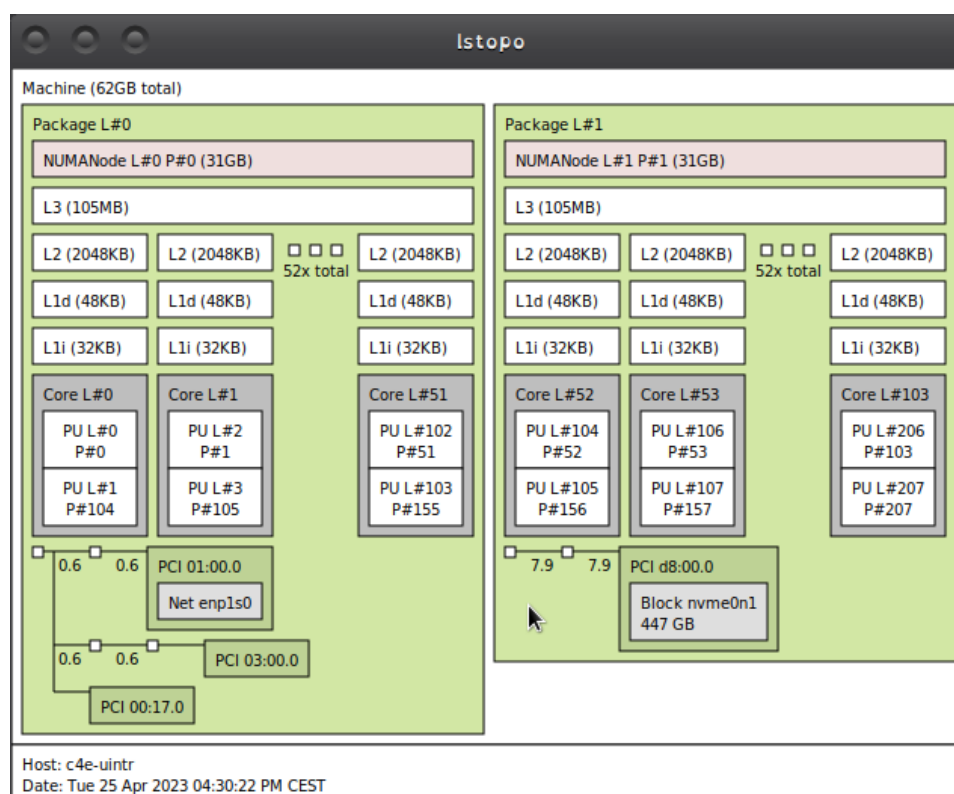
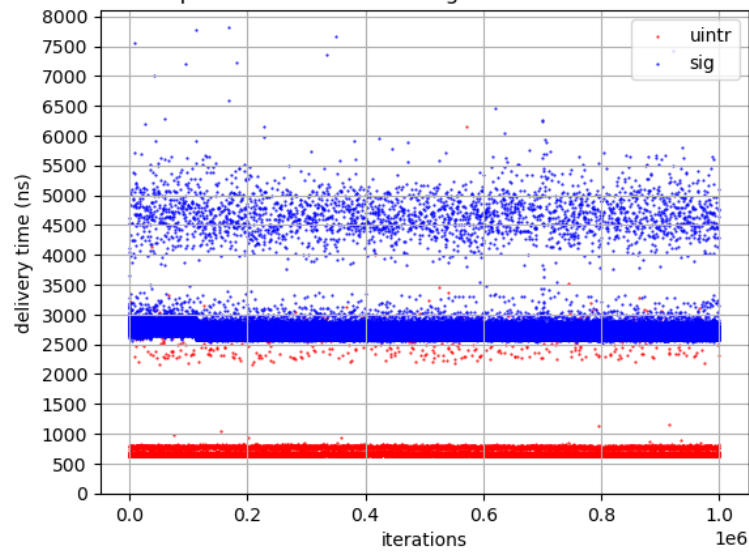


Figure 14 – Topologie de la machine fourni par Atos obtenus grâce à la command *lstopo*

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



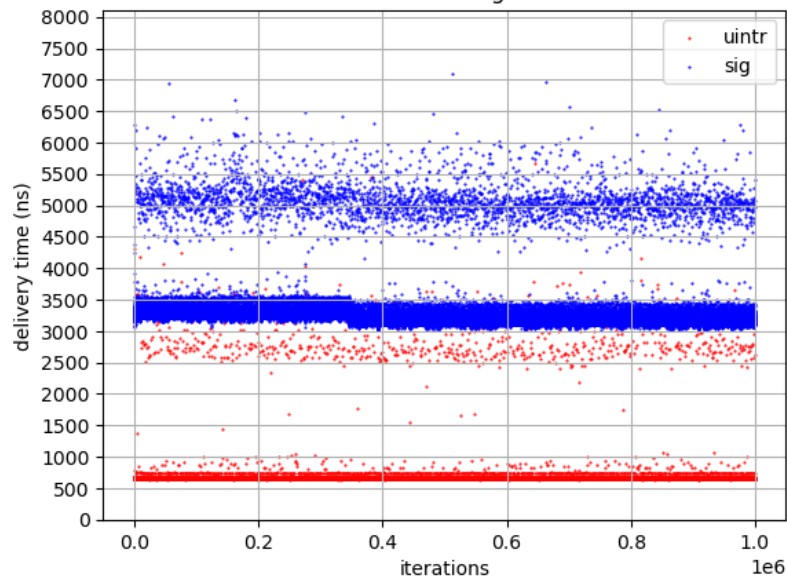
(a)

	mean	std	min	10%	50%	95%	max
sig	2675	147	2520	2620	2658	2781	44311
uintr	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)



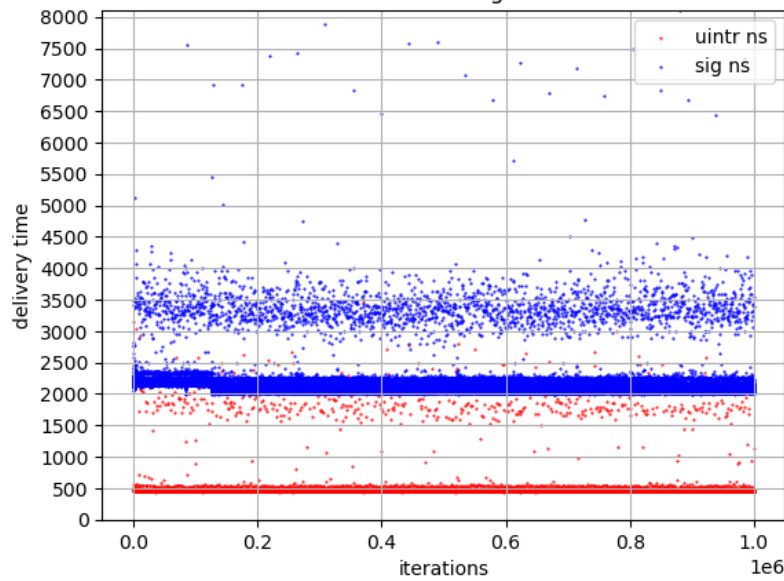
(a)

	mean	std	min	10%	50%	95%	max
sig	3210	156	3014	3141	3177	3302	63690
uintr	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)



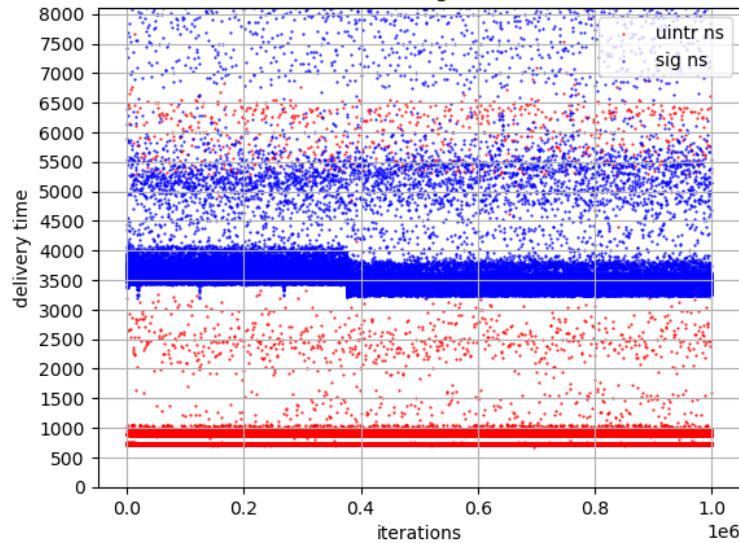
(a)

	mean	std	min	10%	50%	95%	max
sig	2097	282	1981	2064	2082	2178	270625
uintr	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
sig	3520	327	3189	3383	3481	3678	51798
uintr	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

9 Bibliographie

- [1] Olivier Aumage, Elisabeth Brunet, Nathalie Furmento, and Raymond Namyst. NewMadeleine : a Fast Communication Scheduling Engine for High Performance Networks. In *Workshop on Communication Architecture for Clusters (CAC 2007)*, workshop held in conjunction with IPDPS 2007, Long Beach, California, United States, March 2007.
- [2] Mathieu Barbe. Rapport de projet de fin d'études : Design and implement interrupt-handling code of a low-latency network adapter linux driver, 18 mars 2019 - 27 septembre 2019. Chez Bull (Atos). Grenoble INP - ENSIMAG.
- [3] Alexandre Denis. pioman : a pthread-based Multithreaded Communication Engine. In *Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Turku, Finland, March 2015.

- [4] Alexandre Denis, Julien Jaeger, Emmanuel Jeannot, and Florian Reynier. One core dedicated to MPI nonblocking communication progression? A model to assess whether it is worth it. In *International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, Taormina, Italy, May 2022.
- [5] Alexandre Denis and François Trahay. MPI Overlap : Benchmark and Analysis. In *International Conference on Parallel Processing, 45th International Conference on Parallel Processing*, Philadelphia, United States, August 2016.