# Politique de placement optimal dans une architec NUMA

Amira Artebas Damien Carver Christophe Julien
Hakan Metin Olivier Pitton Gauthier Voron

10 mai 2013

# Table des matières

**Résumé**

Les systèmes NUMA(*Non Uniform Memory Access*) connaissent un fort développement ces dernières années. Ces architectures possèdent des zones mémoires séparées utilisant différents bus. Or le noyau Linux n'est pas adapté pour prendre en compte cette séparation, cela entraîne des phénomènes de saturation des contrôleurs mémoire et de l'interconnexion entre les noeuds. Ce PSAR a pour objectifs d'-explorer différentes politiques de placement mémoire et d'implémenter dans le noyau un modèle permettant de supporter les systèmes NUMA.

# Chapitre 1

# IBS

## 1.1 Mesurer les performances

### 1.1.1 Problématique

Réaliser un programme permettant de mesurer les performances mémoire de divers programmes et les remonter en mode utilisateur.

### 1.1.2 Introduction

La complexité des systèmes a énormément augmenté depuis les dernières décennies. Systèmes hiérarchiques de cache, mémoire non-uniforme, multithreading simultané ... ont un impact important sur les performances et la capacité de calcul des processeurs modernes. Les mesures classiques d'utilisation de CPU ne sont pas assez précises. Détaillons ces mesures.

#### Lorsque l'utilisation CPU ne nous donne pas l'utilisation du CPU

Le pourcentage d'utilisation du CPU obtenu par les systèmes d'exploitation (abrégé OS) est une mesure qui a été utilisé pour plein de raisons comme l'ordonnancement de tâches, la planification de capacité de calcul, etc... L'implémentation actuelle de cette mesure (le nombre que l'utilitaire "top" sur les UNIX et le Windows task manager donne) montre la portion de créneaux temporels que l'ordonnanceur, dans le système d'exploitation, peut assigner à l'exécution de processus, ou de l'OS lui-même ; Le reste du temps est vain. Pour les tâches de calcul CPU (compute-bound), l'indicateur d'utilisation du CPU calculée de cette manière prédit très bien la capacité du CPU pour les architectures des années 80, qui avaient des performances plus uniformes et prédictibles que les systèmes modernes. Les avancées dans les architectures ont fait de cet algorithme une métrique douteuse à cause de l'introduction au multi-coeur, aux systèmes à plusieurs CPU, caches de plusieurs niveaux, mémoire non uniforme, pipelining etc...

**Mesures avancées**

Il est important de pouvoir mesurer les détails sur les performances d'un programme dans le but de trouver des améliorations. Jusqu'à maintenant, il y avait seulement deux manières de le faire. La première est via l'instrumentation, soit ajouter du code au programme pour regarder l'horloge, le compteur de cycles, ou juste pour compter le nombre de fois qu'une instruction ou une boucle est exécutée. L'instrumentation peut être ajoutée par le développeur ou par le compilateur. Malheureusement, cela perturbe sérieusement l'application, et le code instrumenté n'a généralement pas les mêmes caractéristiques que le code original, surtout quand on manipule les données et instructions des caches. De plus, l'instrumentation ne peut pas observer les caches matériels, donc il ne peut pas rassembler des informations sur le comportement d'un cache.

La seconde méthode traditionnelle de mesure de performances est d'utiliser les hardware performance counters. Ceux-ci comptent les événements matériels et génèrent une interruption après qu'un nombre d'événements programmé soit arrivé. Les compteurs peuvent donner des informations sur des événements qui sont trop difficiles à instrumenter (comme le nombre d'instructions x86) ou qui ne sont pas visibles au programme (comme les cache miss). Ils offrent donc de profondes informations sur l'application et les performances du système. Cependant, à chaque fois qu'un échantillon de données est assemblé, le processeur doit envoyer une interruption à un driver noyau, prenant plusieurs centaines ou milliers de cycles. Le driver, simplement en s'exécutant, change les contenus du cache de données et du cache d'instruction et peut perturber les performances de l'application. Les compteurs peuvent seulement être configurés, démarrés et stoppés depuis le mode système, donc une application doit appeler un driver du système d'exploitation pour les contrôler. Enfin, plusieurs systèmes ne provoquent pas de commutation de contexte des compteurs de performances lorsque les threads ou processus changent, et sur ces systèmes, la surveillance des performances peuvent seulement être faites globalement par un utilisateur à la fois.

### 1.1.3 Conclusion

La seconde approche est de loin la meilleure compte tenu de nos objectifs. Le noyau Linux gère l'interface entre les "hardware performance counter" Intel et AMD, facilitant le développement et l'utilisation des compteurs. Nous préconisons donc l'utilisation de Instruction Based Sampling, l'implémentation d'AMD, détaillée ci-après.

## 1.2 Instruction Based Sampling

### 1.2.1 Introduction

*Instruction Based Sampling*, IBS, est une technique utilisée pour obtenir des informations précises quant aux événements effectués par des processeurs. Pour

cela, il utilise des statistiques afin d'éliminer les dérapages potentiels, et met donc en évidence un résultat global.

Les instructions pipeline des processeurs ont deux phases principales : "récupérer l'instruction" et "exécuter l'instruction". La phase de récupération rapporte l'instruction à exécuter et la phase d'exécution effectue le traitement associé à l'opération, nommé "ops". Puisque les deux phases sont distinctes, IBS offre deux types d'échantillon : "fetch sampling" et "ops sampling", correspondant respectivement à un échantillonage sur la récupération des instructions à effectuer et un échantillonage des exécutions de ces instructions.

Le fonctionnement des deux techniques est similaire puisque basé sur un échantillonage. Périodiquement, lBS sélectionne une opération. Celle-ci est profilée durant la totalité de son passage dans le pipeline et les événements associés sont enregistrés. A la fin de l'exécution de l'opération, les événements et l'adresse de l'instruction sont envoyés au profiler.

Enfin, IBS offre plusieurs avantages face aux performance counter sampling, PCS.

1. Les événements matériels sont précisément attribués aux instructions causant les événements. Les PCS ne sont pas aussi précis rendant difficile, voir impossible, cette capacité ;
2. Les adresses physiques et virtuelles du chargement et du déchargement des opérandes sont collectés ;

Dans cette partie, nous présenterons IBS en profondeur et son utilité pour réaliser ce projet.

### 1.2.2 Objectifs

IBS est un formidable outil pour le monitoring de matériel. Grâce à lui, nous pouvons récupérer un ensemble d'informations, présentées ci-après, nous permettant de placer les pages mémoire intelligemment. Notre objectif est donc d'utiliser cette technique pour la réussite de notre projet, en réalisant un module Linux capable d'interagir avec le mode utilisateur.

### 1.2.3 IBS Fetch Sampling

IBS fetch sampling compte la totalité des récupérations et sélectionne périodiquement une récupération pour la taguer et la surveiller. Toutes les informations sont enregistrées pour chaque IBS fetch sampling.

Différents types d'informations sont collectés :

1. L'adresse récupérée.
2. Lorsque la récupération s'est terminée ou a été annulée.

3. Lorsque la récupération a raté, fait un miss, dans le cache d'instruction (IC), les caches de niveau 1 et 2 de la instruction translation lookaside buffer (ITLB).
4. La taille de la page de l'adresse traduite.
5. La latence, soit le nombre de cycles entre le commencement et la fin de la récupération.

### 1.2.4 IBS OP Sampling

Un op est une exécution d'opération discrète, réalisée par la famille des processeurs AMD 10th.
IBS op sampling compte le nombre de cycles processeurs et périodiquement sélectionne un op pour le taguer et le surveiller, de la même manière qu'IBS fetch sampling. Un IBS op sample est généré lorsque l'op taggué est retiré. En revanche, rien n'est généré si l'op taggué est abandonné.

Les informations obtenues par IBS op sampling sont les suivantes :

1. L'adresse de l'instruction pour le op.
2. Le nombre de cycles entre le moment où l'op a été taggué et a été retiré.
3. Le nombre de cycles entre le moment où l'op s'est terminé et a été retiré.
4. Lorsque l'op effectue une opération de chargement ou de déchargement :
   (a) Si l'opération a fait un miss dans le cache de données (data cache)
   (b) Si l'opération a fait un miss dans les caches de niveau 1 et 2 de la data translation lookaside buffer (DTLB)
   (c) La taille de la page de la traduction d'adresse de niveau 1 ou 2
   (d) La latence, en nombre de cycles, si l'opération a fait un miss dans le cache de données.
   (e) Les adresses virtuelles et physiques de zone mémoire demandée
   (f) Si l'accès a été réalisé sur un noeud local ou distant

### 1.2.5 Résultats obtenus

Les développements furent infructueux du au matériel des machines testées. En effet, le noyau Linux, dès l'initialisation d'IBS, renvoie un code d'erreur et ne charge rien d'IBS. Ainsi, il n'était pas possible d'utiliser les événements matériels.
Néanmoins, afin de faciliter le développement du module, nous nous sommes penchés sur l'étude d'un profiler fonctionnant avec IBS, OProfile.

## 1.3 OProfile

### 1.3.1 Introduction

OProfile est un profiler mémoire pour Linux. Il est composé d'un module noyau, d'un démon et d'un ensemble d'utilitaires en espace utilisateur. Les modes d'instrumentation portable de OProfile utilisent les timers système pour générer des événements de mesure à intervalles réguliers. Certains modes, spécifiques à certains processeurs mais moins intrusifs, consistent à utiliser les hardware performance counters intégrés. Oprofile permet de profiler le système entier ou bien un sous ensemble tel que les routines d'interruption, les pilotes de périphériques ou les process en espace utilisateur. Le surcoût de l'instrumentation reste faible.

### 1.3.2 Objectifs

Utiliser et modifier le module Linux d'OProfile pour renvoyer les informations dont nous avons besoin.

### 1.3.3 Fonctionnement

OProfile collecte des statistiques sur des échantillons, tout comme IBS. Le processeur est interrompu à des intervalles réguliers (les interruptions arrivent après un certain temps écoulé, ou qu'un compteur matériel de performances à accumler un certain montant d'événements) et le driver OProfile identifie quel code avait le contrôle à ce moment précis. La partie de code qui a eu la chance d'être interrompue par le profiler se voit attribuer un échantillon OProfile. Les parties de code qui prennent beaucoup de temps à s'exécuter sont naturellement plus aptes à accumuler des échantillons. En fait, le montant d'échantillons OProfile collecté pour une fonction tend à être directement proportionnel au temps d'exécution pris par cette fonction. Le fonctionnement peut être perçu d'une certaine manière comme proche de la méthode de Monte Carlo.

OProfile s'appuie directement sur les événements matériels pour instrumenter un processus. Un module s'occupe de la gestion des événements, de faire l'interface avec le code noyau d'IBS, ... et un driver a à sa charge de renvoyer les résultats vers l'espace utilisateur. Son utilisation en mode utilisateur est simpliste, apportant un véritable gain pour le projet.

Deux utilitaires vont nous intéresser :

1. *opcontrol* : permet de contrôler le daemon qui collecte les données qui sont sauvées périodiquement sous le répertoire /var/lib/oprofile/samples.
2. *opreport* : permet d'afficher les données de base.

OProfile nous permet de spécifier la vmlinux à utiliser et les événements que nous souhaitons observés. Nous avons donc un contrôle total sur les données à observer. Enfin, l'utilitaire nous permet de récupérer les échantillons récoltées

après l'extinction du profiler. Ces données peuvent être formattées en XML, gprof, ...

### 1.3.4 Résultats obtenus

Puisque les événements matériels ne fonctionnent pas sur les machines hôtes, bien évidemment leur utilisation avec OProfile ne fonctionne pas, affichant un message d'erreur. Néanmoins, ce profiler nous offre la possibilité de nous abstraire d'une forte partie de code, puisque les grandes fonctions ont déjà été écrites, qui plus est par le personnel d'AMD.

## 1.4 Conlusion

Le travail effectué derrière cette partie a été bloqué par le matériel utilisé, rendant impossible le développement d'un module noyau. Il en ressort trois idées pour l'avenir de cette partie du projet.
Tout d'abord, développer directement dans le noyau en se basant sur IBS. Une documentation et des modèles UML ont été fournis en supplément de ce document.
Ensuite, utiliser le code d'OProfile et le modifier pour en obtenir ce que l'on désire. Obtenir son propre format de sortie, ...
Enfin, utiliser OProfile et les données qu'il nous fournit. Cela requiert de s'adapter au format du profiler, mais est de loin la solution la plus simple, puisque permettant de s'abstraire du développement.

# Chapitre 2

# Politique déterministe

## 2.1 Équilibrage de Charge

Dans cette section, nous proposons un algorithme itératif pour équilibrer le nombre d'accès sur les contrôleurs mémoires. Cette approche Best Effort autorise uniquement la migration de page comme mécanisme d'équilibrage. Le placement des threads n'est pas modifié et les pages ne sont pas dupliquées. Grâce à cette simplification du problème, on peut espérer obtenir une réponse dans un temps raisonnable. En contre partie, il faudra appliquer en amont de ce traitement un autre algorithme de placement des threads. En effet, on peut considérer que pour un contrôleur, la charge produite par un accès d'une thread distante est la même que celle produite par un accès d'une thread locale. Ainsi, le placement des threads n'influe pas sur la charge.

À chaque début d'itération, le noyau peut mettre à jour les paramètres d'entrée. En fin d'itération, si le gain semble attractif, l'algorithme peut ordonner une migration de page, dans le cas contraire, on pourra soit attendre le résultat de plusieurs itérations, soit abandonner. Cet algorithme est efficace si la variance des accès sur les contrôleurs mémoire est initialement élevée, car son objectif est d'ordonner au plus vite une migration afin de diminuer la variance. L'algorithme prend en entrée la charge des contrôleurs mémoires, le placement initial des pages et la charge de ces pages.

L'itération consiste à rechercher la migration qui minimise la variance. Naïvement, il suffit d'enumérer toutes les migrations possibles (il y en a N*P) et de calculer la variance engendrée. On remarque rapidement que la page à migrer se trouve sur le noeud le plus chargé et qu'elle migre vers le noeud le moins chargé. Ceci réduit la recherche aux pages du noeud le plus chargé.

En ce qui concerne le calcul de la variance, si l'on observe sa formule, il n'y a que deux termes dans la somme qui changent : le terme concernant le

noeud le plus chargé qui sera déchargé, et le terme concernant le noeud le moins chargé qui sera surchargé. Il n'est donc pas nécessaire de recalculer entièrement la variance si l'on possède déjà la moyenne et l'ancienne variance. Notons X la charge moyenne, Cp la charge de la page à migrer, Xi la charge du noeud le plus chargé et Xj celui du moins chargé. Le gain s'exprime ainsi :

$$Gain * (N-1) = (X - X_i)^2 - (X - (X_i - C_p))^2 + (X - X_j)^2 - (X - (X_j + C_p))^2$$

Nous avons réalisé des simulations et les resulats sont satisfaisants. Les deux types de recherche (complète et avec l'heuristique du noeud max et du noeud min) donne des resulats équivalents mais avec des temps d'exécution différents. Nous avons considéré 10 noeuds, 1000 pages dont la charge est distribuée uniformément entre 0 et 10. Ces pages ont été aléatoirement distribuées sur les 10 noeuds. Les deux recherches effectuent ici le même nombre de migration : 65. Mais la recherche avec heuristique se termine en 0.7 sec alors que la recherche complète se termine en 44.3 sec.

## 2.2 Résultats de l'implémentation en Octave

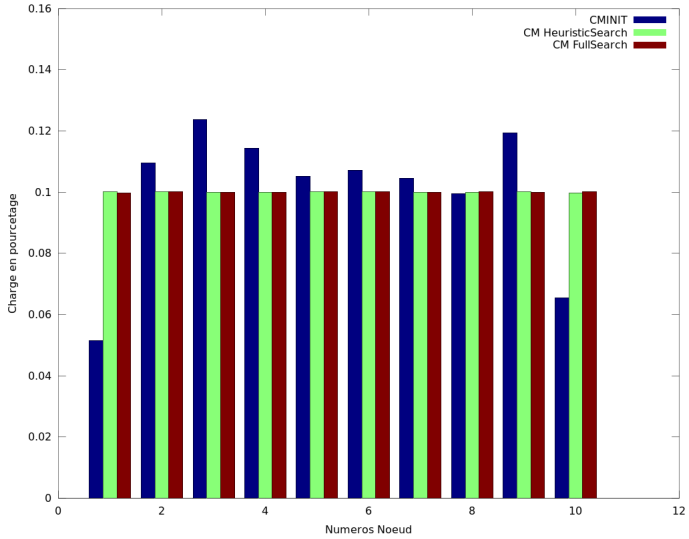



FIGURE 2.1 – Charge des contrôleurs à l'état initial et aux états finaux

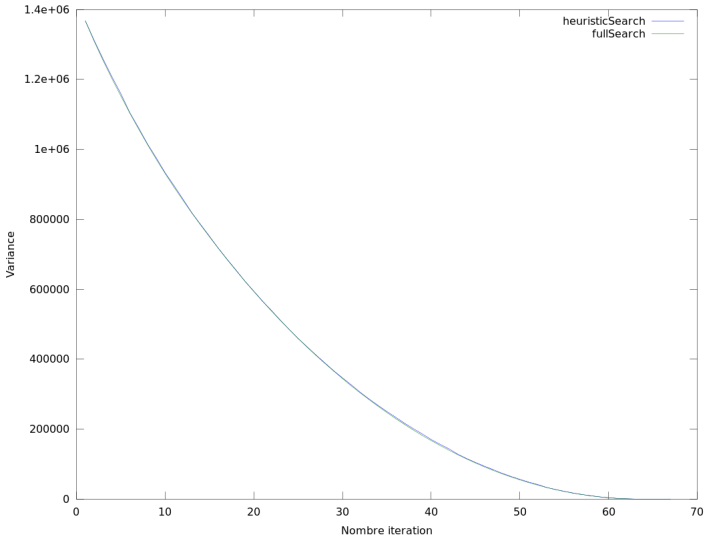



FIGURE 2.2 – Evolution de la variance en fonction du nombre d'itérations effectuées (les courbes sont presque confondues)

## 2.3 Recherche d'un placement optimal

Dans cette section nous proposons de résoudre ce problème à l'aide de la programmation par contraintes. L'objectif étant de proposer le meilleur placement des threads et des pages, la donnée essentielle au problème est le nombre d'accès effectués par chaque thread sur chaque page. Les pages pouvant être dupliquées, c'est-à-dire présentes dans plusieurs noeuds, une autre donnée à prendre éventuellement en compte serait le coût des mécanismes de synchronisation de ces pages dupliquées.

L'espace de recherche est particulièrement vaste. Nous ne tenterons pas dans ce projet de fournir des heuristiques de recherche. Nous nous contenterons de proposer un modèle simple permettant de décrire au mieux le problème de placement.

On définit trois variables principales de décision : une variable décrivant le placement des threads et une variable decrivant le placement des pages. Ces variables nous indiquent sur quel noeud une page ou une thread est présente. La troisième variable concerne le routage. Elle indique aux threads présentes sur un certain noeud, vers quel noeud s'adresser pour faire son accès sur une page donnée.

Ces variables doivent impérativement satisfaire les contraintes suivantes :
- Une thread est présente sur un seul noeud.
- Une page est présente sur au moins un noeud.
- Le nombre de threads dans un noeud est borné.
- Le nombre de pages dans un noeud est borné.
- Une thread ne doit pas accéder à une copie distante d'une page lorsqu'il existe une copie locale.
- Lorsqu'une thread souhaite accéder à une page, la variable de routage doit router vers un noeud qui contient la page concernée.

Afin d'obtenir une vision globale du système, il est possible de calculer à partir des variables de décision et des entrées le nombre d'accès qu'effectue un noeud sur un autre. Repésenter ces informations dans une matrice offre un double avantage :

Les lignes de la matrice représentent le nombre d'accès qu'effectue un noeud. Ceci permet d'introduire la contrainte de localité. La localité est respectée lorsqu'un noeud effectue plus d'accès sur lui même que sur les autres. En contraignant la matrice à être de diagonale dominante, on ordonne sur chaque noeud que le nombre d'accès locaux soit supérieur à la somme des nombres d'accès distants. Il est possible de rajouter un coefficient de dominance afin de pondérer l'importance de la localité.

Les colonnes de la matrice représentent le nombre d'accès effectués sur un noeud, c'est-à-dire la charge du contrôleur mémoire de ce noeud. Différentes stratégies peuvent être appliquées afin d'équilibrer les charges. Il est possible de minimiser la charge du contrôleur le plus chargé, mais les autres contrôleurs ne seront pas pris en compte. On peut également choisir de minimiser la variance des charges, mais le solveur aura tendence à tout dupliquer et ainsi augmenter la charge totale. On se propose ici d'utiliser une fonction de coût à minimiser qui représente le coût d'un accès sur un contrôleur saturé.

Pour estimer le coût de n accès sur un contrôleur saturé, on estime le nombre d'accès qu'il aurait pu faire dans un même temps si le contrôleur n'était pas saturé. Lorsqu'une application demande à effectuer n accès en parallèle sur un contrôleur mémoire saturé, ces accès se feront au mieux en séquentiel après n unités de temps. Le ième accès est effectué après i unités de temps. Si le contrôleur n'était pas saturé, la thread exécutant le ième accès aurait pu faire (i-1) autres accès sans attendre. Ainsi en faisant la somme totale du temps perdu à attendre, on obtient :

$$f(n) = n * (n + 1)/2$$

D'autre part, cette fonction favorise l'équilibrage de charge car la fonction "somme des coûts" (notée F) atteint son minimum lorsque la variance des charges est nulle. C'est-à-dire que F est minimisée lorsque la fonction de répartition (notée rep) répartit les charges partielles (notées ci) de manière identique (ci = C / n où C est la charge totale).

$$rep : C \mapsto (c_1, ..., c_n) \mid \sum_{i}^{n} c_i = C$$

$$F : rep \mapsto Cost \mid F(rep) = \sum_{i}^{n} f(c_i)$$

## 2.4 Résultats de l'implémentation en Éclipse CLP

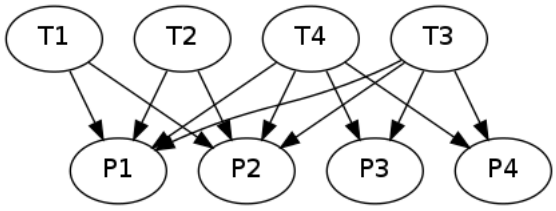



FIGURE 2.3 – Voici 4 threads et 4 pages à placer sur 4 noeuds. Un noeud peut contenir une thread et 2 pages.

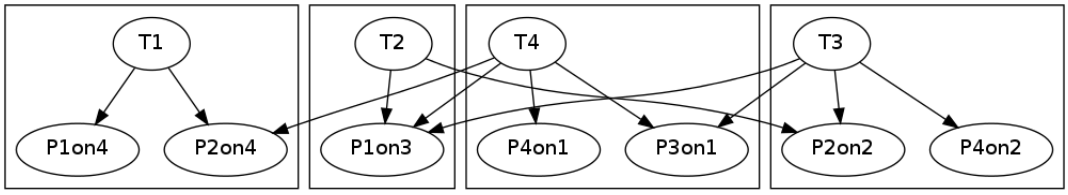



FIGURE 2.4 – Voici la solution proposée par l'implémentation du modèle. Les cadres représentent les noeuds. On observe que le mécanisme de routage force T2 et T4 à faire leurs accès sur des copies distincts de P2 afin de mieux équilibrer la charge.

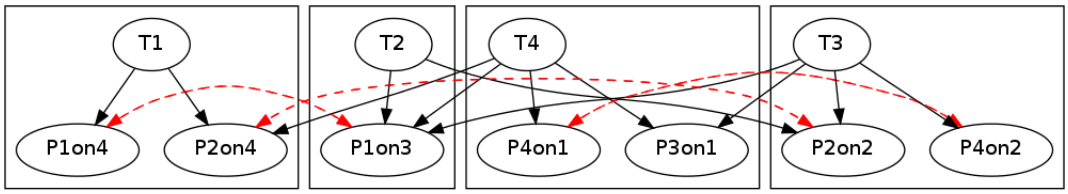



FIGURE 2.5 – Il faut cependant rappeler le coût des synchronisations ici représentées en rouge.

# Chapitre 3

# Algorithme génétique

## 3.1 Objectif

Dans notre présente étude on souhaite trouver un équilibre pour les accès mémoire entre les cœurs, ainsi que de minimiser la distance à parcourir entre un cœur et la mémoire à laquelle il accède, et pour atteindre nos objectifs il faudrait pouvoir définir des règles permettant de trouver un placement idéal des pages/threads sur les différents nœuds de façon à minimiser la latence d'accès, augmenter la localité et assurer que les contrôleurs mémoire soient utilisés de façon équitable.

Pour trouver un placement idéal des pages sur les différents nœuds, il suffit de tester tous les placements possibles. Cependant, le nombre de placement se démultiplie à chaque essai, ce qui rend cette méthode trop coûteuse voire irréalisable !

Dans de nombreux problèmes de ce genre, l'ensemble des possibilités est trop vaste pour un examen exhaustif des solutions, ce type de problème appartient au groupe des NP-Complet ; c'est-à-dire qu'il n'existe pas à l'heure actuelle un algorithme permettant de trouver une solution optimale à un tel problème en un temps raisonnable. En revanche, il existe tout une classe d'algorithme permettant de trouver des solutions acceptables, et les al algorithmes génétiques en font partie.

## 3.2 Solution proposée et Motivation

C'est pour cela, que les algorithmes génétiques semblent une bonne piste, qui nous permettra de rechercher une solution de façon efficace, une solution assez proche de l'optimale, Car les AGs fournissent des solutions aux problèmes n'ayant pas de solutions calculables en temps raisonnable de façon analytique ou algorithmique.

En fait, le principe de base des AGs provient de la théorie de Darwin sur l'évolution des espèces. Dans le monde du vivant, la sélection naturelle oriente

l'évolution des populations dans le sens d'une meilleure adaptation des individus à leur milieu.

Dans le monde informatique, les AGs permettent de rechercher une solution de façon efficace. Les AGs s'inspirent du phénomène de sélection naturelle pour la construction itérative d'une solution optimale à partir du croisement/mutation de solutions non optimales bien choisies. L'évolution d'une population est rendue possible par les croisements génétiques entre individus et par les modifications ou mutations de leur code génétique. L'utilisation d'un AG est particulièrement adaptée à cet exemple vu l'indépendance des solutions potentielles, la simplicité de créer des solutions potentielles et vu sa formalisation 'relativement' simple. Dans ce rapport, on va proposer une implémentation en C, un peu générique, adaptable à ce genre de problèmes.

## 3.3 Mécanismes de base :

- Page co-location placer une page sur le nœud du thread qui l'accède
- Page interleaving placer une page sur un nœud au hasard en respectant une distribution équitable
- Pages replication placer une copie d'une page existante dans un autre nœud
- Thread clustering placer les threads qui se partagent des pages sur le même nœud.

Nous cherchons à réutiliser des données et des mécanismes similaires dans notre approche.

## 3.4 Déroulement globale de l'algorithme

### 3.4.1 Evolution des générations

Dans notre exemple : notre population c'est l'ensemble des individus, un individu est un Thread, son évaluation est le déplacement total effectué et la page à la quelle il accède. De manière plus formelle, voici un algorithme génétique de base :

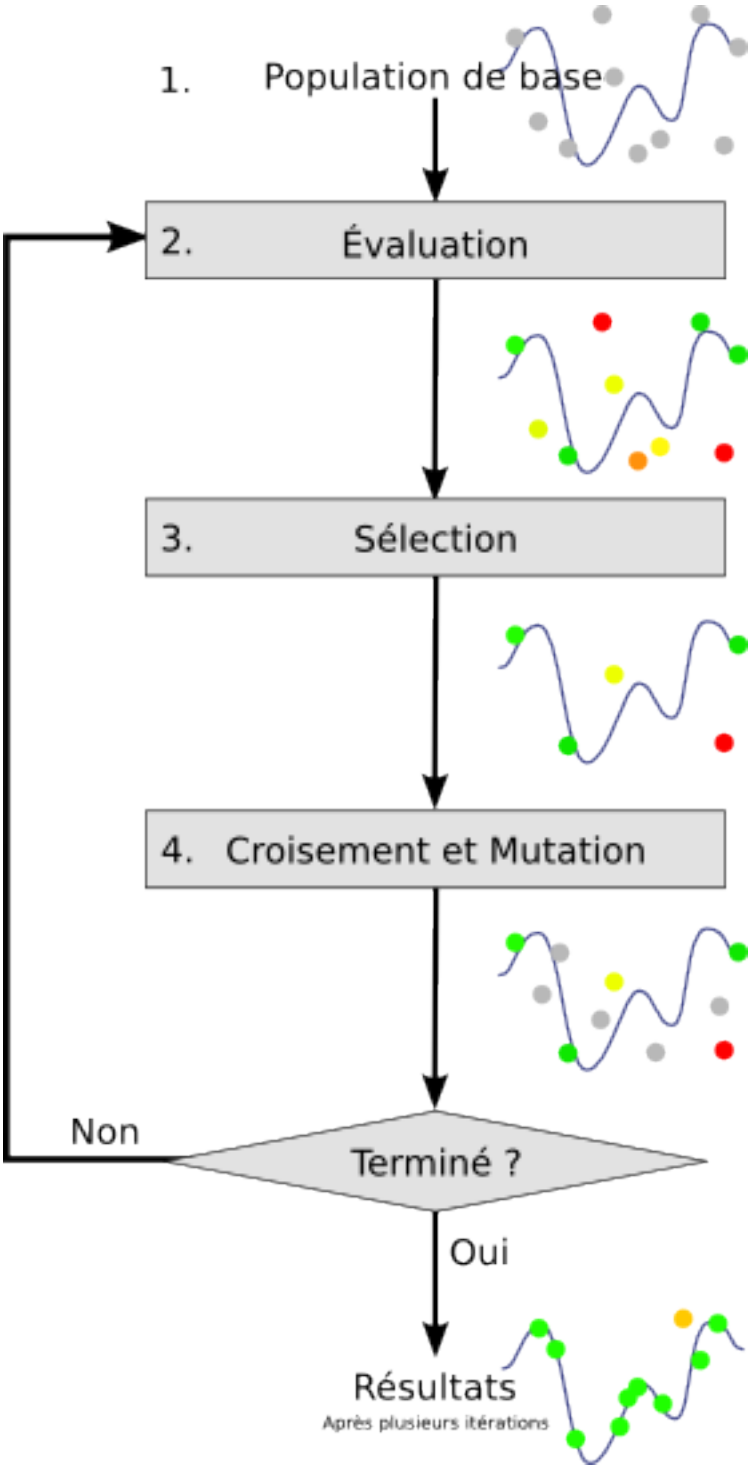

1. Population de base
2. Évaluation
3. Sélection
4. Croisement et Mutation
Non
Terminé ?
Oui
Résultats
Après plusieurs itérations

FIGURE 3.1 –
15

### 3.4.2 Évaluation (calcule de fitness) :

C'est sans doute la partie la plus importante et délicate, car il faudrait prendre en compte différents critères d'efficacités. On placera tout d'abord nos pages, après on lancera nos threads et on cherche a ce que les threads et les pages sois sur le même cœur, si ce n'est pas le cas on pénalise. Donc pour notre exemple, nous devons prendre en compte les contraintes suivantes : Tout cela en gardant à l'esprit qu'il faudrait :

Eviter de migrer un Thread d'un nœud sur un autre. Essayer de maintenir les threads d'un processus sur le même nœud. Essayer de maintenir chaque page sur son nœud local.

En d'autre terme, et si nous classons nos contraintes par ordre d'importance, il faudrait : Et s'assurer que les contrôleurs mémoire soient utilisés de façon équitable (équilibrage de charge). Augmenter la localité (Minimiser la latence d'accès). On pourrait avoir la fonction de Fitnesse Générale (qui vise à Minimiser la surcharge des inter-connecteurs =¿ Equilibrer la charge).

### 3.4.3 Sélection :

Comment choisir les parents pour la reproduction ? Il existe plusieurs possibilités : au hasard préférer les individus avec une meilleure évaluation (stratégie élitiste) toujours choisir le meilleur individu comme premier parent et un autre individu au hasard comme deuxième parent (variante de la stratégie élitiste) sélectionner les deux parents selon un critère de proximité dans la liste des individus Il semble qu'une stratégie élitiste soit plus efficace au démarrage de l'algorithme, puis au fur et à mesure que l'on se rapproche de l'optimum, chaque individu a à peu près autant de chances d'être choisi pour être le nouveau candidat de notre solution. Pour notre problème on a pensé à utiliser une stratégie élitiste inspirée de la "roulette wheel" : On considère la sélection d'un individu comme le lancement d'une bille sur une roulette. Chaque individu possède plusieurs cases dans la roulette. Les meilleurs individus possèdent plus de cases et sont donc sélectionnés plus souvent.

### 3.4.4 Croisement :

C'est la fonction principale de l'algorithme génétique. La recombinaison (ou reproduction ou crossover) consiste à créer un individu à partir de deux individus parents. Il existe presque autant d'algorithmes de recombinaison que d'implémentations d'algorithmes génétiques... On va décrire ici un des plus simples, le croisement à 1 point (c'est loin d'être le plus efficace mais il a le mérite d'être assez simple à coder). Il s'agit de recopier dans le fils une partie du parent 1, jusqu'à une "cassure", puis ensuite la partie correspondante du parent 2.

Exemple :

Parent 1 : On a un vecteur où :

0-2-1-0-0-0-1-0-1-3 // ici dans le 2éme cœur on a deux thread/page et un seul thread/page dans le 3éme 0 dans le 1er, le 4éme, le 5éme, et le 6éme etc..

Parent 2 : 2-0-1-3-0-1-0-0-3-1
Fils : 0-2-1-0-0-1-0-0-3-1

On détermine aléatoirement la cassure : ici par exemple entre le 4eme et le 5eme élément. Avant la cassure (pour les 4 premiers éléments du fils) : On recopie exactement chaque élément du premier parent dans le fils. Après la cassure (pour les 6 derniers éléments du fils) : On recopie chaque élément du deuxième parent dans le fils Nouvelle génération = sélection[roulette ?] ( ancienne génération + éléments mutés + éléments croisés) + échantillon des meilleurs individus de la génération courante Dans tous les cas nous sommes obligés de jeter certains individus, autant prendre le moins de risques.

## 3.5 Mutation :

Il s'agit d'une modification (plus ou moins aléatoire) du code génétique d'un individu. Cela permet de sortir des minimums locaux, grâce à une perturbation. On peut par exemple permuter un ou plusieurs couples de threads/pages pour retomber sur un résultat potentiellement meilleur. On peut considérer la mutation de deux manières : elle peut créer un nouvel individu qui correspondra à l'individu muté ou bien elle pourra directement modifier l'individu muté. La première approche permet de conserver l'individu originel au cas où sa mutation l'aurait rendu moins bon. C'est ce qu'on a choisi dans notre implémentation.

### 3.5.1 Réinsertion :

Comment réinsérer le fils dans le groupe ? Les solutions sont : Eliminer le moins bon du groupe Eliminer l'individu qui ressemble le plus au nouvel individu Remplacer un des deux parents La nouvelle génération remplace l'ancienne génération (sauf le meilleur individu de l'ancienne génération)

Ce qui est certain est qu'il ne faut pas supprimer le meilleur individu du groupe, sous peine de voir l'adaptation globale diminuer ! Paramétrages divers : Quelle est la taille d'un groupe (nombre d'individus) ? Faut il utiliser une heuristique pour initialiser la population (exemple : meilleurs noeuds) Le paramétrage doit il changer en fonction de la phase de calcul (par exemple, plus de mutations à la fin...) Certaines fonctions sont elles plus efficaces avec un nombre de threads /pages faibles ou élevé ? Dans le croisement, faut il préférer la rapidité (afin de faire plus d'itérations) ou des opérateurs plus sophistiqués mais plus longs ?

## 3.6 La partie modélisation :

### 3.6.1 -1ere étape !

Définir une structure de donnés : une seule structure nous intéresse, celle permettant de représenter notre solution. Il faut garder à l'esprit que cette structure va être recombinées afin d'en générer des nouvelles. Il faudrait donc, que notre structure soit la plus simple possible afin de ne pas trop compliquer

l'implémentation. L'idéal serait donc de travaille sur une structure à une dimension, c'est-à-dire un tableau de vecteurs dans la mesure du possible sinon on utilisera une matrice, nous pouvons alors imaginer la modélisation suivantes :

Soit : un thread[numpage][numthread] et page[numpage]
Si (ces deux tableaux sont égaux) =>le thread et la page son sur le même cœur
Sinon : on duplique la page sur le cœur ou se trouve le thread.
Ses tableaux représentent la distribution des pages et des threads sur les cœurs
Exemple :page[3] = 5 veut dire que la page 3 est sur le cœur 5 etc..
Et si on a thread[2][3] = 5 veut dire le 3eme thread de la 2eme page est sur coeur 5
Il faudrait :

1. assigner des pages à des nœuds dans un tableau page[nbpage], où chaque élément sera un nœud (ici on parle pas de page de réplication )
2. on a la matrice thread[i,j] dont les coefficients sont (i, j) i représente la page et j le numéro du thread qui utilise les ressources pages !
3. On assignera des nœuds à un élément de matrice.
4. Par la suite, on va récupérer les threads et les pages qui sont sur le même nœud

Pour les threads et les pages qui ne sont pas sur les mêmes nœuds, on dupliquera les pages et on appellera les threads des threads migrés et puis on donnera le coût de chaque opération. Pour l'évolution on va manipuler la matrice thread/population uniquement la matrice page ne changera plus. Les éléments de nos matrices sont bornés. (Infos qu'on récupère partir d'IBS pour qu'on puisse les utiliser en suite dans notre fct de fitness) On aura besoin de certains nombres d'informations (Infos qu'on suppose récupérer à partir d'IBS pour qu'on puisse les utiliser en suite dans notre fct de fitness)

- Constant : capacité des nœuds en terme de threads/ pages.
- Variables : qui représente le nb thread / pages qui ont migré/migrant/dupliqué/supprimé
- Temps exécution d'un thread/nœud.
- Temps de migration 2*X tel que X est nombre d'accès locaux par unité de temps.

**Caractéristiques du modèle**

**Hypothèses** On suppose qu'il est possible de savoir pour chaque thread, son nombre d'accès à une page par unité de temps grâce à IBS, on supposera aussi qu'on connait pages les plus chaudes (ça signifie détecter dans le noyau la liste des pages les plus chaudes ou les plus, c'est à dire celles qui coutent réellement à l'exécution).

**Objectif** Répartir les threads et les pages dans les nœuds afin de minimiser le coût global de tous les accès

Donnés et variables de placement

### 3.6.2 -2eme étape !

L'envoi de thread sur les nœuds ! Au début un envoie aléatoire =>calculer le temps d'envoie. Ensuite le temps aléatoire pour le choix des nœuds/thread, va être raffiné avec évolution ! !

### 3.6.3 -3éme étape !

On va s'intéresser à l'exécution (le temps que va rester le thread sur les nœuds/ et sa façon de migrer) =>Gérer la migration/ les threads sur les nœuds !

### 3.6.4 -4éme étape !

Récupération des données : sommer l'ensemble des RES.

### 3.6.5 -5éme étape !

Résultat RES : on prend le modèle, et on décidera de le raffiner selon les expériences obtenues, puis on réitère jusqu'à la meilleure solution ! !

## 3.7 Critiques et discussions

Apres avoir réalisé plusieurs tests et simulation qu'on vous présentera lors de notre présentation, les résultats étaient peu concluant, même en changeant et on jouant sur le nombre d'expériences et d'individus, mais n'empêche que l'utilisation des algorithmes génétiques reste une bonne piste à suivre et à améliorer. Comme l'a si bien dit Thomas Edison « Non, je n'ai pas échoué, j'ai simplement découvert 2000 méthodes qui ne fonctionnent pas ! » De même, il lui a fallu plus de 25.000 essais avant de réussir à fabriquer la première pile électrique. C'est ce qu'il faut faire dans notre cas, Il faudrait essayer plusieurs configurations afin de trouver la solution optimale.

## 3.8 Conclusion

Apres plusieurs tests et tentatives, nous nous somme rendu compte que le codage des données pour modéliser un problème est complexe. D'autre part, nous nous sommes aussi aperçus des difficultés pour choisir pertinemment de bons paramètres pour les divers opérateurs (mutation, croisement, sélection, remplacement). Des choix par rapport aux opérateurs eux mêmes sont aussi à gérer, sachant que certains sont plus appropriés au problème et qu'ils permettent d'optimiser. Les algorithmes génétiques seuls ne sont pas très efficaces dans la résolution d'un problème. Ils apportent cependant assez rapidement une solution acceptable. Néanmoins, il est possible de l'améliorer assez efficacement en le combinant avec un algorithme déterministe.

# Chapitre 4

# Implémentation dans le noyau Linux

## 4.1 Introduction

Dans un premier temps nous détaillerons la méthode et le schéma de développement que nous avons choisi d'appliquer pendant ce PSAR. Nous étudierons ensuite la gestion de la mémoire dans le noyau Linux actuel, ce qui nous amènera à expliquer pourquoi celle-ci n'est pas adaptée aux systèmes NUMA. Nous présenterons alors les changements que nous avons apportés dans le gestionnaire de mémoire pour corriger ces problèmes. Pour finir nous examinerons en détail comment se passe l'allocation, le traitement des fautes de pages et la désallocation dans ce nouveau modèle.

Nous allons commencer par décrire le fonctionnement actuel du gestionnaire de mémoire dans le noyau Linux avant de présenter les modifications que nous y apportons.

## 4.2 Fonctionnement actuel du noyau

L'espace mémoire d'un processus est représenté par la `mm_struct`. Elle contient, entre autre, une liste des différents segments de mémoire (plage d'adresses continues en mémoire) et la table des pages.

Un processus ne contient qu'une seule table des pages qui sera partagée entre les threads. Elle permet de faire la correspondance entre les adresses virtuelles utilisées par le processus et les adresses physiques.

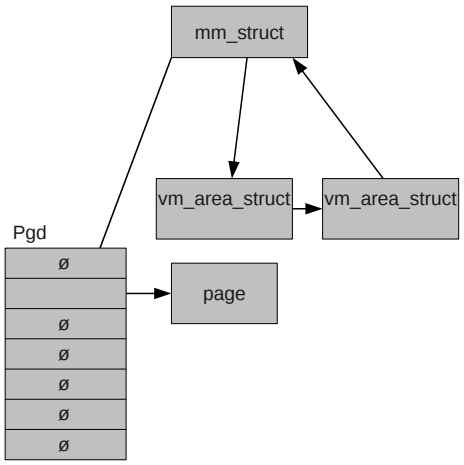
Lors de la création d'un nouveau processus, la table des pages est partagée en *copy on write* avec la table des pages du père. Puis lors du `exec` un nouveau

descripteur mémoire est alloué. La duplication n'intervient que dans le cas d'une écriture par un des processus sur une des pages qu'ils ont en commun.

La table des pages est partagée entre tous les threads d'un même processus.

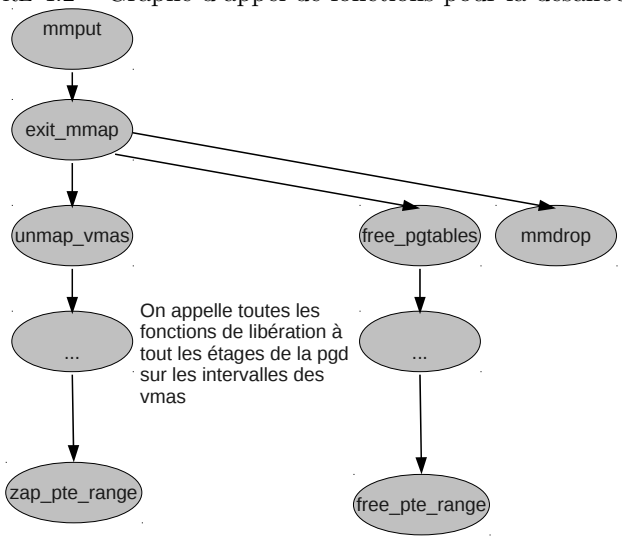
Elle se remplit au fur et à mesure des fautes de pages. Tous les threads accèdent à la même table des pages donc cela donne l'assurance qu'ils travaillent avec le même *mapping*. Cependant nous verrons dans la suite que ce n'est pas adapté aux systèmes NUMA.

FIGURE 4.1 –



La désallocation se fait en plusieurs étapes. La première consiste à désallouer les adresses virtuelles avec l'appel de la fonction `unmap_vmas()`. Elle même appelle `unmap_single_vma()` sur chaque `vma` pour libérer plage par plage les adressses virtuelles et mettre à jour les compteurs de la `mm_struct` afin de pouvoir tester si tout a bien été désalloué. Ensuite la table des pages est libéré à son tour, les `pte` sont également désalloués par plage, afin de mettre à jour les compteurs relatif aux `pte` (dans l'optique d'être certain de tout avoir désalloué). Enfin la `mm_struct` est libérée à l'aide de la fonction `mm_drop`.

FIGURE 4.2 – Graphe d'appel de fonctions pour la désallocation




## 4.3 Modifications apportées

Dans cette partie nous détaillerons les changements apportés aux structures mémoires.

Notre objectif est que chaque noeud puisse avoir son propre mapping. Dans le noyau Linux, c'est impossible puisque chaque processus possède sa propre table et les threads la partage.

Pour pouvoir associer un *mapping* mémoire différent à chaque noeud, nous avons créé une structure `numa_pgd_t` qui représente un ensemble de tables des pages (une par nœud). Les tables ont le même rôle et fonctionnement que celle de base, c'est à dire qu'elles assurent la traduction adresses virtuelles vers adresses physiques, et qu'elles se remplissent de la même façon qu'avant (grâce aux fautes

de pages). Cependant un problème apparait, il faut pouvoir garantir la cohérence entre les tables et avoir la possibilité de détecter un *mapping* déjà existant.

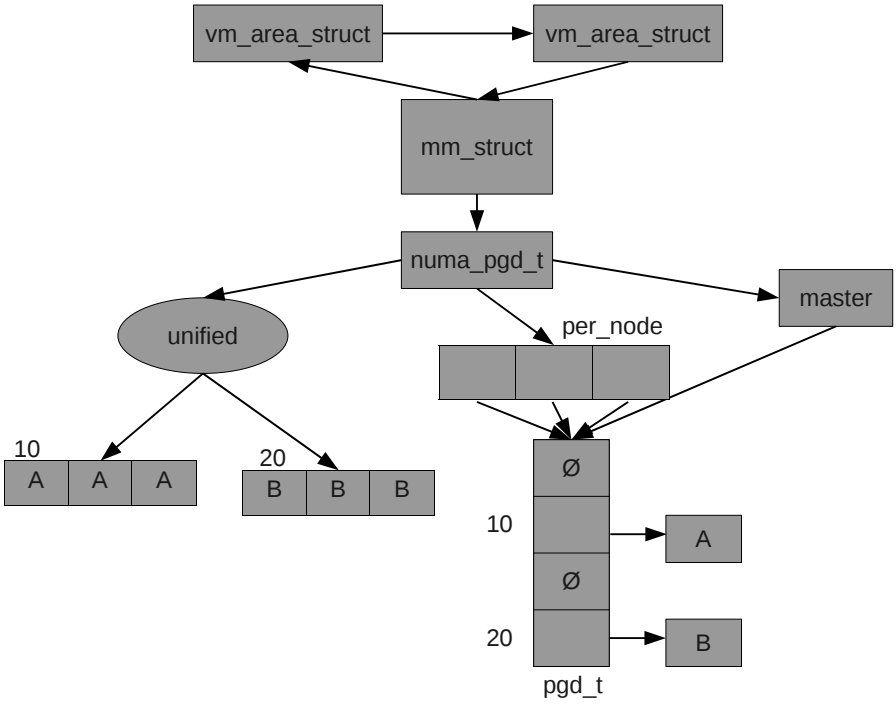
Cependant de nombreux processus sont *mono-threadés* donc il ne sert a rien d'avoir une table des pages dans chaque nœud. Au début, tous les nœuds pointent vers la même table des pages, qui est repérée par le pointeur `master`. Lorsqu'un nœud réclame un mapping qui lui est propre, on lui crée sa propre copie distincte de la `master`.

Nous avons mis en place une structure commune à tous les threads (localisée dans la `numa_pgd_t`), qui représente la table des pages unifiés du processus courant. Dans l'idéal, il faudrait une structure similaire à une `pgd_t`, cependant pour des raisons de temps nous avons préféré l'implémenter sous la forme d'un arbre rouge/noir (appellé `unified`). De plus, si nous avions essayé de respecter la forme d'une `pgd_t` nous aurions eu un problème pour représenter le dernier niveau, qui ne devrait plus pointer vers un `pte` mais vers un tableau de `pte`.

Chaque nœud de l'arbre possède une adresse virtuelle et un tableau `physicals` qui contient les pte de chaque table des pages. Si la table des pages du nœud 1 associe l'adresse virtuelle $x$ à la page mémoire $y$ alors le nœud de la table unifiée dont l'adresse virtuelle est $x$ contiendra dans `physicals[1]` la valeur $y$.

Tous les *mapping* étant présent dans la table unfiée cela nous apporte une vision globale de la mémoire d'un processus (voir le schéma).

FIGURE 4.3 – Schéma de la mémoire avec la numa_pgd_t




## 4.4 Méthodes de développement

La grande difficulté de ce projet est de s'intégrer dans un système complexe préexistant au projet. Cette contrainte implique d'une part, qu'il est impossible de savoir précisement ce qui existe dans le programme, que fait exactement telle ou telle fonction, quelles sont les hypothèses qui peuvent être faites sur cette structure de donnée. D'autre part, il est difficile, voir impossible, de s'assurer que le code produit est fiable car le code du noyau Linux est très vaste, et il n'est souvent pas possible de connaître tous les effets de bord d'un appel à une fonction.

Pour mener ce projet à bien malgré tout, la décision a été prise d'utiliser au maximum les fonctions déjà existantes pour plusieurs raisons. D'abord le noyau fonctionne de manière fiable avant nos modifications, ce qui signifie que toutes ses fonctions sont correctes, ensuite les fonctions du noyau ont été optimisées par des développeurs expérimentés, nous n'avons par leur compétence.

Enfin, de nombreuses fonctions du noyau sont capables de détecter des erreurs d'utilisation grâce à des macros comme `BUG_ON()` ou `WARN_ON()`.

Afin d'utiliser au maximum ce qui existe déjà dans le noyau, nous définissons le plan de développement général :

- La première étape consiste à créer notre nouvelle structure de donnée en parallèle à celle qui existe déjà et de ne travailler dessus uniquement quand le noyau est déjà lancé. Pour le savoir nous regarderons si l'UID courant est celui d'un utilisateur de test (par exemple l'UID 7000). Le développement en parallèle de notre structure permet de la comparer à tout moment à la structure standard et de vérifier qu'elle est cohérente.
- La seconde étape consiste à utiliser la nouvelle structure à la place de l'ancienne, là où elle est définie, par exemple lorsque l'UID vaut 7000. Pour une nouvelle table des pages, cela signifie charger notre nouvelle structure dans la TLB plutôt que l'ancienne. Cela permet de tester de manière plus concrète nos modifications sans que d'autres fonctions du système soient affectées.
- Enfin la dernière étape consiste à substituer complètement la nouvelle structure à l'ancienne, après cette étape, le système ne doit plus utiliser que notre structure.

Cette modification en trois étapes doit permettre de pouvoir comparer à tout instant notre nouvelle structure à l'ancienne, dont on sait qu'elle est valide, et ainsi tirer profit de la fiabilité déjà présente du noyau Linux.

En travaillant sur un programme de cette taille, il est facile de se perdre dans la multitude de sous fonction et de macro existantes, souvent dépourvues de commentaires. Voici la manière dont nous avons procédés pour avancer dans notre développement :

- Commencer par repérer les endroits où notre nouvelle structure doit interagir avec le noyau. Dans notre cas pour une table des pages modifiées, il s'agira de la duplication de processus (la fonction `dup_mm()`), de sa terminaison (la fonction `exit_mmap()`) ainsi que lors des fautes de page (la fonction `handle_mm_fault()`), sans oublier la définition d'un descripteur mémoire (`struct mm_struct`).
- À ces endroits correspondent généralement un appel de fonction, faire un test pour savoir si le noyau est déjà lancé (par exemple, l'UID courant est 7000). Si le test échoue, continuer la fonction normalement, mais s'il réussit, appeler un clone de la fonction originale (dont on préfixera le nom par `PSAR_`), dont on modifiera le comportement pour prendre en compte notre nouvelle structure.

Les sections suivantes abordent plus en détail l'implémentation de cette nouvelle structure.

## 4.5 Allocation

La nouvelle structure que nous définissons (`numa_pgd_t`) est une sous partie d'un descripteur mémoire (`mm_struct`) et doit donc être allouée en même temps que celui-ci. Cette allocation se produit à deux moments au cours de la vie d'un processus : d'une part à sa création (comprenez `fork()`), et d'autre part pendant son éventuel recouvrement (comprenez `exec()`). Une allocation simple est suffisante dans le cas d'un recouvrement, en effet, la plupart du travail se fera sur les adresses virtuelles et les zones mémoires seront physiquement remplies par des fautes de page (voir section suivante). La création, ou plutôt la duplication d'un processus demande plus de travail puisqu'alors les tables de page sont effectivement copiées, seules les pages physiques sont partagées en *copy on write* (COW).

Une méthode de copie d'une `numa_pgd_t` vers une autre est alors nécessaire. N'oublions pas que pour des raisons évoquées dans la section précédente, il est aussi nécessaire d'implémenter une méthode de copie depuis une `pgd_t` standard vers notre `numa_pgd_t`, qui sera appliquée lors d'un passage vers nos utilisateurs test (UID 7000). Pour nous aider dans notre développement, nous disposons déjà des fonctions qui copient depuis une `pgd_t` standard vers une autre. On peut également rappeler que nous n'aurons pas ici à nous soucier de concurrence puisque l'espace mémoire d'un processus ne peut pas être partagé pendant sa création.

Commençons par décrire la copie depuis une `pgd_t` standard vers notre nouvelle structure. Cette copie se veut simple puisque destinée à être supprimée, aussi même si elle doit être correcte, on peut la faire aussi lente et inefficace que souhaitée, cela n'aura aucune incidence. La manière dont procède cette méthode est la suivante :

- Commencer par effectuer une copie standard depuis la `pgd_t` originale vers la table de notre `numa_pgd_t` (on rappelle qu'à la création, tous les nœuds partagent la même table).
- Mettre ensuite à jour la table unifiée (l'arbre rouge-noir) pour qu'elle soit cohérente avec notre table nouvellement copiée.

Le développement de cette copie se fera rapidement étant donné que la première étape, copier une `pgd_t` vers une autre, est déjà implémentée, et que la mise à jour de l'arbre se fait assez facilement en parcourant simplement la table et en reportant les pages présentes.

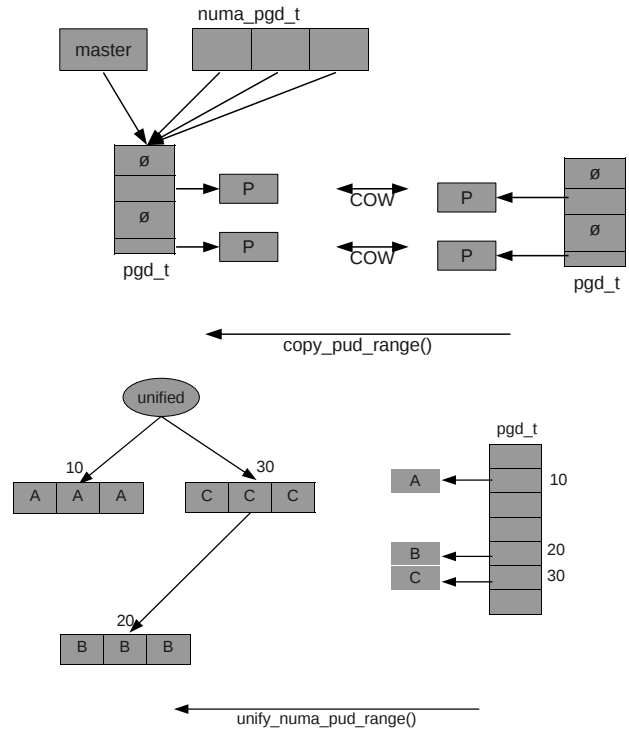



FIGURE 4.4 – Copie depuis une pgd_t vers une numa_pgd_t

Il est à noter que puisque cette fonction est temporaire, en cas d'échec d'allocation mémoire, la structure ne pourra pas être libérée correctement. En effet nous verrons dans la section suivante que notre structure `numa_pgd_t` ne peut être libérée que si l'arbre et les tables sont cohérents entre eux, ce qui n'est pas assuré dans cette fonction.

La seconde copie, depuis une `numa_pgd_t` vers une autre, est plus complexe. En premier lieu, celle-ci doit être fiable et suffisamment performante, et doit de plus pouvoir être désallouée en cas de problème. En second lieu elle travaille sur des données qui ne sont pas sensées être gérées par les fonctions du noyau déjà présentes. Pourtant ce sont bien ces fonctions qui doivent être utilisées si nous voulons éviter de réimplémenter le COW et tout le travail complexe qu'elles fournissent. Ces fonctions prennent en entrée les adresses des structures source et cible ainsi qu'une plage d'adresse. Le problème est que dans notre nouvelle structure, deux nœuds peuvent avoir des *mappings* mémoire différents, aussi les plages d'adresse de notre processus peuvent-elles se trouvées éclatées ou répliquées. On considère qu'avant une copie de descripteur mémoire, on s'est assuré que deux pages associées à une même adresse virtuelle contiennent les mêmes données (synchronisation de page), il ne reste donc qu'à gérer l'éclatement des plages

d'adresse.

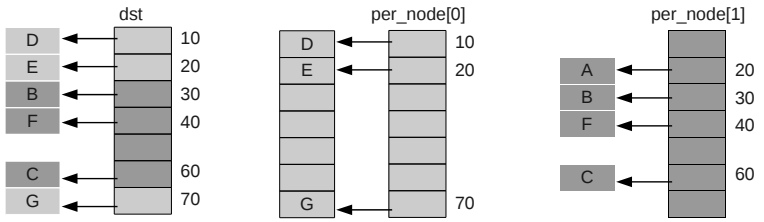



FIGURE 4.5 – Copie d'une plage d'adresse éclatée

L'éclatement est géré de la manière suivante :

- On parcourt l'arbre unifié pour une plage d'adresse donnée, l'itérateur sur un arbre rouge-noir est déjà fourni par l'API de Linux.
- Tant qu'il existe un *mapping* sur le nœud `X`, on continue à itérer.
- Lorqu'il n'y a plus de *mapping* pour ce nœud `X` ou lorsque l'on arrive à la fin de la plage d'adresse, on appelle la fonction du noyau de copie, puis si ce n'est pas la fin, on choisi un nouveau nœud `Y` pour lequel le mapping existe.

L'arbre unifié destination est mis à jour à chaque itération sur l'arbre source. En cas d'échec d'allocation mémoire, il est entièrement libéré pour la plage donnée, il ne restera alors qu'à libérer la table des pages destination (partagée par tous les nœuds) comme une `pgd_t` classique. Cette stratégie gloutonne permet de ne parcourir qu'une seule fois la plage d'adresse spécifiée.

Une fois les copies effectuées, on obtient une structure `numa_pgd_t` cohérente telle que tous les nœuds partagent la même table des pages. À ce stade, notre structure n'apporte donc rien de nouveau, une autre fonction existe qui a pour but d'assurer à un nœud qu'après un appel réussi, il dispose d'une table des pages bien distincte de celle des autres nœuds. Cette fonction procède simplement en vérifiant si le nœud indiqué utilise la même table des pages que la table maîtresse. Si ce n'est pas le cas, le nœud possède déjà sa propre table et la fonction retourne. Dans le cas contraire, on appelle une version modifiée de la fonction de copie de table (celle du noyau original) qui n'apportera pas la protection COW normalement fournie.

## 4.6 Fautes de page

Lors d'une faute de page, une interruption est levée par la MMU. Le gestionnaire d'interruption en charge des fautes de page varie selon l'architecture mais sous Linux, tous appellent finalement la fonction `handle_mm_fault()`.

La principale difficulté de cette fonction est la possibilité de fautes concurrentes. En effet, pour des besoins de performance, Linux autorise le fait que plusieurs threads puissent fauter simultanément sur la même adresse virtuelle. Il est donc nécessaire d'assurer une cohérence, même en cas d'accès concurrent, toutefois, on ne peut pas se permettre de mettre en place une synchronisation trop lourde car les fautes de pages doivent être résolues aussi rapidement que possible. La politique de Linux pour satisfaire ces contraintes est la suivante : essayer de travailler comme s'il n'y avait pas de concurrence et si après coup, on se rend compte que le système a été modifié, annuler le travail superflu.

Typiquement l'allocation d'un niveau de table des pages se fait de la manière suivante :

- On commence en n'ayant aucun verrou
- On alloue une page physique (cette routine synchronise comme elle l'entend).
- On prend un verrou (ici, un *spinlock*)
- On vérifie que personne d'autre n'a alloué entre temps
  - Si c'est le cas, on libère la page allouée
  - Sinon, on insère effectivement notre page
- On relâche notre verrou.

Ici le travail coûteux est l'allocation d'une page, placer cette allocation en section critique pourrait signifier des ralentissements lors des fautes de page, ce qui n'est pas souhaitable.

Une fois les allocations effectuées, la fonction `handle_mm_fault()` vérifie que les allocations sont effectives (pas d'échec), si ce n'est pas le cas, la fonction retourne en indiquant d'essayer à nouveau "plus tard". Au bout d'un certain nombre d'essai, le processus est tué. Si les allocations se sont passées correctement, la fonction `handle_pte_fault()` est appelée, son rôle est de déterminer quel est le type de faute : la page est-elle absente ou protégée, s'agit-il d'une page mémoire ou mappée sur un fichier, est-elle dans le *swap* ? Une fois le type de faute déterminé, une fonction de gestion spécialisée est appelée qui va détecter une erreur de segmentation ou bien gérer la faute de manière appropriée (en mettant au besoin la MMU à jour).

Notre objectif est que la nouvelle structure `numa_pgd_t` soit mise-à-jour correctement lors d'une faute de page :

- Si aucun nœud ne possède de page associée à l'adresse virtuelle en défaut, une faute de page normale a lieu.
- Si au moins un autre nœud possède une page associée, son *page frame number* (l'adresse de la page) est simplement copiée dans le nœud courant.

Pour cela, nous choisissons de faire exactement comme la gestion de faute de page originale à une exception près : au début des fonctions spécialisées qui insèrent une nouvelle page, on teste si un autre nœud que le nôtre possède déjà un *mapping* mémoire. Si tel est le cas, on copie le *page frame number* dans notre nœud et on retourne immédiatement. Si en revanche la faute de page doit avoir lieu, on n'oubliera pas de mettre à jour l'arbre unifié. Pour les raisons de synchronisation évoquées plus haut, on s'assurera de prendre un *spinlock* sur le nœud de l'arbre unifié avant de tester la présence d'un *mapping* distant, et de le relâcher immédiatement après sa mise-à-jour.

Deux derniers détails doivent encore être réglés : d'une part, la mise à jour de l'arbre unifié peut entraîner des allocations mémoire (allocation d'un nœud de l'arbre), ce qui est ingérable à l'intérieur d'une fonction spécialisée, d'autre part, cette mise-à-jour peut nécessiter de "rebalancer" l'arbre. Cette modification doit faire l'objet d'une section critique en lecture et écriture (un *thread* ne doit pas parcourir l'arbre pendant qu'il est "rebalancé"). Pour régler le premier point, on peut s'inspirer du code déjà existant qui alloue par avance les niveaux intermédiaires de la table des pages dans `handle_mm_fault()`, c'est également là que nous choisissons d'allouer notre nouveau nœud de l'arbre. Cette fonction retournera également un pointeur sur le nœud alloué (ou déjà existant), ce qui permettra de régler le second point, en effet, en passant ce pointeur aux fonctions spécialisées, on peut ainsi éviter de passer par l'arbre, ce qui permet de ne placer que l'allocation en section critique.

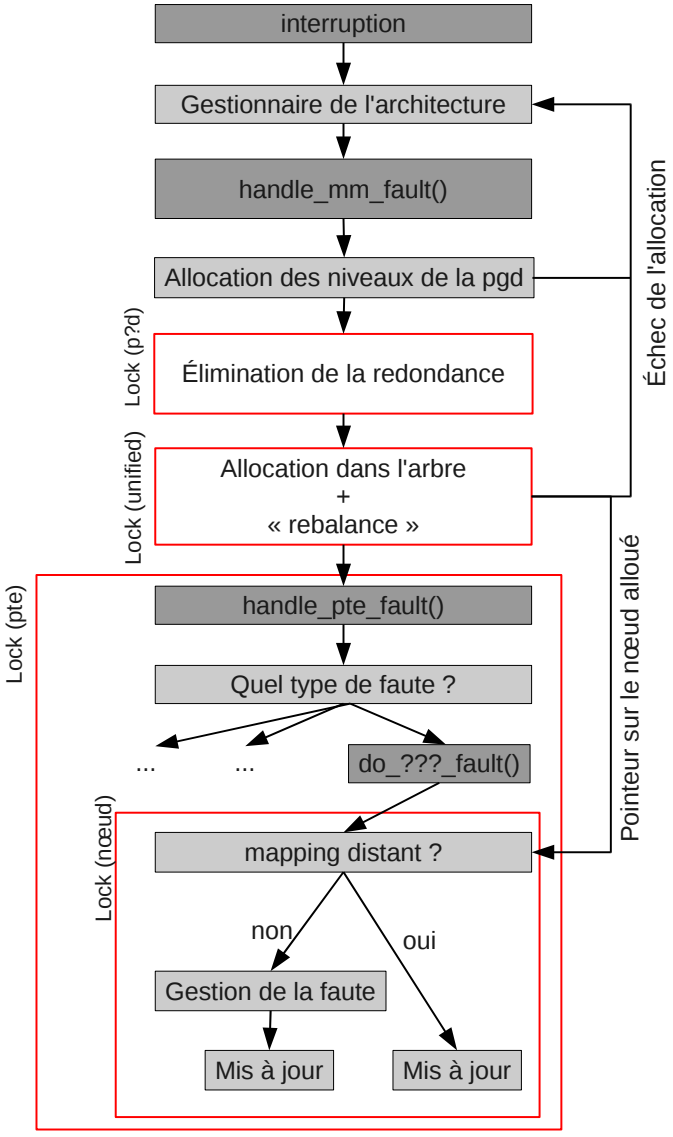


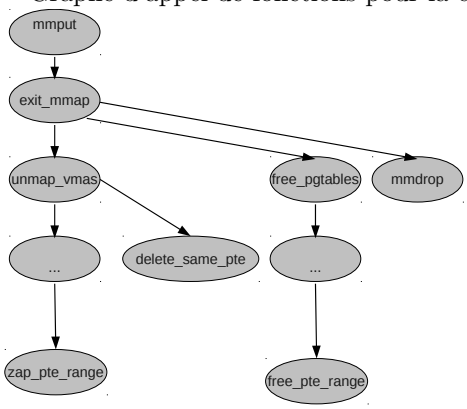

FIGURE 4.6 – Déroulement d'une faute de page sous Linux

## 4.7 Désallocation

Nous avons vu précédemment comment se déroule la désallocation dans le noyau actuel. Un même processus peut avoir plusieurs tables des pages avec potentiellement des `pte` identiques. Donc si nous essayons de désallouer séquentiellement les tables de pages, nous risquons de supprimer plusieurs fois un même `pte`. Pour éviter les incohérences nous avons décidé que dans le cas où deux tables des pages possèdent un même `pte` nous n'en garderons qu'un. Pour cela nous avons besoin de parcourir le `unified` et dans le cas où plusieurs pte d'un `physicals` sont identique nous appellons la fonction `clear_pte()` (elle met à *none* le `pte`) sur la table unifiée et la table des pages pour n'en garder qu'un.

Nous avons placé la suppression des mêmes `pte` dans une sous fonction de `unmap_vmas()` pour avoir la certitude que la suppression de certain `pte` n'aura pas d'impact sur l'exécution. Cette opération nous permet d'appliquer les fonctions de désallocation du noyau sur les différentes tables des pages sans que cela ne pose de problème.

FIGURE 4.7 – Graphe d'appel de fonctions pour la désallocation



# Chapitre 5

# Conclusion

Ce PSAR a été l'occasion de créer des modèles représentant le problème du placement de mémoire dans les architectures NUMA. Au niveau de l'implémentation, une couche d'abstraction a été définit. Toutefois en raison du delai imparti, les points suivant non pas pu être traité, l'inclusion de meilleurs heuristiques dans les algorithmes de placement, les déplacements de pages, la copie et synchronisation des pages. Malgrès cela nos travaux peuvent servir de base pour des travaux ultérieurs.

# Bibliographie


[1] K. Apt, M. Wallace, *Constraint Logic Programming Using ECLiPSe*. Cambridge University Press, 2007

[2] G. Audemard, L. Simon, *Predicting Learnt Clauses Quality in Modern SAT Solvers*, 2009

[3] G. Muller, J. Lawall, X. Lorca, F. Hermenier, J-M. Menaud, *Entropy : a Consolidation Manager for Clusters*, 2009

[4] E. Berger, T. Liu , *SHERIFF : Precise Detection and Automatic Mitigation of False Sharing*, 2011

[5] J-P. Lozi, G. Thomas, J. Lawall, F. David, G. Muller, *Remote Core Locking : Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications*, 2011

[6] E. Berger, T. Liu, C. Curtsinger, *DThreads : Efficient Deterministic Multithreading*, 2011

[7] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, M. Roth, *Traffic Management : A Holistic Approach to Memory Placement on NUMA Systems*, 2013

[8] D. Bovet, M. Cesati, *Understanding the Linux Kernel, Third Edition*, 2005

[9] R. Love, *Linux Kernel Development (3rd Edition)*, 2010

[10] M. Gorman, *Understanding the Linux Virtual Memory Manager*, 2004

[11] A. Fedorova, S. Blagodurov, *User-level scheduling on NUMA multicore systems under Linux*, 2011

[12] P. Drongowski , *Instruction Based Sampling : A New Performance Analysis Technique for AMD Family 10h Processors*, 2007

[13] Advanced Micro Devices, *AMD64 Architecture Programmer's Manual Volume 2 : System Programming*, 2012

[14] J. Levon, *OProfile Manual*, `http://oprofile.sourceforge.net/doc/index.html`, 2000-2004

[15] S. N. Sivanandam, S. N. Deepa, *Introduction to Genetic Algorithms*, 2007

[16] J. LeFlohic, *Genetic Algorithms Tutorial*, `http://www-cs-students.stanford.edu/~jl/Essays/ga.html`, 1999

[17] *Implémentation et mise à l'épreuve d'un algorithme génétique*, http://ithel.free.fr/index.php/2011/02/01/implementation-et-mise-a-lepreuve-dun-algorithme-genetique/, 2011