Compas'2016 : Parallélisme/ Architecture / Système
Lorient, France, du 5 au 8 juillet 2016

Une bibliothèque portable de profilage mémoire bas niveau

Manuel Selva[1], Lionel Morel[1], Kevin Marquet[1]

[1]Univ Lyon, Insa Lyon, Inria, CITI
F-69621 Villeurbanne, France
prénom.nom@insa-lyon.fr

Résumé
Pour suivre l'augmentation du nombre de cœurs intégrés au sein d'une même machine, le sous-système mémoire des architectures multi-cœurs devient de plus en plus complexe. Cette complexité induit des besoins en terme de profilage, afin de permettre aux développeurs une meilleure compréhension de l'usage que font leurs programmes du sous-système mémoire. Les processeurs modernes intègrent des mécanismes qui permettent de construire des outils logiciels répondant à ces besoins. Concernant le profilage mémoire, de nombreux processeurs fournissent des moyens de suivre le trafic mémoire et d'échantillonner les accès en lecture et en écriture vers la mémoire. Néanmoins, ces mécanismes sont très complexes à utiliser et sont spécifiques à chaque micro-architecture. Afin de simplifier l'utilisation de ces mécanismes matériels, nous proposons `numap`, une bibliothèque dédiée au profilage du sous-système mémoire des architectures multi-cœurs modernes. `numap` est portable sur de nombreuses micro-architectures et fournit une interface simple et claire permettant de construire des outils de profilage.

1. Introduction

L'augmentation du nombre de cœurs partageant de la mémoire au sein d'une même machine conduit au fameux *memory wall*. Le sous-système mémoire ne peut pas satisfaire de façon efficace les requêtes simultanées provenant des différents cœurs et devient donc un goulot d'étranglement important. Pour faire face à ce problème, de nombreuses modifications à l'architecture des systèmes mémoires ont été proposées. Les systèmes récents contiennent notamment de nouveaux niveaux de cache et parfois plusieurs mémoires distinctes (architectures Non Uniform Memory Access, i.e., NUMA). Ces mémoires distinctes sont néanmoins vues par les cœurs comme une seule et unique mémoire. Pour pouvoir exploiter de manière optimale ces nouvelles architectures mémoires, le logiciel doit prendre les bonnes décisions [1, 7]. Autrement dit, le logiciel doit avoir connaissance de l'architecture mémoire pour l'exploiter au mieux.
Cette connaissance de l'architecture mémoire n'est néanmoins pas une condition suffisante pour exploiter pleinement les performances du matériel. En effet, le dynamisme des systèmes d'exploitation modernes et des applications qu'ils exécutent conduit à une forme de non déterminisme concernant l'utilisation de la hiérarchie mémoire. Les développeurs ont donc besoin d'outils de profilage afin de comprendre en détails comment le logiciel interagit avec la mémoire et afin d'identifier les goulots d'étranglement. Les processeurs modernes proposent des mécanismes matériels, souvent appelés *Performance Monitoring Unit* (PMU) afin de répondre à

ces besoins de profilage. Nous trouvons aujourd'hui une PMU dans tous les processeurs Intel, AMD et ARM. Concernant la mémoire, la PMU permet d'effectuer un échantillonnage des accès mémoire ainsi que de compter le nombre de requêtes servies par les contrôleurs mémoires. La bande passante mémoire peut être calculée en comptant ces requêtes. Concernant l'échantillonnage, de nombreuses PMUs permettent de récupérer, pour chaque échantillon, des informations concernant la source de l'accès mémoire (le pointeur d'instruction initiant la requête mémoire), le niveau dans la hiérarchie mémoire ou la donnée est trouvée ainsi que la latence de l'accès.
L'utilisation de ces mécanismes de profilage mémoire est une tâche très complexe car elle concerne des mécanismes du plus bas niveau. Il faut notamment écrire du code dépendant de la micro-architecture visée afin d'accéder aux registres dits *Model Specific Registers* (MSR) permettant de configurer le profilage matériel. Ce code de configuration peut être différent même entre deux micro-architectures du même constructeur.
Du point de vue logiciel, la complexité des nouvelles architectures parallèles a conduit à l'émergence de nouveaux modèles de programmation intégrant directement la notion de concurrence. Les outils de profilage ont besoin de connaître le modèle de programmation utilisé pour écrire les applications. En effet, il faut être capable d'associer les informations de profilage bas niveau fournies par la PMU aux objets présents dans le modèle de programmation afin de donner des informations pertinentes au développeur. Une abstraction des mécanismes matériels de profilage mémoire est requise afin de construire ces informations à haut niveau. Celle-ci permettrait de séparer les deux tâches complexes que sont l'utilisation de la PMU et la corrélation des échantillons avec le modèle de programmation.
La contribution principale de ce travail est la bibliothèque `numap` (Non Uniform Memory Access architectures Profiling) dédiée au profilage mémoire. `numap` abstrait les mécanismes de profilage mémoire communs aux PMUs de différents processeurs en fournissant une interface simple et flexible. `numap` est une bibliothèque open-source [1] supportant d'ores et déjà de nombreux processeurs Intel. Bien que `numap` ait été conçue pour profiler des architectures NUMA, elle peut également être utilisée pour profiler des architectures mémoire uniformes.
La suite de ce papier est organisée de la façon suivante. La Section 2 décrit la PMU ainsi que son utilisation. La Section 3 présente les travaux existants concernant cette utilisation. La Section 4 décrit l'interface et la mise en oeuvre de la bibliothèque proposée. Un cas d'usage de `numap` est ensuite présenté en Section 5. Enfin, la Section 6 conclut ce travail et présente les perspectives.

2. Contexte

La PMU fournit un moyen de caractériser l'utilisation faite du matériel à l'aide de compteurs de performance. Ceux-ci sont configurés par le logiciel afin de compter certains événement particuliers parmi un grand nombre de possibilités. Ces compteurs sont localisés au niveau des cœurs ou au niveau des contrôleurs mémoires. Concernant les cœurs, la PMU permet par exemple de compter le nombre d'instructions flottantes exécutées ou le nombre de *cache miss* de niveau 1. Concernant la mémoire, le nombre de lectures ainsi que le nombre d'écritures peuvent être comptées.
En plus de ce mode qui permet de compter des événements, la plupart des PMUs fournissent un moyen de faire de l'échantillonnage des événements se situant au niveau des cœurs. La technique permettant de faire cet échantillonnage est appelée *Precise Event-Based Sampling* (PEBS) pour les processeurs Intel [5] et *Instruction-Based Sampling* (IBS) [4] pour les processeurs AMD.

1. https://github.com/numap-library/numap

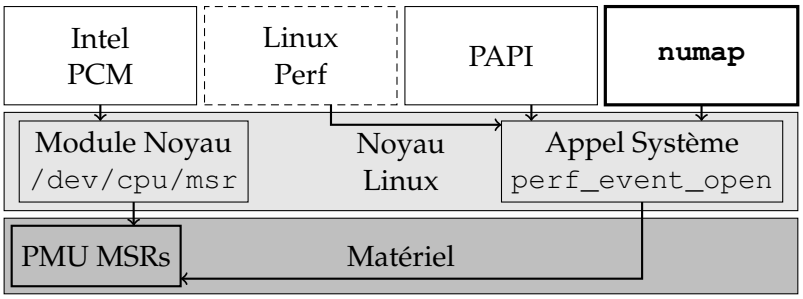

FIGURE 1 – Différentes façons d'accéder à la PMU. Au dessus de Linux, les MSRs de la PMU peuvent être programmés via `perf_event_open` ou via un module noyau. Contrairement à Intel PCM et PAPI, Linux perf n'est pas une bibliothèque mais un outil en ligne de commande.

Sur des architectures Intel, l'échantillonnage permet de sauvegarder des informations détaillées sur un évènement toutes les `X` occurrences de ce type d'évènement particulier (`X` étant la fréquence d'échantillonnage spécifiée par le logiciel). Sur des architectures AMD, l'échantillonnage concerne toutes les instructions indépendamment du type d'évènement.
L'utilisation des compteurs de performance requiert une connaissance très fine de l'architecture visée ainsi que l'écriture de code très bas niveau. La PMU est accessible via des MSRs qui ne peuvent être écrits qu'en mode superviseur. La Figure 1 illustre les deux principales façons d'accéder à ces registres dans un système Linux. Tout d'abord, un module noyau tel que `/dev/cpu/msr` peut être utilisé. Ce module permet d'accéder aux registres de configuration de la PMU en mode superviseur mais ne fournit aucune abstraction. Ensuite, l'appel système `perf_event_open` peut être utilisé. Il a été introduit dans le noyau pour le développement de l'outil en ligne de commande Linux Perf [13]. Cet appel système fournit un premier niveau d'abstraction pour accéder à la PMU. Il permet d'abstraire l'écriture au bit près des MSRs pour démarrer et arrêter le profilage matériel. Il reste néanmoins beaucoup de détails à préciser pour le développeur souhaitant utiliser la PMU.
Les principales difficultés consistent à trouver quels évènements compter ou échantillonner, à définir correctement tous les paramètres nécessaires pour `perf_event_open`, et enfin à effectuer plusieurs appels système en fonction du nombre de threads que l'on souhaite profiler et du nombre de cœurs présents. La page d'aide de `perf_event_open` longue de 1583 lignes reflète la complexité de cet appel système [2].
En plus d'être très complexe à utiliser, l'utilisation de `/dev/cpu/msr` et de `perf_event_open` requièrent des paramètres non portables qui sont spécifiques à chaque micro-architecture.

3. Travaux Connexes

Pour faciliter l'accès à la PMU, des bibliothèques de plus haut niveau ainsi que des outils en ligne de commande ont été proposés.
La bibliothèque Intel PCM [12], disponible pour Linux et Windows, permet d'accéder à la PMU. Néanmoins, elle ne propose pas d'abstraction concernant l'échantillonnage mémoire et comme son nom l'indique, n'est disponible que pour des processeurs Intel. PAPI [3] est une bibliothèque dont l'objectif est de simplifier l'utilisation de la PMU sur de nombreuses architectures. PAPI propose une abstraction pour tous les événements communs aux architectures suppor-

2. http://man7.org/linux/man-pages/man2/perf_event_open.2.html

	API	*Outil*	*Basé sur*	*Portable*	*Echantillonage Mémoire*
Intel PCM	Oui	Non	`/dev/cpu/msr`	Non	Non
PAPI	Oui	Non	`perf_event_open`	Oui	Non
numap	Oui	Non	`perf_event_open`	Oui	Oui
Linux Perf	Non	Oui	`perf_event_open`	Oui	Oui
MemProf	Non	Oui	Module ad-hoc	Non	Oui
HPCToolkit	Non	Oui	Module ad-hoc	Oui	Oui
ScaAnalyzer	Non	Oui	Module ad-hoc	Oui	Oui

TABLE 1 – Bibliothèques dédiées à la PMU et outils utilisant l'échantillonnage mémoire. Les bibliothèques fournissent une API dédiés à du code alors que les outils fournissent une interface utilisateur.

tées, de telle sorte que le développeur ait simplement à dire "je veux compter le nombre de cache miss de niveau 1". Malheureusement, comme Intel PCM, PAPI ne propose pas d'abstraction concernant l'échantillonnage mémoire. De plus, pour compter les accès mémoires en utilisant PAPI, le développeur doit connaître l'identifiant exact de l'évènement à utiliser pour chaque micro-architecture visée. Un travail très récent [10] propose d'étendre PAPI avec le support de l'échantillonnage. La différence principale avec notre proposition concerne l'interface exposée au code utilisateur. Contrairement aux auteurs de ce travail, nous pensons qu'une interface avec un haut niveau d'abstraction permet au développeur de se focaliser sur l'analyse des données et non sur la manière d'obtenir les échantillons. De plus, à notre connaissance, cette extension à PAPI n'est pas encore open source, et son intégration officielle dans PAPI n'est pas encore prévue.

De nombreux outils de profilage ainsi que des mécanismes d'adaptation dynamiques basés sur l'échantillonnage des accès mémoires ont été proposés récemment. Ces outils ont nécessité un effort important de développement pour supporter une ou différentes micro-architectures, et ne sont donc souvent pas portables. Tous les outils mentionnés ci-dessous, à l'exception de Linux Perf, bénéficieraient de `numap` afin de simplifier leur développement et de les rendre portables.

Linux Perf [13] est un outil en ligne de commande capable de faire de l'échantillonnage mémoire. Cet outil est portable sur tous les architectures supportées par Linux. Perf se base sur `perf_event_open` et le code dépendant de l'architecture visée est écrit par les développeurs du noyau. Cet outil très complexe est pensé pour être utilisé par un utilisateur. La seule interface fournie est la ligne de commande. Il est donc impossible de construire des outils de profilage de plus haut niveau au dessus de Perf. Memphis [11] et MemProf [6] permettent d'identifier les accès mémoires distants sur machines NUMA. Ils ne fonctionnent que sur AMD et ont requis le développement de modules noyau complexes pour utiliser la fonctionnalité IBS des processeurs AMD [4]. Carrefour [2] étend MemProf afin d'adapter dynamiquement le placement des données fait par le noyau Linux. Comme MemProf, Carrefour a nécessité des développements complexes pour ne fonctionner que sur des processeurs AMD. HPCToolkit est un autre outil qui a récemment été étendu pour identifier les problèmes de performance liés à la mémoire [8]. Ce profileur est disponible pour de nombreux processeurs. Il se base sur PAPI pour les fonctionnalités non liées à la mémoire, et sur du code dépendant de la micro-architecture pour l'échantillonnage mémoire. ScaAnalyzer [9] est une autre extension à HPCToolkit se basant sur l'échantillonnage mémoire.

```
// Init sampling parameters
int init_samp_session(struct samp_session *s, int nbthreads, int freq, int nbpages);

// Start and stop memory sampling
int samp_read_start (struct samp_session *s);
int samp_read_stop  (struct samp_session *s);
int samp_write_start(struct samp_session *s);
int samp_write_stop (struct samp_session *s);

// Analyze the samples
int print_rd(File *f, struct samp_session *s);
int print_wr(File *f, struct samp_session *s);
int cpy_samples(struct samp_session *s, struct samp **dest, int *nbsp);

// Init counting parameters
int init_count_session(struct count_session *ses, int nb_nodes, int *nodes);

// Start and stop counting memory traffic
int count_read_start (struct count_session *s);
int count_read_stop  (struct count_session *s);
int count_write_start(struct count_session *s);
int count_write_stop (struct count_session *s);

// Get count value
long get_count(struct count_session *s);

// Get human readable error message
const char *error_message(int error);
```

FIGURE 2 – API de `numap`. `numap` propose des fonctions pour démarrer/arrêter l'échantillonnage des accès mémoires et du comptage des requêtes vers la mémoire.

Le Tableau 1 résume la liste des différentes bibliothèques dédiées à la PMU ainsi que les outils de profilage existants utilisant l'échantillonnage mémoire.

4. `numap`

Nous proposons donc la bibliothèque `numap` pour simplifier l'utilisation de la PMU afin de faire de l'échantillonnage des accès mémoires et de mesurer la bande passante mémoire. En choisissant automatiquement la bonne configuration pour le matériel sous-jacent, `numap` propose un moyen portable d'accéder au profilage mémoire bas niveau fourni par la PMU.
Les fonctions de l'interface de programmation proposée par `numap` sont présentées sur la Figure 2. Ces fonctions permettent de compter les requêtes mémoire et de générer des échantillons d'accès mémoire afin de pouvoir analyser le comportement des applications. `numap` différencie les accès mémoires en lecture et en écriture à la fois pour le comptage et l'échantillonnage. Toutes les fonctions, à l'exception de celles concernant la gestion des erreurs, ont une structure type `session` comme paramètre qui permet de fournir les résultats à l'appelant et qui contient les détails nécessaires à la configuration de la PMU. Cette structure diffère entre les fonctions liées à l'échantillonnage et celles liées au comptage.
Pour l'échantillonnage, cette structure s'appelle `samp_session`. Elle contient une liste d'identifiants de threads. Le code utilisateur doit remplir cette liste afin de spécifier quels threads il veut profiler avant de passer la structure à la fonction démarrant l'échantillonnage. La struc-

ture `samp_session` est également utilisée pour spécifier la fréquence d'échantillonnage et le nombre de pages à utiliser pour sauvegarder les résultats. La version actuelle de `numap` permet seulement de sauvegarder des échantillons tant qu'il reste de la place dans ces pages. La fonction `init_samp_session` permet de définir ces paramètres.
Pour le comptage, la structure se nomme `count_session`. Elle contient la liste des nœuds NUMA à profiler. Comme pour l'échantillonnage, cette liste doit être remplie par le code utilisateur avant d'utiliser la fonction qui démarre le comptage. La structure contient aussi un descripteur de fichier pour chaque noeud NUMA profilé. Une fois le comptage arrêté, le code utilisateur récupère la valeur des compteurs en utilisant la fonction `get_count` qui utilise ce descripteur.
`numap` propose aussi des fonctions offrant au code utilisateur un accès simple aux échantillons générés. Ces fonctions prennent aussi en entrée la structure `samp_session`. Pour récupérer la liste des échantillons, `numap` propose la fonction `samp_cpy`. Cette fonction produit une liste d'échantillons allouée dans le tas du processus appelant. Son adresse et sa taille sont respectivement renvoyées via les pointeurs `dest` et `nbsp`. Chaque élément de la liste est de type `struct samp`. Cette structure contient l'adresse de l'instruction qui a généré l'échantillon, l'adresse visée par l'accès mémoire, la latence (weight) de celui-ci et le niveau mémoire qui a servi l'accès. Ces informations sont fondamentales pour n'importe quel outil souhaitant construire des mécanismes de profilage ou d'adaptation dans un contexte particulier. La fonction `samp_cpy` masque tous les détails techniques nécessaires pour lire les échantillons générés par `perf_event_open`. La bibliothèque fournit aussi des fonctions pour écrire les échantillons soit dans un fichier soit sur la sortie standard.

5. Cas d'Usage

`numap` a été initialement créée pour profiler des applications de type *dataflow* exécutées sur des architectures NUMA [14]. Une application dataflow est décrite par un graphe. Chaque nœud du graphe, appelé *acteur*, représente une entité indépendante produisant des données en sortie en fonction de données en entrée. Les arrêtes du graphe, ayant un comportement de type First-In-First-Out (FIFO) spécifient les dépendances de données entre les acteurs. Ce découpage explicite entre calcul et communication permet notamment d'exécuter des applications dataflow de façon parallèle sur des machines multi-cœurs. Dans le contexte dataflow, il est crucial de profiler l'utilisation mémoire afin de placer correctement sur l'architecture multi-cœurs les acteurs ainsi que les buffers utilisés pour qu'ils communiquent. `numap` permet d'obtenir facilement des échantillons d'accès mémoire pour des application dataflow et ce de façon portable.
Nous utilisons `numap` pour profiler des applications dataflow écrites avec le langage RVC-CAL et compilées avec Open RVC-CAL Compiler (Orcc)[3]. Ce compilateur est un compilateur source-to-source qui produit du code C multi-thread ensuite compilé de manière classique. La Figure 3 montre la structure générale de notre profileur telle que nous l'avons implémentée dans Orcc. Le compilateur Orcc a été modifié afin de générer des appels aux fonctions `numap`. Ces appels démarrent et arrêtent l'échantillonnage mémoire à chaque fois qu'un thread est créé puis détruit. Les échantillons récoltés sont enregistrés dans des fichiers pour ensuite faire une analyse hors ligne. Cette analyse hors ligne associe les échantillons mémoires aux acteurs et aux FIFOs du modèle dataflow. Cette association est nécessaire pour pouvoir fournir des informations pertinente au développeur de l'application ou au développeur du compilateur

3. http ://orcc.sourceforge.net/

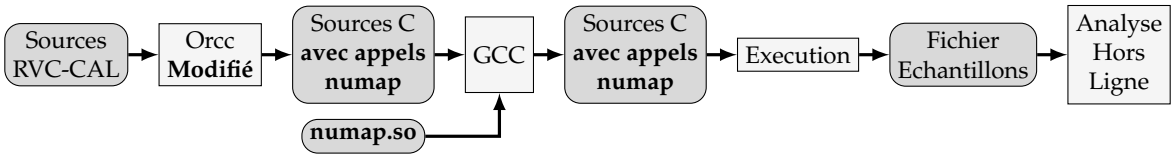

FIGURE 3 – Profileur mémoire RVC-CAL basé sur `numap`. Le compilateur Orcc est modifié pour générer du code utilisant `numap`. Les échantillons récoltés par `numap` pendant l'exécution sont sauvegardés puis analysés.

sur la base desquelles l'exécution du programme peut-être optimisée.
Le Tableau 2 présente les résultat de profilage du décodage d'une vidéo H.265 par un décodeur RVC-CAL et exécuté sur une machine NUMA disposant de deux processeurs Intel Westmere hexa-cœurs. La fréquence d'échantillonnage est de 20000 et seul les accès de type lecture (limitation de la PMU du matériel) sont rapportés sur ce tableau. A l'aide de cette information, il est possible de prendre des décisions de placement concernant les acteurs liés au FIFOs (FIFO-18 et FIFO-2) les plus coûteuses en terme de latence mémoire ainsi que le placement concernant les structures internes utilisées pour respectée la sémantique dataflow (schedWait et sched).

Source	Coût(%)
schedWait	7.88
sched	3.92
FIFO-18	3.86
FIFO-2	3.58

TABLE 2 – Structures dataflow les plus coûteuses en terme de latence mémoire

6. Conclusion et Perspectives

Ce travail propose la bibliothèque `numap` pour faciliter l'utilisation du profilage mémoire fourni par la PMU de nombreux processeurs récents. `numap` est à la fois portable et simple à utiliser. L'utilisabilité repose sur une interface simple masquant l'essentiel de la complexité de l'utilisation de la PMU. Nous sommes convaincus que de telles abstractions sont nécessaires afin de faciliter la création d'outils de profilage et de mécanismes d'adaptation pour les architectures de demain de façon efficace et portable.
Nous travaillons actuellement au portage de `numap` sur des processeurs AMD. Cette tâche requiert une bonne compréhension du mécanisme IBS d'AMD. Néanmoins, comme `perf_event_open` supporte déjà ce mécanisme [4], et grâce à l'expérience acquise pour le développement de `numap` sur des processeurs Intel, cette tâche consiste essentiellement à répéter le travail déjà effectué. Indépendemment de la complexité d'utiliser l'IBS, l'interface de `numap` reste inchangée lorsque l'on ajoute le support de processeurs AMD. En effet, les informations contenues dans les échantillons `numap` sont toutes fournies par l'IBS. Enfin, nous pensons que l'abstraction proposée par `numap` pourrait et même devrait être intégrée dans la bibliothèque

4. https ://lwn.net/Articles/490418

PAPI [3] déjà très largement utilisée. Nous sommes actuellement en discussion avec les développeurs PAPI afin de voir comment intégrer et rassembler l'interface de `numap` ainsi que celle proposé par Lopez et al. [10] très récemment.

Bibliographie

1. Blagodurov (S.), Zhuravlev (S.), Dashti (M.) et Fedorova (A.). – A case for numa-aware contention management on multicore systems. – In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference, USENIXATC'11*, pp. 1–1, Berkeley, CA, USA, 2011. USENIX Association.
2. Dashti (M.), Fedorova (A.), Funston (J.), Gaud (F.), Lachaize (R.), Lepers (B.), Quema (V.) et Roth (M.). – Traffic management : A holistic approach to memory placement on numa systems. – In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13*, pp. 381–394, New York, NY, USA, 2013. ACM.
3. Dongarra (J.), London (K.), Moore (S.), Mucci (P.) et Terpstra (D.). – Using papi for hardware performance monitoring on linux systems. – In *In Conference on Linux Clusters : The HPC Revolution, Linux Clusters Institute*, 2001.
4. Drongowski (P. J.). – Instruction-based sampling : A new performance analysis technique for amd family 10h processors, 2007.
5. Intel Corporation. – *Intel® 64 and IA-32 Architectures Software Developer's Manual*. – Intel Corporation, January 2015.
6. Lachaize (R.), Lepers (B.) et Quéma (V.). – Memprof : A memory profiler for numa multicore systems. – In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference, USENIX ATC'12*, pp. 5–5, Berkeley, CA, USA, 2012. USENIX Association.
7. Lepers (B.), Quema (V.) et Fedorova (A.). – Thread and memory placement on numa systems : Asymmetry matters. – In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pp. 277–289, Santa Clara, CA, juillet 2015. USENIX Association.
8. Liu (X.) et Mellor-Crummey (J.). – A tool to analyze the performance of multithreaded programs on numa architectures. – In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'14*, pp. 259–272, New York, NY, USA, 2014. ACM.
9. Liu (X.) et Wu (B.). – Scaanalyzer : A tool to identify memory scalability bottlenecks in parallel programs. – In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC'15*, pp. 47 :1–47 :12, New York, NY, USA, 2015. ACM.
10. Lopez (I.), Moore (S.) et Weaver (V.). – A prototype sampling interface for papi. – In *Proceedings of the 2015 XSEDE Conference : Scientific Advancements Enabled by Enhanced Cyberinfrastructure, XSEDE '15*, pp. 27 :1–27 :4, New York, NY, USA, 2015. ACM.
11. McCurdy (C.) et Vetter (J.). – Memphis : Finding and fixing numa-related performance problems on multi-core platforms. – In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pp. 87–96, March 2010.
12. https://software.intel.com/en-us/articles/intel-performance-counter-monitor, 2015. (Retrieved 2016-02-19).
13. https://perf.wiki.kernel.org/index.php/. (Retrieved 2016-02-19).
14. Selva (M.). – *Performance Monitoring of Throughput Constrained Dataflow Programs Executed On Shared-Memory Multi-core Architectures*. – Theses, Institut National des Sciences Appliquées de Lyon, juillet 2015.