Ipanema : un langage dédié pour le développement d'ordonnanceurs multi-coeur sûrs

Redha Gouicem, Julien Sopena, Julia Lawall, Gilles Muller, Baptiste Lepers, Willy Zwaenepoel, Jean-Pierre Lozi, Nicolas Palix

Compas'2017 : Parallélisme/ Architecture / Système
Sophia Antipolis, France, du 27 au 30 juin 2017

Ipanema : un langage dédié pour le développement d'ordonnanceurs multi-coeur sûrs

Redha Gouicem*, Julien Sopena*, Julia Lawall*, Gilles Muller*, Baptiste Lepers^Λ, Willy Zwaenepoel^Λ, Jean-Pierre Lozi^◇, Nicolas Palix^Ω

* Sorbonne Universités, Inria, LIP6, *prenom.nom@lip6.fr*
^Λ EPFL, *prenom.nom@epfl.ch*
^◇ Université Nice Sophia-Antipolis, *jplozi@unice.fr*
^Ω Université Grenoble Alpes, *nicolas.palix@univ-grenoble-alpes.fr*

Résumé
L'ordonnanceur est un point clé au niveau des performances fournies par un système d'exploitation. Il est cependant difficile d'écrire une politique d'ordonnancement au sein d'un système d'exploitation actuel. Nous proposons Ipanema, un langage dédié permettant d'écrire des politiques d'ordonnancement multi-coeur sûres.

Mots-clés : Ordonnancement, Systèmes d'exploitation, Multi-coeur

---

1. Introduction

L'ordonnanceur du système d'exploitation occupe une place prépondérante dans la gestion des ressources de calcul des machines multi-coeur. Il est cependant fréquent que ce dernier ne tire pas entièrement partie des ressources de calcul disponibles. Ainsi, Lozi et al. [10] ont démontrés que l'ordonnanceur utilisé par défaut dans le système d'exploitation de Linux, le CFS (*Completely Fair Scheduler*), est victime d'erreurs de conception qui conduisent à laisser des coeurs inoccupés alors que des tâches sont en attente. Ces défaillances vont à l'opposé du comportement attendu cherchant à minimiser le nombre de coeurs inactifs (*work conservation*). Elles se traduisent par des performances suboptimales aussi bien en termes de temps d'exécution que de consommation énergétique.
La présence de telles erreurs s'explique en grande partie par la complexité actuelle de CFS. En effet, depuis son introduction dans le noyau 2.6.23, son volume de code n'a cessé d'augmenter dû à l'introduction de la gestion de nombreux cas particuliers afin d'assurer de bonnes performances pour de multiples comportements applicatifs. Ce problème présent dans le CFS est inhérent à tout ordonnanceur qui se veut à la fois performant et générique. En effet, il devient rapidement difficile de comprendre le flot d'exécution de l'ordonnanceur, et de garantir manuellement des propriétés d'ordonnancement comme la vivacité, l'équité ou la *work conservation*. Une réponse à cette complexité consiste à construire plusieurs petits ordonnanceurs spécifiques à certains comportements d'applications. Plusieurs travaux de recherche ([14],[3]) proposent ce type d'ordonnanceurs en espace utilisateur afin d'éviter le surcoût de développement dans le noyau Linux.
La limite des approches en mode utilisateur réside dans la perte de performances par rapport aux approches en mode noyau. Or, la complexité du développement noyau rend ces approches

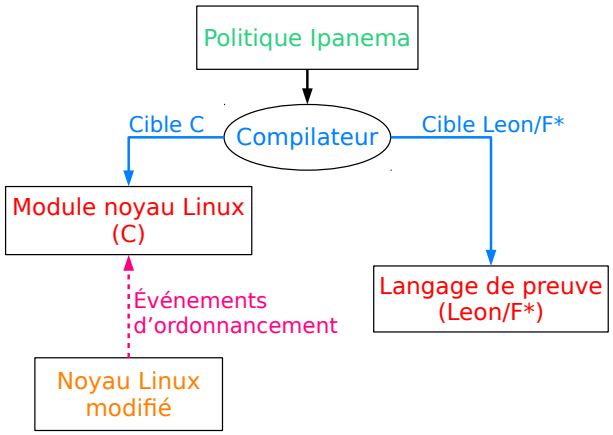


FIGURE 1 – Schéma de compilation d'une politique Ipanema

inadéquates pour de nombreux développeurs. Une façon de masquer cette complexité consiste à utiliser un langage dédié. C'est dans ce sens que nous proposons Ipanema, un langage dédié à l'écriture de politiques d'ordonnancement multi-coeur. La compilation d'une politique produit un module noyau (en langage C) pouvant être insérer dans un noyau Linux modifié (Figure 1). Dans cet article, nous présenterons tout d'abord une étude de l'évolution de CFS, puis nos contributions, à savoir les abstractions d'un ordonnanceur multi-coeur que nous avons définies, ainsi qu'un aperçu de la méthodologie de vérification de propriétés et une première évaluation de nos travaux.
Le reste de cet article sera organisé comme suit : la section 2 présente nos motivations ; la section 3 présente nos contributions ; la section 4 présente les évaluations de nos travaux, et la section 5 nos conclusions et futurs travaux.

2. Complexité de CFS

Dans cette section, nous présentons comment l'évolution de CFS a motivé la proposition d'Ipanema. Dans un premier temps, nous décrivons l'ordonnanceur par défaut de Linux, le CFS. Ensuite, nous étudions son évolution depuis dix ans et sa complexification.

2.1. Motivations

Le *Completely Fair Scheduler* a été introduit dans la version 2.6.23 du noyau Linux en octobre 2007 pour remplacer l'ordonnanceur $O(1)$ [1]. Il s'agit d'une implantation de l'algorithme *weighted fair queueing* [9]. C'est un ordonnanceur réparti, chaque coeur dispose de sa propre file (implanté sous la forme d'un arbre rouge-noir) de tâches à exécuter. Ces arbres sont triés par ordre de temps d'exécution croissant : les tâches les moins exécutées sont donc prioritaires, en vue d'équilibrer le temps d'exécution de toutes les tâches. En effet, le CFS tente de "modéliser un CPU multi-tâches idéal et précis" [11]. Par ailleurs, périodiquement (toutes les 4ms par défaut), un évènement d'équilibrage de charge est déclenché sur tous les coeurs. Cet évènement vise à répartir équitablement la charge générée par les tâches exécutées entre tous les coeurs.
Le rééquilibrage de charge tire parti d'une propriété de CFS, à savoir qu'il est aussi hiérarchique [5]. De manière simple, cela signifie que les coeurs de la machine partageant certaines propriétés (des caches par exemple) sont partitionnés en domaines, et ces domaines sont ensuite partitionnés en groupes de coeurs. L'équilibrage de charge se fait entre domaines, et entre

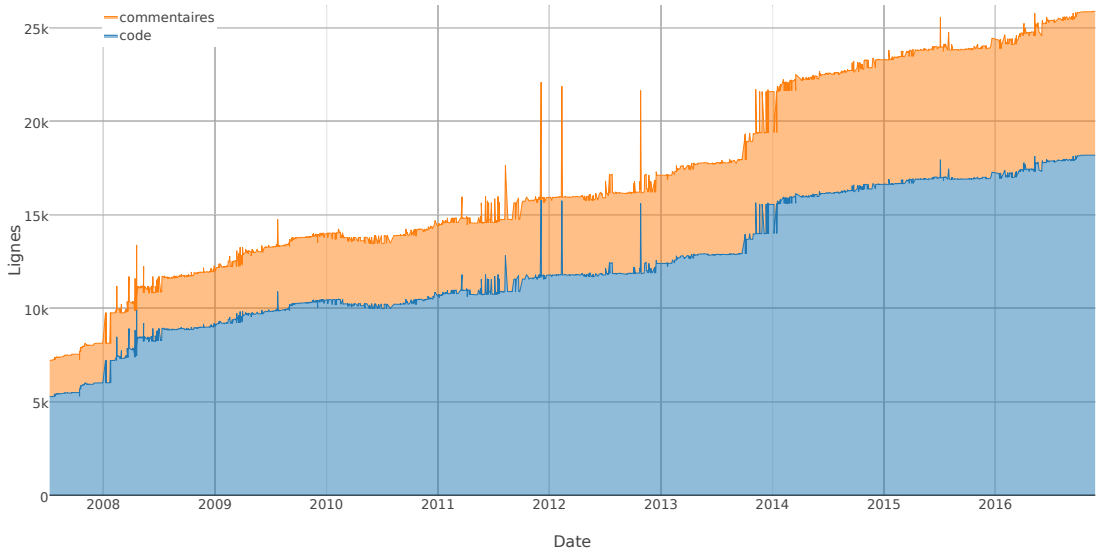


FIGURE 2 – Évolution de la base de code de CFS

Configuration	Nombre de lignes de code
Configuration par défaut	6637
Configuration minimale	1481
Configuration A	3131
Configuration B	4796

FIGURE 3 – Nombre de lignes de code de `fair.c` selon différentes configurations

groupes dans un domaine. Ainsi, sur des architectures de type NUMA, il est possible d'éviter les migrations de tâches vers un coeur placé sur un noeud distant (donc dans un autre domaine) pour optimiser les accès mémoire locaux.

2.2. Évolution de CFS

Le CFS se veut générique et implante donc quantité de cas particuliers afin de répondre aux problèmes de performance dûs à des comportements applicatifs spécifiques. En dix ans, le nombre de lignes de code de l'ordonnanceur a été multiplié par 3,45, atteignant aujourd'hui 18.182 lignes. La Figure 2 montre l'évolution de la base de code de CFS depuis son introduction dans le noyau Linux en 2007. Les accroissements soudains de la taille du code sont dus à l'ajout de fonctionnalités importantes dans l'ordonnanceur. Outre ces ajouts, les corrections de bogues et les optimisations provoquent une croissance lente mais continue du volume de code, et donc de la complexité de CFS.

De plus, il est important de remarquer que le code de CFS ne contient pas un, mais plusieurs ordonnanceurs. En effet, l'utilisation de macros et de conditionnelles sur ces dernières (dans des blocs `#ifndef`) engendre la création de différents fichiers sources lors de l'étape de préprocesseur. Afin d'illustrer ce phénomène, nous listons l'ensemble des macros utilisées de façon conditionnelles du fichier `fair.c` de CFS et générons l'ensemble des 1024 combinaisons de ces macros. Nous appliquons ensuite le préprocesseur sur ce fichier avec chacune de ces combinaisons de macros. La Figure 3 contient le nombre de lignes de plusieurs des fichiers générés (les

configurations A et B sont sélectionnées au hasard), exposant ainsi la variété d'ordonnanceurs présents dans le code de CFS, obtenu simplement en modifiant la configuration de compilation du noyau.
La tendance d'évolution de CFS nous pousse à réfléchir à des solutions alternatives, notamment les ordonnanceurs spécifiques qui peuvent être plus légers et simples car ils n'ont pas pour but d'être génériques et de gérer tous les comportements applicatifs. Cela nous motive donc à proposer un langage dédié permettant de développer ces ordonnanceurs dans le noyau, et non plus en espace utilisateur comme cela était souvent le cas jusqu'à maintenant.

3. Contributions

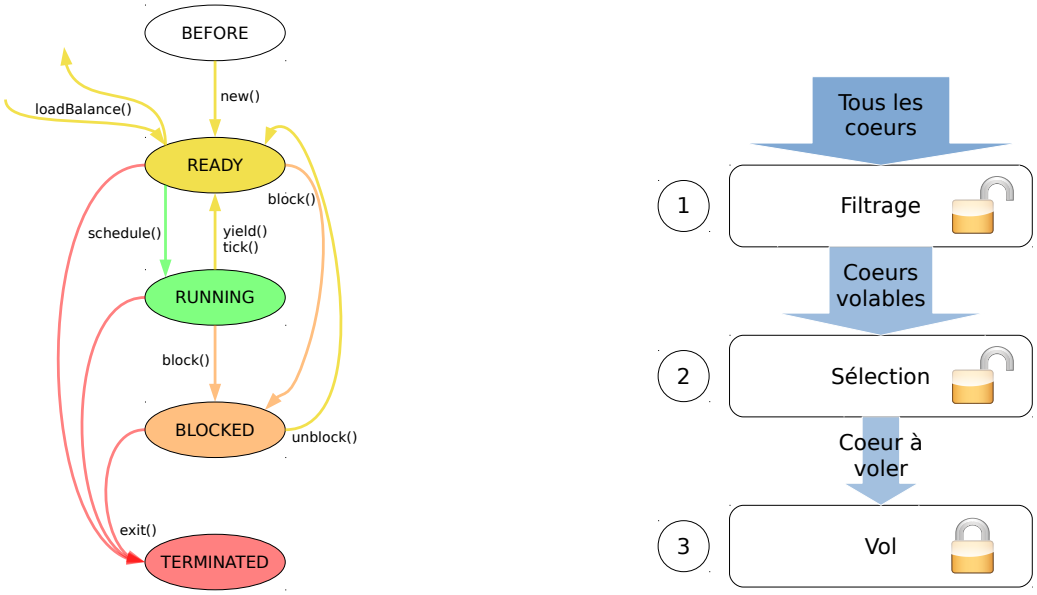
Afin de décrire Ipanema, notre langage dédié, il est impératif d'introduire certaines notions liées aux ordonnanceurs multi-coeur. Nous définissons donc des abstractions concernant l'équilibrage des files locales, puis les vérifications à opérer pour assurer certaines propriétés d'ordonnancement.

3.1. Abstraction d'un ordonnanceur multi-coeur

Nous raisonnons sur des architectures multi-coeur, il est donc nécessaire de choisir un modèle d'ordonnanceur permettant d'éviter les problèmes de contention inhérents à ce type d'architectures. Nous choisissons un modèle d'ordonnanceur réparti, où chaque coeur dispose d'un ordonnanceur local. La charge de ces ordonnanceurs est ensuite équilibrée périodiquement. Ce modèle est utilisé dans la plupart des systèmes d'exploitation (Linux, FreeBSD, Solaris, Windows, etc...) [8].
Pour l'ordonnancement local, nous nous basons sur les abstractions définies par le langage dédié Bossa [13] pour des ordonnanceurs mono-coeur. Il modélise l'ordonnanceur comme un automate. Après sa création, une tâche peut être dans les état suivants :

— RUNNING : la tâche est exécutée sur le processeur ;
— READY : la tâche est prête à être exécutée sur le processeur, en attente dans la file locale ;
— BLOCKED : la tâche est bloquée, en attente d'une ressource ;
— TERMINATED : la tâche a terminé son exécution.

La Figure 4a montre les différents états ainsi que les transitions les reliant. Les transitions sont étiquetées par le nom de l'évènement provoquant cette transition. On voit notamment que l'équilibrage de charge ne peut avoir lieu qu'entre des files de l'état READY de deux coeurs ; ainsi, on évite de voler une tâche en cours d'exécution ou en attente d'une ressource sur un coeur spécifique.
L'équilibrage de la charge est donc la brique supplémentaire pour rendre un ordonnanceur multi-coeur. Un équilibrage consiste en fait en un vol de tâches effectué par un coeur (*voleur*) auprès d'un autre coeur (*volé*). La Figure 4b montre les trois étapes qui composent un vol de tâches. Tout d'abord, le coeur voleur filtre les coeurs volables. Il sélectionne donc un sous-ensemble des coeurs de la machine selon un critère spécifique à la politique d'ordonnancement. Ensuite, parmi ce sous-ensemble de coeurs volables, il choisit le coeur à voler. Ces deux étapes forment la phase de sélection effectuée sans prise de verrou puisque ces opérations sont en lecture seule. La troisième et dernière étape est celle de vol où le coeur voleur essaie de voler de façon optimiste. Cela signifie que si le vol échoue, l'équilibrage est abandonné. Le coeur voleur verrouille donc sa file de tâches puis tente de verrouiller celle du coeur à voler : si ce verrouillage aboutit, le coeur voleur vole un certain nombre de tâches au coeur à voler, selon un critère défini dans la politique ; sinon, l'équilibrage est abandonné.
Ces abstractions nous permettent de faciliter la rédaction d'une politique d'équilibrage pour



(a) Ordonnanceur mono-coeur événementiel (b) Étapes d'un vol de tâches

FIGURE 4 – Abstractions d'un ordonnanceur multi-coeur

l'utilisateur, puisque ce dernier ne se concentre que sur une suite de critères (choix des coeurs, choix des tâches), et pas sur les difficultés d'implantation dans le noyau Linux. De plus, cela met en évidence les sections critiques nécessitant une prise de verrou(s). Ainsi, nous pouvons abstraire ces dernières et les générer automatiquement autour de l'étape de vol, évitant donc au développeur d'avoir une connaissance fine des verrous implantés dans le noyau pour gérer la concurrence des accès aux variables partagées. Cela nous permet également de minimiser le nombre et le temps de prise de verrous, assurant ainsi un meilleur passage à l'échelle.

3.2. Vérification de sûreté

Une partie de la sûreté des politiques générées est assurée par construction par le langage Ipanema. Des règles de typage plus strictes qu'en C sont imposées, notamment en interdisant les ensembles d'objets de types différents ou le transtypage. L'interdiction d'effectuer des transitions invalides dans l'automate de processus (cf. Figure 4a) garantit la cohérence des états d'un processus (par exemple, un processus bloqué sur une entrée-sortie ne peut passer qu'à l'état BLOCKED). De plus, pour éviter de "perdre des tâches", on interdit également le retrait d'une tâche d'un ensemble sans l'ajouter à un autre, évitant ainsi un cas de famine puisque la tâche ne serait alors plus dans aucune file d'attente. On assure également le fait que l'on ne puisse pas accéder à des variables désallouées en interdisant la déclaration de références globales sur des éléments partagés.

La génération des prises et relâchement de verrous par le compilateur Ipanema→C permet d'éviter les interblocages au niveau de l'ordonnanceur. En effet, dans le code C généré, les prises (et relâchement) imbriquées de verrous sont faites suivant un ordre total. Ces dernières n'interviennent qu'en cas de réelle nécessité, comme la modification d'une variable partagée, ou lors de l'exécution d'un évènement modifiant l'état de l'ordonnanceur. Ces prises de verrous, implicites pour le développeur, assurent ainsi la cohérence des variables partagées de la politique.

4. Évaluation

En plus d'assurer certaines propriétés d'ordonnancement ainsi que la sûreté du code C généré, il est nécessaire d'obtenir des performances de fonctionnement proches d'un ordonnanceur implanté "à la main" dans le système d'exploitation. Afin d'évaluer notre contribution, nous implantons une politique simple en Ipanema que nous compilons vers un module en C, puis nous comparons les performances de notre politique par rapport à CFS sur deux applications : *kbuild* (compilation d'un noyau Linux) et *hackbench* (une application stressant l'ordonnanceur en faisant communiquer de nombreux processus par des tubes).

4.1. Configuration de l'évaluation

Les expériences sont effectuées sur une machine équipée d'un processeur Intel Xeon E7-8870 (80 coeurs physiques, 160 logiques) et de 512 Go de RAM avec un Ubuntu Server 16.04. La configuration utilisée pour CFS est celle par défaut. La partie équilibrage de charge de la politique développée en Ipanema (*ipanema_simple*) est présentée en figure 5. Le bloc `steal` (l. 16-29) présente les trois abstractions définies pour l'équilibrage de charge : le *filtrage* selon la différence de charge entre le coeur voleur et le coeur cible ; la *sélection* du coeur à voler parmi les coeurs volables, le premier de l'ensemble ici ; et le *vol* de tâches au coeur à voler tant que la charge entre les deux coeurs n'est pas équilibrée.

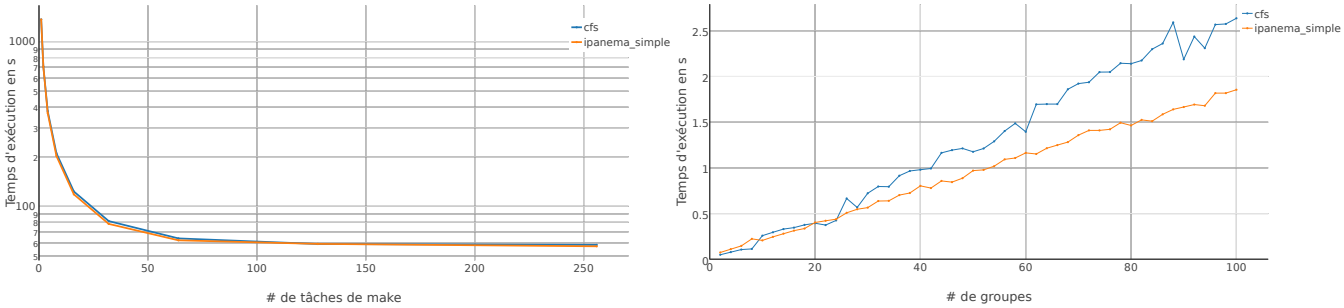
```
1 thread = {
2   int progress, load, last_sched, curr_quanta;
3 }
4
5 core = {
6    threads = {
7       RUNNING thread current;
8       shared READY set<thread> ready:order = {
9          lowest progress
10      };
11      BLOCKED blocked;
12      TERMINATED terminated;
13   }
14   int load = sum(ready.load) +
15              valid(current) ? 1024 : 0;
16   steal = {
17      filter_core(core c) {
18        sum(c.ready.load) > 0 &&
19        c.load-self.load >= 2*min(c.ready.load)
20      }
21      select_core(set<core> stealable_cores) {
22         first(stealable_cores)
23      }
24      steal_thread(core c, thread t) {
25         if (c.load - self.load >= 2*t.load) {
26             t => self.ready;
27         }
28      } until { c.load == self.load }
29   }
30 }
```

FIGURE 5 – Politique *ipanema_simple* écrite en Ipanema

4.2. Résultats

Les applications choisies pour évaluer les performances de notre politique simple par rapport à CFS sont issues d'un article rédigé par le créateur de CFS, Ingo Molnár, pour comparer un nouvel ordonnanceur pour Linux, BFS [12]. La figure 6a montre le temps d'exécution d'une compilation d'un noyau Linux en fonction du nombre de processus utilisé en parallèle. Quel que soit le nombre de processus, notre politique rivalise avec CFS, il n'y a ni amélioration, ni dégradation. La figure 6b montre le temps d'exécution d'une instance de *hackbench* en fonction du nombre de groupes de processus communiquant entre eux, plus le nombre de groupes est élevé, plus le nombre de processus est élevé (1 groupe contient 40 processus). Plus le nombre de processus augmente, plus l'écart de performance s'agrandit. Notre politique est plus performante car elle ne prend pas en compte la topologie de la machine : CFS essaie de garder les processus proches les uns des autres avant de les répartir sur toute la machine pour favoriser le partage des caches, alors que notre politique tente de maximiser le nombre de coeurs utilisés le plus vite possible. Or, dans le cas de *hackbench*, il vaut mieux utiliser toute la machine plutôt

qu'une fraction partageant des caches. On n'évalue cependant pas le surcoût de nos modifications du noyau Linux car pour cela, il faudrait écrire la politique de CFS en Ipanema, ce qui est impossible dans un temps raisonnable compte tenu de la complexité de cette dernière.



(a) *kbuild* (b) *hackbench*

FIGURE 6 – Évaluations de la politique *ipanema_simple*

5. Conclusion et travaux futurs

Les bogues soulevés par Lozi et al. [10] montrent l'importance de l'ordonnanceur pour les performances des applications. La croissance du Completely Fair Scheduler, l'ordonnanceur de Linux se voulant le plus générique possible, montre qu'il est difficile d'allier généricité et simplicité. De nombreux travaux proposent de pallier à cette complexité en développant des ordonnanceurs spécifiques. Ces approches sont cependant difficiles à appliquer dans le noyau Linux, compte tenu de la difficulté de développement dans ce dernier, ces travaux sont donc souvent réalisés en espace utilisateur, en dépit d'une perte de performances. Le principal objectif de nos travaux est donc de permettre le développement d'ordonnanceurs pour le noyau Linux avec une expertise minimale du développement dans le système d'exploitation.

Nous présentons dans ce papier des abstractions pour étendre celles fournies par Bossa aux systèmes multi-coeur. Nous permettons le développement d'ordonnanceurs multi-coeur en espace noyau à l'aide d'un langage dédié, Ipanema, supprimant ainsi la complexité du développement dans le noyau du système d'exploitation. Cette approche permet également de rendre les prises de verrous implicites et sûres, point essentiel dans un système concurrent, et de vérifier des propriétés de sûreté, notamment au niveau du typage. Nous montrons également que sur certaines applications, avec une politique simple, il est possible d'être plus performant que CFS.

De nombreux travaux récents s'attellent à prouver que des composants de systèmes d'exploitation sont corrects par rapport à leur spécification. Amani et al. [2] et Chen et al. [4] prouvent des systèmes de fichiers ; Gu et al. [6] proposent un noyau certifié ; Klein et al. [7] vérifient formellement un noyau. Les abstractions que nous avons défini ont également été pensées dans le but d'effectuer des preuves de certaines propriétés d'ordonnancement (vivacité, équité, *work conservation*), qui seront présentées dans de futurs travaux. L'écriture de ces preuves et leur génération automatique à partir du code écrit en Ipanema (Figure 1) constituent nos travaux en cours.

Bibliographie

1. Josh Aas. Understanding the linux 2.6.8.1 cpu scheduler. https://github.com/bdaehlie/linux-cpu-scheduler-docs/blob/master/linux_cpu_scheduler.pdf, April 2015.
2. Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, et al. Cogent : Verifying high-assurance file system implementations. In *ASPLOS*, pages 175–188, 2016.
3. Christos D Antonopoulos, Dimitrios S Nikolopoulos, and Theodore S Papatheodorou. Scheduling algorithms with bus bandwidth considerations for smps. In *ICPP*, pages 547–554, 2003.
4. Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *SOSP*, pages 18–37, 2015.
5. Jonathan Corbet. Scheduling domains. https://lwn.net/Articles/80911, April 2004.
6. Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos : an extensible architecture for building certified concurrent os kernels. In *OSDI*, 2016.
7. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4 : Formal verification of an os kernel. In *SOSP*, pages 207–220, 2009.
8. Matthias Kohler. Multiple run-queues for bfs. https://lwn.net/Articles/529280/, December 2012.
9. Tong Li, Dan Baumberger, and Scott Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *PPoPP*, pages 65–74, 2009.
10. Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler : a decade of wasted cores. In *EuroSys*, page 1, 2016.
11. Ingo Molnár. Cfs scheduler. https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt.
12. Ingo Molnár. Bfs vs. mainline scheduler benchmarks and measurements. https://lwn.net/Articles/351058, September 2009.
13. Gilles Muller, Julia L Lawall, and Hervé Duchesne. A framework for simplifying the development of kernel schedulers : Design and performance evaluation. In *HASE*, pages 56–65, 2005.
14. Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS*, volume 45, pages 129–142, 2010.