Master d'Informatique
Cursus Master Ingénierie - Informatique, systèmes et réseaux

Travail d'Étude et de Recherche

Évaluation de l'unikernel Unikraft

Auteur :
Lucas CHEEKHOOREE

Encadrant : Pierre DAVID

8 mai 2022

Table des matières

1 Introduction

La partie "Introduction" et "Unikernel" sont un rappel, vous pouvez trouver des explications plus développé dans [1, 2, 3]

De nos jours on essaie de plus en plus d'optimiser l'utilisation des ressources que ce soit dans le Cloud, les super calculateurs ou bien même nos programmes. Se posent en plus des questions concernant la sécurité avec toutes les données sauvegardées numériquement. Pour répondre à ces besoins il y a d'abord eu les machines virtuelles [1] puis les conteneurs avec notamment Docker [2]. Suivi d'une troisième solution qui voit doucement le jour, les Unikernels.

Ce papier est une évaluation de performance d'Unikraft, un unikernel récent. Les mesures effectuées permettent de mettre en perspective l'évolution des unikernels. On reproduira les mêmes expériences que El-Kabir, avec Nano, Hermitux, Docker et Unikraft. On verra une nette amélioration concernant la taille de l'image et les appels systèmes.

Dans un premier temps je vais revenir sur la définition d'un unikernel. Expliquer leurs buts, leur fonctionnement et la manière dont ils sont créés. Puis je présenterai de manière succincte Unikraft. Qui sera par la suite évalué avec deux autres Unikernels : Nano et Hermitux, ainsi que Docker, solution de conteneurisation. L'analyse sera ciblée sur les performances processeur, appels systèmes, temps de démarrage et l'appel de fonctions.

2 Les Unikernels

2.1 Définition

2.2 Définition

Les unikernels sont des noyaux spécialement conçus pour exécuter une et unique application. Ils sont créés à partir des fonctions de base d'un noyau et fusionner avec le code d'une application. Ils sont tellement optimisés qu'il ne reste des fonctions du noyau que le strict nécessaire pour faire fonctionner l'application. Cette fusion permet de supprimer la séparation entre l'utilisateur et le noyau pour ne former plus qu'un, un noyau. On obtient donc une unique image permettant de faire tourner un unique processus.

Dans la figure **??** nous pouvons voir la structure des trois principales solutions. Aujourd'hui il est possbile d'executer un Unikernel à partir d'un hyperviseur ou bien directement depuis la mahcine, solution aussi nommé "bare metal". Il existe plusieurs hyperviseur tel que KVM, Xen qui sont des hyperviseur plutôt "*classique*", puis il en existe également des spécialisé pour des unikernels. A la différence des hyperviseurs "*classique*" un hyperviseur spécialisé pour Unikernel sera plus optimisé, notamment par la suppresion de fonctionnalité inutile à un unikernel. Parmis ces hyperviseurs on retrouve Uhyve, hyperviseur de l'Unikernel Hermitux [3] mais également Firecracker [3] qui lui est plutôt spécialisé pour les petits noyaux (pas forécement des unikernels).

Dans la figure 1, on peut voir trois stuctures, machine virtuelle, conteneur ainsi que les Unikernels. On peut voir que les unikernels ressemblent aux machines virtuelles dans le cas où on l'utilise avec un hyperviseur, à la différence prêt que l'unikernel est uniquement faite d'un espace noyau.

1. Machine virtuelle : `https://en.wikipedia.org/wiki/Virtual_machine`
2. Docker `https://www.docker.com/`
3. Firecracker : `https://firecracker-microvm.github.io/`

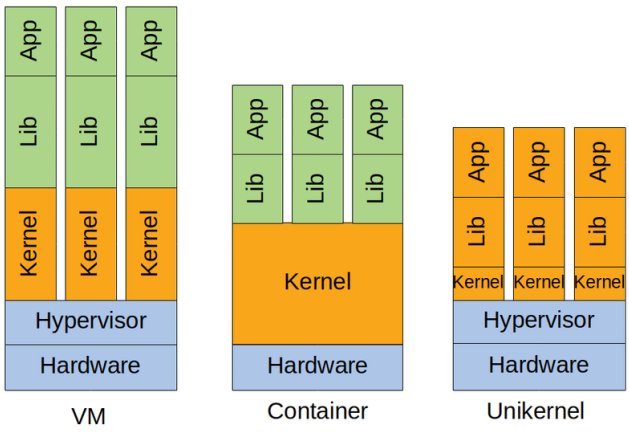

FIGURE 1 – Structure de machine virtuelle, conteneur, unikernel [4]

2.3 Les avantages et inconvénients

La taille de l'image d'un Unikernel à l'avantage d'être beaucoup plus petite que l'image d'une machine virtuelle ou bien d'un conteneurs. Cela s'explique par les besoins d'un Unikernel, en effet il a uniquement besoin de faire fonctionner un programme. De ce fait l'image sera uniquement formé du code de l'application et des librairies noyau nécessaire au fonctionnement. Cela réduit la taille du noyau au stricte minimum. Le code du noyau (optimisé) va être fusionné avec le code de l'application pour ne former plus qu'un. On vera dans la suite 5 que la taille des Unikernels est si petite qu'elle peut être mesuré en Mio, voir Kio pour Unikraft.

Les unikernels sont des systèmes chargé d'excuter qu'un seul programme. Dans le programme il se peut qu'il y est des appels systèmes. Pour optimiser l'excution de ces appels les unikernels ont choisi de supprimer la couche noyau-utilisateur. En ne formant plus qu'une couche les appels n'ont pas à changé de mode ils peuvent directement avoir accès aux données nécessaire. Ainsi on obtient des temps d'execution beaucoup plus rapide.

Nous pouvons également noté que l'unikernel ne fait tourner qu'un seul processus permettant ainsi de faire abstraction d'un ordonnanceur. Cela apporte un gain de temps mais en contre parti si on a besoin de faire fonctionner plusieurs programme il va falloir les faires communiquer par le réseau.

La surface d'attaque dans un unikernel est fortement réduit. En effet moins on utilise de code, moins il y a de possible faille.

2.4 Les 3 moyens de créer un Unikernels

La première technique consiste à retirer ou d'ajouter des librairies à un système existant. De cette manière on personnalise et optimise l'image que l'on veut obtenir. Cependant le nombre de dépendances inter-librairie compléxifie à très haut point la sélection de certaines librairies, il se peut qu'une librairie utilisé en utilise une autre et ainsi de suite. On peut retrouver dans l'article d'Unikraft [2] un graphe de dépendances montrant la complexité du problème. Cette technique est utilisée pour Rumprun et Osv par exemple.

La deuxième technique consiste à outre-passer le système d'exploitation pour mieux

gérer les entrées/sorties mais cela a pour inconvénient le tas en mémoire reste en place. Le gain de temps est finalement si faible qu'il n'y a pas grand intéret à l'utiliser.

La dernière technique consiste à partir de rien et en fonction des nécessités ajouter les librairies requise. Unikraft a retenu cette dernière solution. Pour sélectionner les librairies

3 Unikraft

3.1 Le groupe Unikraft

Le code est accessible sur Github sous licence BSD-3. Comparée à d'autres unikernels l'équipe est très active et le dépôt est régulièrement mis à jour. De plus le projet commence à devenir de plus en plus populaire. Il est partiellement financé par "the European Union's Horizon 2020" dans le cadre du programme Unicore, ils participent à la GSOC22 (Google Summer of Code 2022 [4]), organisent des hackatons et leur papier de recherche a reçu plusieurs prix. La communauté d'Unikraft travaille sur un Unikernel basé sur LibOS.

3.2 Les lignes directrice d'Unikraft

Unikraft s'est fixé plusieurs règles de développement. Elles sont retrouvables dans [2], et elles vont être re-expliquées dans cette partie.

Premièrement, Unikraft étant un unikernel, il est donc impératif qu'il fasse tourner une unique application. Ainsi donc on peut utiliser un système d'adressage simple. Dans le cas où l'on aimerait utiliser plusieurs applications, il faudra faire plusieurs unikernels qui communiqueront entre eux via le réseau.

Secondement, le principe d'unikernel veut qu'on ait un unique espace noyau. On supprime ainsi l'espace utilisateur pour ne conserver que l'espace noyau. Exécuter une application directement depuis le noyau permet de supprimer l'étape intermédiaire qui traduisait les instructions.

Unikraft se démarque surtout par son travail effectué sur la modularité. Ils ont étudié les dépendances et créé leurs propres librairies pour optimiser au maximum l'unikernel. Les librairies choisissent peuvent être sélectionnées directement en modifiant les fichiers de configuration, par une interface en ligne de commandes, ou bien même par une interface graphique.

En plus des optimisations d'adressage et des espaces supprimés, les créateurs d'Unikraft ont ajouté un chaînage statique. Celui-ci permet l'élimination de code mort et l'optimisation "Link Time Optimization" (LTO) [5].

On a vu précédement qu'Unikraft créé ses propres librairies et permet l'ajout de librairies externes. La compatibilité avec les fonctions POSIX permet de couvrir de nombreuses applications, notamment les "*legacy applications*"

Le dernier point concerne le support sur différentes plateformes. J'ai uniquement expérimenté QEMU/KVM, il est cependant possible d'utiliser Xen, Firecracker, Solo5 et "*bare metal*".

4. GSOC `https://summerofcode.withgoogle.com/`
5. LTO `https://gcc.gnu.org/wiki/LinkTimeOptimization`

4 Protocole expérimentale

Les tests sont utilisés sont inspiré de ceux de El-Kabir. La principale différence à été de retirer les variables de type **float** pour les remplacer par des entiers. En effet il m'a été déconseillé par les créateurs d'Unikraft d'utiliser des nombres réels dans un Unikernel. Les tests se font en plusieurs étapes. Tout d'abord (lorsque cela est nécessaire) j'ai compilé l'image de l'unikernel, puis j'ai lancé les tests 100 fois de manière automatique et les résultats ont été placé dans des fichiers qui ont ensuite été analysés. Certains tests tels que les mesures processeur, les appels systèmes et les appels de fonctions ont eux-mêmes une boucle **for** qui itére un million de fois. Dans ces résultats, on divisera le temps par un million pour obtenir un temps approximatif d'une itération. Cela nous permet de mieux comparé nos résultats à ceux trouvé par El-Kabir.

Certains unikernels tel qu'Unikraft, proposent d'optimiser le code. On peut le modifier dans les paramètres lors de la compilation. Il est possible d'améliorer la taille de l'image ainsi que les performances (*optimisation du temps d'exécution*). Lorsqu'on sélectionne l'amélioration de performances on obtient un code compiler avec un équivalent de **gcc ... -O3 ...**. Sélectionner cette option n'était pas intéressante pour nous étant donné que l'on cherche à comparer les unikernels de manière, soit un code égal. Si l'on active l'optimisation notre code sera modifié. De plus lorsque j'ai expérimenté unikraft avec cette option j'obtenais des temps si faibles qu'ils n'étaient pas mesurables.

Le temps des tests est mesuré avec la primitive système **gettimeofday**. Sauf le temps de démarrage et d'arrêt qui lui eux sont mesuré grâce à la commande bash **time**.

4.1 Le temps de démarrage et d'arrêt de l'Unikernel

Ici on reproduit l'expérience effectuée par Monsieur El-Kahbir [1] c'est-à-dire un code C contenant uniquement une fonction **main** avec un **return**. Le temps d'exécution est mesuré à l'aide de la commande **time**. La commande **time** retourne trois valeurs **real, user, sys**. Personnellement j'ai récupéré la durée **real** pour avoir l'ensemble du temps.

4.2 Processeur

On mesure la performance du processeur dans un Unikernel en chronométrant le temps de calcule qu'il lui faut pour la suite de fibonacci jusqu'à sa 40^{ieme} itération. En effet au-delà de la 40^{ieme} itération (44^{ieme} pour être précis) il y a un débordement qui fausse les valeurs.

J'ai choisi d'utiliser la suite de fibonacci au lieu d'un trie tel effectué dans le papier de référence [1] car je n'ai pas réussi à utiliser **9pfs**[6] qui permet d'ajouter des fichiers lors de l'exécution de la machine virtuelle. Cela m'aurait permis d'ajouter un fichier à lire. Malheueusement, pour unikraft, je n'arrivais pas à lire le fichier même après l'ajout de la librairie **vfscore** qui permet la lecture de fichier. On aurait pu aussi faire des calculs de matrices mais la taille aurait pu poser des problèmes de pages dans la mémoire. Le calcul de la suite de fibonacci paraît donc pertinent.

4.3 Appel Système

Pour mesurer le temps des appels systèmes on ferme un fichier inexistant d'identifiant 3 tels effectué par El-Kabir [1]. On effectue cette opération avec la primitive **close**.

6. 9pfs :`https://wiki.qemu.org/Documentation/9psetup`

4.4 Appel fonction

Pour mesurer les appels de fonctions on utilise un algorithme qui appelle un million de fois une simple fonction qui retourne uniquement le nombre 100. Mesurer les appels de fonctions permettent de vérifier les linux ABI [7] tel évoqué dans l'article parite 3.2. [5]

4.5 Temps entre deux appels gettimeofday

Pour mesurer le temps dans nos différentes expérimentations on utilise la fonction **gettimeofday**. Notons qu'il existe un temps entre le moment où l'appel est effectué et le moment où la réponse est notée (voir 2). Dans cette expérience là on va donc mesurer cette différence pour voir s'il faut le prendre en considération ou pas.

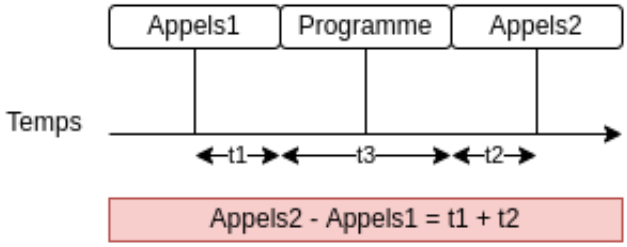

FIGURE 2 – Temps entre deux appels gettimeofday. Les appels correspondent à la fonction **gettimeofday** et le bloc programme représente une partie de code que l'on cherche à mesurer. Entre le moment du lancement de l'appel et le résultat enregistré s'écoule certains un temps **t**. Cette latence est mesurable en exécutant deux **gettimeofday**, l'un après l'autre.

4.6 Tailles des images

Les tailles des images sont aussi très importantes dans notre expérimentation. On cherche non seulement à voir si les unikernels sont performants mais aussi s'ils ne prennent pas trop de place. Pour mesurer la taille des unikernels j'ai utilisé la commande **ls -l** dans les bons répertoires (voir tableau ci-dessous).

Unikernel	Emplacement
Unikraft	/build/*kvm-x86_64
Nano	/home/<user>/.ops/images/monApp.img
Hermitux	/hermitux/hermitux-kernel/prefix/x86_64-hermit/extra/tests/hermitux

Notons également que pour obtenir l'image d'Hermitux total il faut ajouter la taille du binaire compilé au préalable.

Pour récupérer les images docker, après avoir construit l'image j'ai utilisé la commande **docker image ls**.

7. Description de Linux ABI `https://en.wikipedia.org/wiki/Application_binary_interface`

5 Expérimentations

5.1 Différentes solutions expérimentées

On effectuera les expériences sur trois Unikernels et Docker. Nous aurons évidemment Unikraft qui sera comparé à Hermitux et Nano OS. Avoir plusieurs unikernels nous permet de comparer avec plus de données. Par exemple si une expérience sur Hermitux diffère de l'expérience d'origine effectuée par El-Kabir on aura toujours le second Unikernel pour vérifier, en plus de Docker. Docker est surtout important pour avoir une comparaison unikernel et conteneur, pour voir si les unikernels pourraient devenir une alternative.

5.2 Configuration

Système	Ubuntu 20.04.4 LTS
Ordinateur	Dell Inspiron 15 5515
Mémoire vive	16 GB
Processeur	AMD® Ryzen 7 5700u with radeon graphics × 16
Coeurs physiques	8
Coeurs virtuels	16
Fréquence processeur	1.8 GHz
Fréquence processeur max	4.3 GHz
Fréquence processeur min	1.4 GHz
gcc version	(Ubuntu 9.4.0-1ubuntu1 20.04.1) 9.4.0
qemu version	4.2.1 (Debian 1 :4.2-3ubuntu6.21)
kraft version	0.5.0.dev491
nano os version	0.0
Ops version	0.1.30
Hermitux version	1.0

5.3 Tableaux des résultats

Dans la figure 3 nous avons un tableau à double entrée. Les lignes représentent les différentes solutions testées et les colonnes contiennent les résultats selon les tests effectués. Les cases rouges mettent en avant les résultats "gagnants", dans d'autres termes ce qui ont atteint les meilleures performances.

Dans la figure 4 nous avons un tableau à double entrée. Les lignes représentent les différentes solutions testées et les colonnes contiennent les écarts-types des différents tests effectué.

Dans la figure 5 nous avons un tableau à double entrée. Les lignes représentent les différentes solutions testées et les colonnes contiennent les tailles d'images des différents programmes testés. Les cases rouges mettent en évidence les meilleurs résultats.

5.4 Analyse des résultats obtenue

5.4.1 Le temps de démarrage et d'arrêt de l'Unikernel

On peut ici faire la même faire conclusion que le papier auquel nous nous référons [1]. Hermitux est l'unikernel ayant le meilleur temps de lancement puis fermeture qui s'explique tel évoqué dans la référence par l'hyperviseur Uhyve [8]. Cependant l'équipe d'Unikraft tra-

8. `https://github.com/hermitcore/uhyve`

	Temps de lancement et fermeture (ms)	Mesure de performance CPU (ns)	Mesure d'appel système (ns)	Mesure d'appel de fonction (ns)	Temps entre deux appels (µs)
Unikraft	223,79	116,21	20,09	1,41	0,24
Hermitux	56,73	115,71	1214,40	1,63	0,13
Nano	3489,65	116,43	501,18	1,92	0,65
Docker	39,56	123,86	70,65	3,74	0,1

FIGURE 3 – Tableau de performances

	Temps de lancement et fermeture (ms)	Mesure de performance CPU (ns)	Mesure d'appel système (ns)	Mesure d'appel de fonction (ns)	Temps entre deux appels (µs)
Unikraft	5,47	1,44	0,22	0,05	0,43
Hermitux	3,51	1,06	26,82	0,05	0,34
Nano	269,07	1,67	2,03	0,73	0,50
Docker	6,25	3,05	2,65	1,28	0,30

FIGURE 4 – Tableau des écarts-types

taille en Mio	Boottime	Cpu3	Funcall	Syscall
Unikraft optimisé	0,26	0,26	0,26	0,33
Unikraft	0,26	0,33	0,33	0,46
Hermitux	4,04	4,04	4,04	4,04
Nano	15,64	15,65	15,65	15,65
Docker	1 370,00	1 370,00	1 370,00	1 370,00

FIGURE 5 – Tableau des tailles d'images

vaille aussi sur la compatibilité avec Firecracker [9] qui présente selon leurs expériences de meilleur résultat qu'avec QEMU (voir [2] fig.10).

On constate cependant que Docker a un temps meilleur à tous les Unikernels, même Hermitux. Ce constat était déjà fait dans un travail (voir [3]), cependant il diffère des résultats trouvés dans [1] qui avait trouvé un meilleur tant pour Hermitux et des écarts plus grands entre Hermitux et Docker.

5.4.2 Processeur

Les différents temps processeurs sont à peu près tous égaux, ce qui est un résultat plutôt satisfaisant. Cela signifie que l'utilisation du processeur est à peu près similaire partout. Si les résultats semblent élevés cela provient du fait qu'on n'utilise pas l'option **O3** lors de la compilation. Lorsqu'on utilise l'option le code est tellement optimisé qu'on obtient une valeur proche de zéro, soit trop faible pour être mesuré.

On ne peut pas réellement comparé nos valeurs telquel par rapport au travail auquel on se refaire. Par contre on peut observer les écarts entre les différentes solutions. Dans mes résultats on a un écart très faible. Cela provient sûrement de la lecture de fichiers effectués dans le travail de El-Kabir. La lecture du fichier qu'il effectue peut impacter de manière importante les résultats.

5.4.3 Appel Système

Le but des unikernels étant de réduire le temps des appels systèmes, ce test est clé pour notre analyse. On peut observer ici qu'Unikraft atteint les résultats les plus satisfaisants. Comparé aux autres Unikernels, il construit une image en amont avec des librairies déjà compilées et liées de manière statique (pour plus de détails voir [2] part.4).

Comparé au travail de El-Kabir on a des temps totalement différent sauf pour Nano OS. J'ai obtenu un temps bien meilleur avec Docker, j'ai sûrement utilisé une configuration différente. Pour Hermitux j'ai obtenu un temps beaucoup moins bon, j'ai réitéré l'expérience plusieurs fois mais rien n'a changé. Je ne sais pas expliquer l'écart de cette mesure. J'ai utilisé la compilation statique qui devrait normalement produire un meilleur résultat.

5.4.4 Appel fonction

On constate comme pour les tests processeurs, qu'ici les temps sont très proches. Cependant on remarque tout de même un faible écart entre tous les résultats, mise à part Docker qui est plus de deux fois plus long d'Unikraft. Les écarts montrent qu'il existe tout de même une différence dans la gestion des appels de fonctions surtout entre conteneurs et unikernels.

5.4.5 Temps entre deux appels gettimeofday

On observe que tous les résultats se situent sous une microseconde. On peut donc négliger cette durer et ne pas la prendre en compte dans notre analyse.

5.5 Analyse des écarts-types

La plupart des écarts-types sont assez faibles. On peut noter que certaines expériences ont de plus grand écart que d'autres notamment le temps de lancement et de fermeture des Unikernels. Les valeurs étant plutôt faible révèlent une certaine stabilité. Cependant

9. `https://firecracker-microvm.github.io/` hyperviseur pour micro VM

on observe que le temps de démarrage et d'arrêt de Nano OS est assez faible. Cette valeur est dotant plus intéressante qu'est son temps plus long que toutes les autres solutions. Cela provient assez certainement des opérations supplémentaires efffectuées telle l'attribution d'une adresse ip. On peut également noter qu'Hermitux à l'écart-type le plus élevé dans l'expérience des appels systèmes et de même c'est l'expérience où Hermitux exergue la moins bonne performance en comparaison des autres Unikernels.

5.6 Analyse des tailles d'images

Les différences entre les tailles d'images sont ici non négligeables. Unikraft expose les tailles d'images les plus petites ce qui s'explique par la possibilité de personnaliser chaques librairies incluse avec en plus des librairies créées spécialement pour Unikraft. J'ai aussi testé leur option d'optimisation de la taille d'image, "Unikraft optimisé", qui montre des tailles encore plus petite de 7Kio à 13Kio. On peut aussi noter que l'image la plus optimisée est celle des appels systèmes, donc on pourrait émettre l'hypothèse qu'Unikraft optimise encore mieux les appels systèmes. Pour en être certain il faudrait faire des tests avec beaucoup plus de code différents, contenant différents appels système.

En seconde place on a Hermitux qui applique aussi une sélection de librairie mais comparé à Unikraft se ne sont pas des librairies personnalisées. De plus Unikraft utilise comme librairie de standard C **nolibc** qui est une librairie plus légère que **newlibc** utilisé par Hermitux. En dernière position on a Docker qui est beaucoup moins optimisé.

On peut aussi noter que toutes les tailles d'images sont plutôt fixes à part pour Unikraft où les tailles peuvent énormément varier selon le programme compiler. Cela s'explique par l'optimisation de code et la sélection de librairie.

6 Retour d'expérience d'Unikraft

6.1 Installation

J'ai pu essayer Unikraft sur Rasberry Pi 4 modèle B, Ubuntu 20, Kali Linux (version 2022) en live boot. L'installation d'Unikraft est plutôt classique seulement les tutoriels d'installations divergent entre eux et proposent des solutions différentes, certaines ne sont plus à jour. L'installation sur Rasberry Pi a pu être faite mais je n'ai pas réussi à l'utiliser dû à des problèmes lors de la compilation.

6.2 Communauté

Concernant l'utilisation elle est en surface assez simple mais c'est un projet en pleine évolution qui manque encore de documentation et peut s'avérer très technique à certains moments. Il y a heureusement une communauté active sur github, discord et autres réseaux. Cette évolution peut être vue comme un point positif, mais cela est aussi source de problèmes car les versions sont encore très instables.

6.3 Utilisation

Unikraft comparé aux autres Unikernel permet de personnaliser l'image à un très haut niveau de détails. On peut ajuster les librairies, utiliser des outils de débbugage, choisir les options de compilation, les architectures etc... Toutes ces options personnalisables font d'Unikraft un outil pratique pour prendre la main sur la création de notre Unikernel. Cependant cette personnalisation a son coût. Pouvoir modifier autant de paramètres implique

une charge de travail supplémentaire comparé aux Unikernels faisant principalement une traduction binaire.

7 Conclusion

Unikraft émergeant comme nouveau unikernel est un projet très actif en plein développement qui essaie de se faire connaître par le monde professionnel. En effet le groupe d'Unikraft est à l'écoute de ses utilisateurs et accompagne les entreprises qui souhaitent utiliser leur création.

On peut de plus évoquer FlexOS qui utilise Unikraft pour développer sa propre technologie. En effet FlexOS [**?**] créés d'abord un unikernel optimisé et léger avec Unikraft puis ajoute des librairies pour sécuriser et optimiser avec LibOS.

8 Annexe

8.1 Tableau de performances de El-Kabir

	Nanos	OSv	Hermitux	Rumprun	VM	Docker
Latence appel système (ns)	536,963 ±4,076	19,479 ±0,50	45,3 ±0,8	67,77 ±4,22	66,848 ±1,20	145,932 ±2,28
Latence appel de fonction (ns)	3,965 ±0,296	3,372 ±0,19	2,288 ±0,13	2,784 ±0,57	3,08 ±0,32	4,160 ±0,27
Temps CPU (µs)	895 ±40	$2,52 \times 10^6$ $\pm 0,13 \times 10^6$	216 ±11	N.A.	95 ±6,5	668 ±21,32
Empreinte mémoire (Mio)	114	78,3	6	102,4	1518	1,7
Temps de boot (ms)	305 ±6	368 ±5,2	4,65 ±0,5	647 ±8	24000[1] 2170[2] ±0,25[1] ±0,25[2]	48 ±1,9
Taille image (Mio)	15,6	7,3	3,1	29	7800[3]	217

Figure 6 – Tableau de performances de El-Kabir [1]

Références

[1] D. EL-KABIR, "Etat de l'art des unikernel," pp. 1–25, 2021.

[2] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, c. Teodorescu, C. Răducanu, C. Banu, L. Mathy, R. Deaconescu, C. Raiciu, and F. Huici, "Unikraft : Fast, specialized unikernels the easy way," in *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, (New York, NY, USA), p. 376–394, Association for Computing Machinery, 2021.

[3] P. Olivier, D. Chiba, S. Lankes, C. Min, and B. Ravindran, "A binary-compatible unikernel," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2019, (New York, NY, USA), p. 59–73, Association for Computing Machinery, 2019.

[4] T. Goethals, M. Sebrechts, A. Atrey, B. Volckaert, and F. De Turck, "Unikernels vs containers : An in-depth benchmarking study in the context of microservice applications," in *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, pp. 1–8, IEEE, 2018.

[5] I. Briggs, M. Day, Y. Guo, P. Marheine, and E. Eide, "A performance evaluation of unikernels," in *Technical Report*, 2014.