Un prototype de cache de métadonnées pour le passage à l'échelle de NixOS-Compose

Dorian Goepp, Samuel Brun, Quentin Guilloteau, Olivier Richard

Compas'2024 : Parallélisme/ Architecture/ Système
IETR/LS2N/Nantes-U - Nantes - France, 2-5 juillet 2024

Un prototype de cache de métadonnées pour le passage à l'échelle de NixOS-Compose

Dorian Goepp[a], Samuel Brun[a], Quentin Guilloteau[b], Olivier Richard[a]

[a] Univ. Grenoble Alpes, INRIA, CNRS, Grenoble-INP, LIG
[b] Department of Mathematics and Computer Science, University of Basel

Résumé
Les gestionnaires de paquets fonctionnels tels que Nix et Guix causent une nouvelle charge pour le système de fichiers. Ils organisent les paquets dans un magasin, ce qui fait que le chargeur de bibliothèques doit chercher chaque dépendance dans un grand nombre de répertoires, causant de nombreux appels système. Dans le cadre de NixOS-Compose, un outil basé sur Nix pour développer et déployer des environnements logiciels distribués, la charge induise par un déploiement pourrait saturer le système de fichiers distribué. Cet article présente les travaux en cours sur la conception et l'évaluation d'un cache partagé pour limiter cette charge.

Mots-clés : système de fichiers, métadonnées, expérience distribuée, Nix, reproductibilité

1. Introduction

Le projet NixOS-Compose [9] a pour objectif de réduire la complexité de la construction et du déploiement d'environnement logiciels pour l'expérimentation en systèmes distribués. Le processus de mise en place d'un environnement logiciel distribué est itératif et chronophage, ce qui incite les expérimentateurs a converger vers un résultat fonctionnel, mais peu reproductible. NixOS-Compose se base sur le gestionnaire fonctionnel de paquets *reproductibles* Nix [6] et la distribution Linux associée, NixOS [7], pour décrire les environnements logiciels d'un système distribué, de façon à pouvoir procéder en itérations locales rapides avec Docker ou des machines virtuelles, puis de les transposer *à l'identique* sur le système sur la plateforme expérimentale Grid'5000 [1]. À noter que NixOS-Compose n'est pas structurellement restreint à Grid'5000.
NixOS-Compose permet de déployer des environnements logiciels sur Grid'5000 de plusieurs façons, appelées *flavours*. L'une d'entre elles (`g5k-image`) construit une image système de l'environnement logiciel et sera ensuite déployée via Kadeploy [10]. Une autre possibilité plus rapide (`g5k-nfs-store`) utilise la fonction `kexec` du noyau Linux qui permet de charger un nouveau noyau sans passer par un redémarrage complet de la machine au prix d'une surconsommation de mémoire vive.
Dans ces deux variantes, NixOS-Compose génère une image `ramdisk`. Afin de limiter la taille ce cette image, la *flavour* `g5k-nfs-store` ne copie pas l'environment logiciel dans l'image, mais uniquement la liste du contenu de l'environment logiciel. Au cours du déploiement, le noyau Linux de l'image `g5k-nfs-store` va réutiliser un montage NFS [20] mis en place par Grid'5000 pour monter le magasin Nix (section 2.1), répertoire qui contient les binaires et bibliothèques de l'environnement logiciel, et ainsi pouvoir définir correctement la variable d'en-

vironment `$PATH` qui donne accès aux binaires de l'environnement logiciel. Cette approche est similaire à nfsroot [16].
Bien que cette approche soit plus rapide et légère, elle impose plus de charge sur le serveur NFS qui héberge le magasin Nix. Nous verrons en section 2.3 que lors du démarrage un nombre important d'appels système sont exécutés par les machines déployées pour démarrer leurs services. Étant donné que le magasin Nix est accédé via NFS, ce « raz-de-marée » d'appels système a le potentiel de rendre le serveur NFS inutilisable [19]. Dans cet article, nous présentons des résultats préliminaires pour réduire la pression sur le serveur NFS.

2. Éléments de contexte

Il nous semble utile d'aborder la façon dont Nix organise les paquets logiciels (section 2.1), et son impact sur le chargement des bibliothèques partagées sous Linux (section 2.2). En découle que NixOS-Compose cause une déferlante de requêtes de métadonnées (section 2.3).

2.1. Introduction à Nix et à son magasin de paquets

Nix [6] est un gestionnaire de paquet fonctionnel (comme Guix [2]). Cela signifie que les sources de chaque paquet forment les entrées d'une fonction dont la valeur de retour est le paquet compilé (ou juste assemblé si on parle de langage interprété). Nul autre que les entrées déclarées de cette fonction ne sont disponibles lors de la construction du paquet, ce qui évite tout oubli. Aussi, la construction de paquet se faisant de manière isolée, il devient possible d'installer deux paquets dépendant de deux versions incompatibles d'une même bibliothèque. Ces caractéristiques permettent de produire des environnements logiciels reproductibles avec Nix.
A la différence de l'architecture usuelle dite *Filesystem Hierarchy Standard (FHS)* [18], qui stocke les binaires et bibliothèques dans un petit nombre de répertoires (*ex.*, `/usr/lib`, `/usr/bin`), Nix stocke chaque paquet dans son propre répertoire. Ce répertoire est nommé selon le schéma `<hash>-<nom>-<version>`, où le `hash` correspond au hash cryptographique des entrées du paquet. Tous les paquets sont installés dans le *magasin* Nix, qui est le répertoire `/nix/store`.
Un autre élément d'importance : le magasin Nix (`/nix/store`) est en lecture seule pour tous sauf le *daemon* de Nix qui a l'autorisation d'ajouter (mais pas de modifier) du contenu. Cette propriété nous sera utile à l'implantation de notre solution (section 3), en nous épargnant des considérations de cohérence de cache distribué.
Cette façon de faire a un coût que nous allons maintenant aborder dans le contexte des bibliothèques partagées, après un court rappel sur la façon dont elles sont chargées.

2.2. Le chargeur de bibliothèques partagées est ralenti par le magasin Nix

La plupart des binaires ELF (Executable and Linkable Format) dépendent de bibliothèques partagées qui sont chargées dans la mémoire au démarrage du programme. Le chargeur de bibliothèques `ld-linux.so` [12] doit chercher ces bibliothèques dans l'arborescence de fichiers. Nix ne se conforme pas au *FHS* : chaque bibliothèque est dans son propre répertoire du magasin de paquets, ce qui amène a une explosion combinatoire des chemins que `ld-linux.so` doit tester pour chaque bibliothèque. Lorsque le chargeur de bibliothèques tente d'ouvrir une bibliothèque dans un répertoire où elle n'est pas, il effectue l'appel système `open()` qui retourne une erreur, n'entraînant aucun transfert de données. On parle alors de déferlante de métadonnées (en anglais *metadata storm* ou *stat storm*) car le système de fichiers est saturé par des requêtes sur des fichiers absents (qui est une métadonnée) plutôt que par le transfert d'une grande quantité de données. Les travaux [3, 22] donnent plus de détails à ce sujet.
Ce phénomène se retrouve également pour les langages dynamiques qui résolvent les dépen-

dances au démarrage [8, 11]. Cette étude se concentre cependant sur les binaires ELF dans le contexte de leur déploiement avec NixOS-Compose.

2.3. La déferlante de métadonnées dans le contexte de NixOS-Compose

L'introduction nous a appris que le déploiement NixOS-Compose le plus efficace sur Grid'5000, la *flavour* `g5k-nfs-store`, consiste en une image Linux minimaliste en `ramdisk` et un magasin Nix monté en NFS depuis un serveur Grid'5000 commun aux utilisateurs d'un centre de calcul.
La déferlante de métadonnées prend de nouvelles proportions dans ce contexte. En premier lieu, elle n'affecte pas un support de stockage local, mais passe par NFS, une ressource partagée à préserver. En second lieu, cette ressource partagée est d'autant plus sollicitée que le nombre de machines déployées est important. De par la nature de NixOS-Compose, l'effet de la déferlante de métadonnées est amplifié.

3. *Chorage* : un cache distribué pour le déploiement d'environment logiciels distribués

3.1. Objectifs et contraintes

Notre objectif principal est d'atténuer la déferlante de métadonnées sur le serveur NFS hébergeant le magasin Nix, sans impacter la durée de déploiement. Afin de garder une approche généraliste qui s'appliquerait au langages dynamiques, nous nous interdisons d'interférer avec le chargeur de bibliothèques partagées `ld-linux.so` [12].
Le serveur NFS du centre de calcul de Grid'5000 doit être utilisé en l'état, sans pouvoir ajouter de logiciel tiers pour optimiser notre cas d'usage. Par contre, l'usage de NixOS-Compose nous permet d'intervenir sur le client NFS des machines expérimentales.
Notre proposition, *Chorage*, est un cache distribué de métadonnées visant à réduire la charge du système de fichiers hébergeant le magasin Nix. Son objectif est de filtrer autant de requêtes inutiles que possible avant qu'elles ne passent par NFS.
Notons que ce cache est à usage unique : il disparaît entre les déploiement NixOS-Compose car rien ne persiste sur les machines. Il doit donc être utile dès son premier lancement, sans phase de pré-chauffe.

3.2. État de l'art sur l'atténuation des déferlantes de métadonnées

Nous récapitulons l'état de l'art dans la table 1. Les travaux [13, 21] sont très efficaces, évitant complètement la déferlante de métadonnée, mais sont limitées aux binaires ELF. Nous préférons une approche pouvant fonctionner avec les langage dynamiques. Nous ne pouvons aussi pas nous permettre de modifier l'infrastructure logicielle du centre de calcul, ce qui écarte les propositions [14, 17]. L'approche de [15] consiste à créer une image disque contenant toutes les dépendances nécessaires et montée sur les machines finales. Elle est pertinente si l'image est persistante sur le disque des machines expérimentales, ce qui nous est impossible avec NixOS-Compose sur Grid'5000.
Très proche de nos travaux, [19] requiert cependant de lancer les programmes une fois en amont pour générer le cache de métadonnées, ce serait utile dans un second temps pour de la production mais est inapproprié avec les nombreuses itérations du développement d'expérience.

3.3. Conception : un cache de métadonnées sélectif et distribué

Nous proposons un cache intermédiaire qui se rajoute à ceux de NFS et Linux, pour filtrer les requêtes sur des chemins qui n'existent pas dans un système de fichiers donné. Nous illustrons notre conception par la figure 1, expliquée au fil de l'eau.

TABLE 1 – État de l'art sur les atténuations de la déferlante de métadonnées. La restriction « infrastructure » signifie qu'il faut modifier l'infrastructure logicielle du centre de calcul.

article	approche	restriction	implantation
[19]	traces, préchargement de cache, métadonnées -> données		FUSE
[15]	métadonnées -> données		image disque
[17]	pré-chargement, cohérence faible, lot de requêtes	Lustre	modifications de Lustre
[14]	limite de débit (QoS) dynamique	invasif	interception POSIX
[13]	cache spécifique à `ld-linux.so` pour Guix	binaire ELF	dans Guix
[21]	chemins absolus pour chaque dépendance	binaire ELF	modification du binaire

Avant de continuer, nous devons ouvrir une parenthèse sur la façon dont Linux traduit l'ouverture d'un fichier avec `open()` ou l'accès à ses métadonnées avec `stat()`. En l'occurrence, admettons que `ld-linux.so` tente d'ouvrir la bibliothèque partagée `libjpeg.so` au chemin `/store/0ad51r-zlib-1.3/lib/libjpeg.so`. L'appel système `open()` correspondant est traduit par Linux en plusieurs appels à la fonction `lookup(<inode>, <nom>)` du système de fichiers. L'argument `<inode>` est son identifiant du point de vue du système de fichiers et `<nom>` est le nom du répertoire ou fichier qu'on cherche en son sein. `lookup()` retourne l'inode de l'entité cherchée. Un chemin se parcourt donc répertoire après répertoire pour enfin parvenir au fichier désigné, appelant chaque fois `lookup()`.
Chorage opère donc au niveau de cette opération `lookup()`. Avant de contacter le système de fichiers NFS sous-jacent, le cache est consulté au cas où le nom recherché dans le répertoire donné n'existe pas. Le cas échéant, l'opération `lookup()` se termine rapidement. Dans le cas contraire, l'opération est relayée au client NFS comme c'est le cas en (1). Comme c'est la première occurrence de cette opération, *Chorage* la relaye au serveur NFS, qui répond par une erreur en (2). *Chorage* enregistre l'association *(inode du répertoire, nom recherché)*. Notons que le cache n'a pas à être mis à jour puisque le magasin Nix est en *lecture seule* et qu'une bibliothèque d'un paquet ne peut pas se trouver dans le répertoire d'un autre paquet.
Par ailleurs, une grande partie de l'environnement logiciel est partagé par les machines d'un déploiement NixOS-Compose car (1) elles partagent les services de base et (2) il est fréquent que de nombreuses machines n'utilisent que quelques environnements logiciels. Nous voulons tirer profit de ces points communs en propageant le cache à tous les participants, pour compenser le fait que le cache est à usage unique (voir section 3.3). Ainsi, à l'étape (3), *Chorage* informe ses pairs de l'absence de `libjpeg.so` dans le répertoire `/store/0ad51r-zlib-1.3/lib/`. Les instances de *Chorage* sur les autres machines incorporent l'information dans leurs caches en (4). Ainsi, lorsqu'une autre machine tente aussi d'ouvrir `libjpeg.so` dans le répertoire de la `zlib` en (5), *Chorage* répond directement (6), sans avoir à interroger le serveur NFS.

3.4. Implantation basée sur un overlay filesystem FUSE

Chorage [1] implante un système de fichiers FUSE [4] qui relaye la plupart des requêtes à un système de fichiers sous-jacent, donnant l'illusion d'être un *overlay filesystem* [5]. Pour la fonctionnalité de cache distribué, nous n'avons modifié que les fonctions `main()` et `do_lookup()`, et

1. Le code source est disponible dans Software Heritage : `https://archive.softwareheritage.org/swh:1:dir:108ab10941b45e943dc8615851186e837ed1acf1`

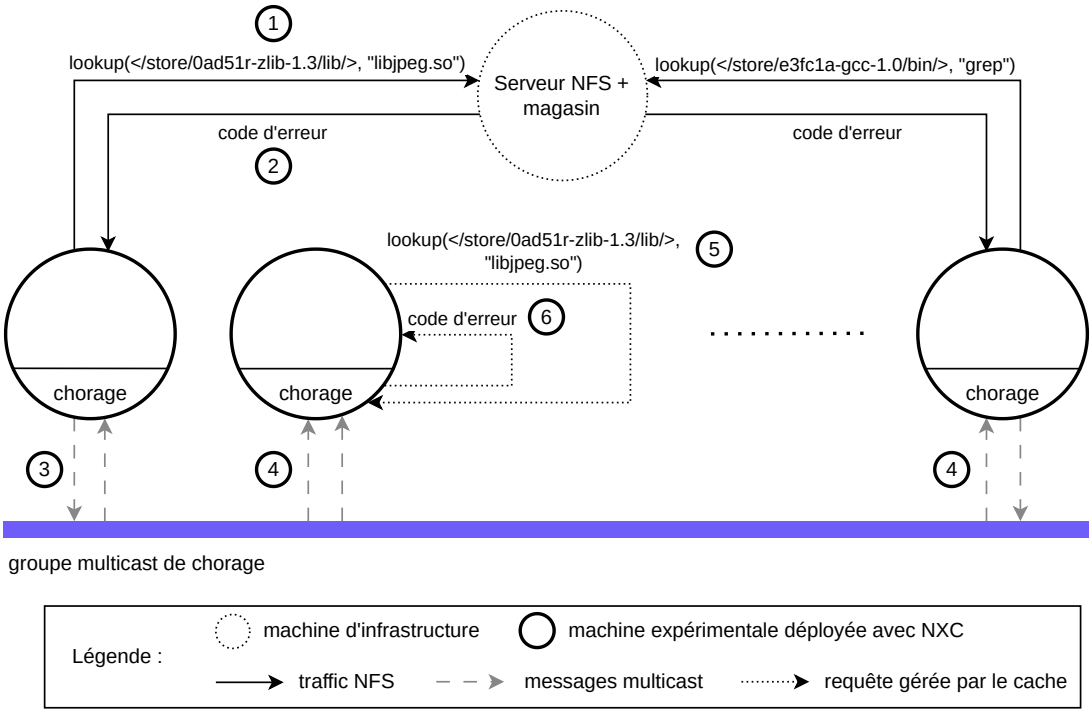

FIGURE 1 – Le fonctionnement du cache de métadonnées *Chorage*. Plutôt que de montrer les inodes (Linux) ou traitant (*handle*) fichier (NFS) dans les appels à `lookup`, nous représentons le répertoire correspondant entre ⟨⟩.

ajouté 292 lignes de code au total (hors débogage et analyse) aux 1038 lignes de code initiales. Le cache a pour clef la paire *(inode du répertoire, nom recherché)* décrite en section 3.3 et pour valeur le code d'erreur venant du système de fichiers sous-jacent (un `int`). Le cache est diffusé par chaque machine à toutes les autres par *multicast IP*.

4. Expériences

4.1. Plan d'expérience

En guise de première évaluation de *Chorage* nous avons compté parmi les opérations `lookup()` ayant échoué celles qui ont bénéficié du cache, et celles qui ont du passer par le système de fichiers sous-jacent, donc par NFS. Nous pourrons donc estimer le nombre d'opérations que nous avons épargné au serveur NFS.

Nous commençons par préparer un logiciel en C avec 1000 dépendances synthétiques chacune placée dans sa propre sous-arborescence d'un magasin imitant l'architecture du magasin de Nix. Nous avons choisi 1000 dépendances dans un premier temps pour représenter un cas pathologique et nous permettre de mieux observer les phénomènes étudiés.

Pour que le chargeur de dépendances puisse faire son travail, nous enregistrons les chemins absolus de chacune des dépendances au `rpath` du binaire ELF, une des façons de spécifier où `ld-linux.so` doit chercher les dépendances [12]. Nous avons vérifié que `ld-linux.so` a le même comportement que nous observions avec des programmes du magasin Nix.

Pour ne pas surcharger le serveur NFS du centre de calcul, nous déployons notre propre serveur NFS sur une des machines expérimentales, où nous plaçons notre substitut de magasin de paquets. Ensuite, 10 autres machines expérimentales ayant monté ce magasin via NFS, nous montons le cache *Chorage* et nous lançons le logiciel de test simultanément sur ces machines.

TABLE 2 – Mesures de performance, avec un pourcentage total de 13,7 % et une médiane de 14,1 %. Les colonnes en cache et hors cache indiquent le nombre d'opérations lookup avec échec qui ont respectivement été gérées par le cache ou par NFS.

machine	en cache	hors cache	ratio de cache	machine	en cache	hors cache	ratio de cache
gros-65	20 514	480 986	4,1 %	gros-70	48 040	453 460	9,6 %
gros-66	46 020	455 480	9,2 %	gros-71	30 341	471 159	6,1 %
gros-67	76 847	424 653	15,3 %	gros-72	127 889	373 611	25,5 %
gros-68	74 893	426 607	14,9 %	gros-73	66 111	435 389	13,2 %
gros-69	108 927	392 573	21,7 %	gros-74	88 123	413 377	17,6 %

Pour préciser notre méthode expérimentale, les configurations logicielles utilisées pour le serveur NFS et les clients exécutant le logiciel de test sont déclarées, réalisées et déployées avec NixOS-Compose sur 11 machines du cluster Gros de Grid'5000 à Nancy [1].

4.2. Résultats

Nous avons vérifié le bon fonctionnement du cache distribué avec une expérience durant laquelle le programme de test était d'abord exécuté sur une machine puis, une fois terminé, sur toutes les autres. Pour celles-ci, le cache était plein et aucune opération `lookup()` inutile n'avait été envoyée au serveur NFS.

L'expérience, plus réaliste, décrite dans la section précédente donna lieu aux résultats du tableau 2. La colonne *machine* correspond au nom de chacune des 10 machine du site Grid'5000 de Nancy sur lesquelles nous avons lancé le programme de test. Les deux colonnes suivantes comptabilisent, parmi les opérations `lookup()` qui ont échoué, celles qui sont passées par le cache et celles qui sont passées outre, donc par le serveur NFS.

Idéalement, nous aurions un grand nombre d'opérations en cache, donc un fort ratio de cache. Ce n'est pas le cas ici. Nous pensons qu'il y a deux causes à ce résultat :

1. le chargeur de bibliothèques partagées est déterministe, parcourant toujours les mêmes chemins dans le *même ordre*
2. le programme de test démarre environ au même moment sur chaque machine.

En découle que les requêtes `lookup()` identiques seront émises à peu près au même instant sur chaque machine. Chaque machine diffuse ainsi l'information sur une opération échouée à peu près en même temps que les autres tentent cette même opération. Considérant cela, on ne peut attribuer les opération *en cache* dans le tableau 2 qu'au décalage relatif d'avancement de l'exécution sur chaque machine. Si toutes les machines avaient exécuté le programme de test strictement simultanément et à la même cadence, elles n'auraient reçu les entrées de cache qu'après avoir déjà fait les opérations correspondantes, limitant l'intérêt du cache.

5. Conclusion

Cet article a traité de la déferlante d'appels systèmes sur les métadonnées de fichiers dus aux gestionnaires de paquets fonctionnels tels que Nix et Guix, dans le contexte de NixOS-Compose déployé avec un serveur NFS. Nous avons présenté nos travaux en cours sur *Chorage*, un cache intermédiaire distribué visant à filtrer les requêtes de métadonnées inutiles avant qu'elles ne soient transmises au serveur NFS.

Nous envisageons la poursuite de ce travail d'une part par une analyse plus poussée du comportement de *Chorage* sur une charge de travail réelle de NixOS-Compose et d'autre part par la génération d'un cache statique par NixOS-Compose en amont du déploiement.

A. Remerciements

L'expérience présentées dans cet article a été réalisée en utilisant la plateforme d'essai Grid'5000, soutenue par un groupe d'intérêt scientifique hébergé à l'INRIA et incluant le CNRS, RENATER et plusieurs universités ainsi que d'autres organisations (voir https://www.grid5000.fr).

Références

[1] D. BALOUEK, A. CARPEN AMARIE, G. CHARRIER, F. DESPREZ, E. JEANNOT, E. JEANVOINE, A. LÈBRE, D. MARGERY, N. NICLAUSSE, L. NUSSBAUM, O. RICHARD, C. PÉREZ, F. QUESNEL, C. ROHR et L. SARZYNIEC. Adding Virtualization Capabilities to the Grid'5000 Testbed. In I. I. IVANOV, M. van SINDEREN, F. LEYMANN et T. SHAN, éditeurs, *Cloud Computing and Services Science*. Tome 367, Communications in Computer and Information Science, pages 3-20. Springer International Publishing, 2013. ISBN : 978-3-319-04518-4. DOI : 10.1007/978-3-319-04519-1_1.

[2] L. COURTÈS. Functional package management with guix. *arXiv preprint arXiv :1305.4584*, 2013.

[3] L. COURTÈS. Taming the 'stat'storm with a loader cache, 2021. URL : https://guix.gnu.org/en/blog/2021/taming-the-stat-storm-with-a-loader-cache/.

[4] L. K. DOCUMENTATION. FUSE. URL : https://www.kernel.org/doc/html/latest/filesystems/fuse.html.

[5] L. K. DOCUMENTATION. Overlayfs. URL : https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html.

[6] E. DOLSTRA, M. de JONGE et E. VISSER. Nix : A Safe and Policy-Free System for Software Deployment. en :14, 2004.

[7] E. DOLSTRA et A. LÖH. NixOS : A Purely Functional Linux Distribution. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 367-378, New York, NY, USA. Association for Computing Machinery, sept. 2008. ISBN : 978-1-59593-919-7. DOI : 10.1145/1411204.1411255. (Visité le 14/05/2024).

[8] W. FRINGS, D. H. AHN, M. LEGENDRE, T. GAMBLIN, B. R. de SUPINSKI et F. WOLF. Massively parallel loading. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 389-398, 2013.

[9] Q. GUILLOTEAU, J. BLEUZEN, M. POQUET et O. RICHARD. Painless transposition of reproducible distributed environments with NixOS compose. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1-12. IEEE, 2022.

[10] E. JEANVOINE, L. SARZYNIEC et L. NUSSBAUM. Kadeploy3 : Efficient and Scalable Operating System Provisioning. *USENIX ;login :* 38(1) :38-44, fév. 2013.

[11] S. M. KELLY et R. KLUNDT. Shared Libraries on a Capability Class Computer. Rapport technique, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2011.

[12] LINUX. ld.so(8) — Linux manual page. URL : https://man7.org/linux/man-pages/man8/ld.so.8.html.

[13] LUDOVIC COURTÈS. Taming the 'Stat' Storm with a Loader Cache — 2021 — Blog — GNU Guix. URL : https://guix.gnu.org/en/blog/2021/taming-the-stat-storm-with-a-loader-cache/ (visité le 03/10/2023).

[14] R. MACEDO, M. MIRANDA, Y. TANIMURA, J. HAGA, A. RUHELA, S. L. HARRELL, R. T. EVANS, J. PEREIRA et J. PAULO. Taming Metadata-intensive HPC Jobs Through Dynamic, Application-agnostic QoS Control. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pages 47-61, Bangalore, India. IEEE, mai 2023. DOI : 10.1109/CCGrid57682.2023.00015. URL : https://ieeexplore.ieee.org/document/10171504/ (visité le 09/02/2024).

[15] C. A. MACLEAN, H. LEONG et J. ENOS. Improving the Start-Up Time of Python Applications on Large Scale HPC Systems. In *Proceedings of the HPC Systems Professionals Workshop*. SC '17 : The International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1-8, Denver CO USA. ACM, 12 nov. 2017. ISBN : 978-1-4503-5128-7. DOI : 10.1145/3155105.3155107. URL : https://dl.acm.org/doi/10.1145/3155105.3155107 (visité le 08/03/2024).

[16] Mounting the Root Filesystem via NFS (Nfsroot) — The Linux Kernel Documentation. URL : https://www.kernel.org/doc/html/latest/admin-guide/nfs/nfsroot.html (visité le 14/05/2024).

[17] Y. QIAN, W. CHENG, L. ZENG, X. LI, M.-A. VEF, A. DILGER, S. LAI, S. IHARA, Y. FAN et A. BRINKMANN. Xfast : Extreme File Attribute Stat Acceleration for Lustre. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, pages 1-12, New York, NY, USA. Association for Computing Machinery, 11 nov. 2023. DOI : 10.1145/3581784.3607080. URL : https://doi.org/10.1145/3581784.3607080 (visité le 08/03/2024).

[18] R. RUSSELL, D. QUINLAN et C. YEOH. Filesystem hierarchy standard. *V3* :50, 2015.

[19] T. SHAFFER et D. THAIN. Taming metadata storms in parallel filesystems with metaFS. In *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*. SC '17 : The International Conference for High Performance Computing, Networking, Storage and Analysis, pages 25-30, Denver Colorado. ACM, 12 nov. 2017. ISBN : 978-1-4503-5134-8. DOI : 10.1145/3149393.3149401. URL : https://dl.acm.org/doi/10.1145/3149393.3149401 (visité le 08/03/2024).

[20] S. SHEPLER, B. CALLAGHAN, D. ROBINSON, R. THURLOW, C. BEAME, M. EISLER et D. NOVECK. Network file system (NFS) version 4 protocol, 2003.

[21] F. ZAKARIA, T. R. W. SCOGLAND, T. GAMBLIN et C. MALTZAHN. Mapping out the HPC Dependency Chaos. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22, pages 1-12, Dallas, Texas. IEEE Press, 18 nov. 2022.

[22] F. ZAKARIA, T. R. SCOGLAND, T. GAMBLIN et C. MALTZAHN. Mapping out the HPC dependency chaos. In *SC22 : International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1-12. IEEE, 2022.