COLLEGES SCIENCES
BRETAGNE POUR L'INGENIEUR
LOIRE ET LE NUMERIQUE

ENSTA
IP PARIS

THÈSE DE DOCTORAT DE

L'ÉCOLE NATIONALE SUPÉRIEURE
DE TECHNIQUES AVANCÉES BRETAGNE

ÉCOLE DOCTORALE N° 648
Sciences pour l'Ingénieur et le Numérique
Spécialité : *STIC – Informatique*

Par
Louis-Marie Nicolas

Optimisation de l'Énergie des E/S par apprentissage frugal et fédéré pour des applications HPC

Thèse présentée et soutenue à Brest, le 23 janvier 2026
Unité de recherche : Lab-STICC, CNRS, UMR 6285, Équipe SHAKER

Rapporteurs avant soutenance :

Pr. Laurent d'Orazio — Professeur des universités, Université de Rennes
Pr. Alain Tchana — Professeur des universités, Grenoble INP

Composition du Jury :

Président :	Pr. François Trahay	Professeur des universités, Télécom Sud Paris
Examinateurs :	Dr. Francieli Boito	Maître de Conférences, Université de Bordeaux
	Pr. Laurent d'Orazio	Professeur des universités, Université de Rennes
	Dr. Shadi Ibrahim	Chargé de Recherche, Inria Rennes
	Pr. Alain Tchana	Professeur des universités, Grenoble INP
	Dr. Philippe Couvée	Co-encadrant entreprise, Eviden
Dir. de thèse :	Pr. Jalil Boukhobza	Professeur des universités, ENSTA Bretagne

Invité(s) :

Dr. Salim Mimouni — Co-encadrant entreprise, Ryax Technologies

TABLE DES MATIÈRES

Remerciements **7**

Publications **9**

Glossaire **11**

1 Introduction **13**
1.1 Contexte scientifique 13
1.2 Problématiques de recherche 15
1.3 Contributions 16
1.3.1 Contribution 1 : Modélisation et prédiction d'E/S à l'aide de la pile d'appels et de graphes 17
1.3.2 Contribution 2 : Étude de l'effet des indices d'usage de données sur des systèmes de fichiers parallèles 18
1.3.3 Contribution 3 : Étude de l'effet de la variation de la fréquence et de la tension sur des applications haute performance 18
1.4 Cadre des travaux 19
1.5 Plan de la thèse 19

2 Connaissances de base et état de l'art **21**
2.1 Pile logicielle de gestion de fichiers et d'objets dans le HPC 21
2.1.1 Système de fichiers Linux 21
2.1.2 Systèmes de fichiers parallèles et distribués 23
2.1.3 Stockage objet 26
2.2 Usages du stockage par les applications HPC 27
2.2.1 Motifs d'E/S historiques : flux séquentiels dominés par l'écriture . . 28
2.2.2 Motifs d'E/S émergents : accès variés et non structurés 28
2.3 Leviers pour l'optimisation des E/S et de leur énergie dans un contexte HPC 30
2.3.1 Placement de données, préchargement et éviction 30
2.3.2 Variation dynamique de la tension et de la fréquence (DVFS) . . . 32

2.4 Modélisation et prédiction des E/S 34
2.4.1 Approche boîte blanche : Modification du code source applicatif . . 34
2.4.2 Approche boîte noire : Interception des E/S 34
2.4.3 Approche boîte grise : Interception des E/S et de leur pile d'appels 35

3 Contribution 1 : Algorithme frugal de modélisation et de prédiction des E/S à l'aide de la pile d'appels et de graphes 37
3.1 Motivation 37
3.2 Solution proposée : GrIOt, une approche utilisant des graphes et la pile d'appels 38
3.2.1 Instrumentation des E/S et de la pile d'appels 39
3.2.2 Modélisation en ligne à l'aide de graphe 42
3.2.3 Prédiction en ligne des E/S futures 48
3.2.4 Prédiction de la pile d'appels des E/S futures 48
3.2.5 Prédiction des paramètres des E/S 49
3.3 Évaluation expérimentale 50
3.3.1 Applications instrumentées 50
3.3.2 Métriques d'évaluation 51
3.3.3 Méthodologie 53
3.4 Résultats expérimentaux et discussion 53
3.4.1 Comparaison de GrIOt à l'état de l'art 53
3.4.2 Comparaison des différentes configurations de GrIOt 57
3.4.3 Caractérisation de l'instrumentation de la pile d'appels 63
3.5 Conclusion 65

4 Contribution 2 : Étude de l'effet des indices d'usage de données sur des systèmes de fichiers parallèles 67
4.1 Motivation 67
4.2 Proposition : méthodologie d'évaluation de l'effet des indices d'usage de données 69
4.2.1 Caractérisation du système 70
4.2.2 Caractérisation des indices de motifs d'E/S 73
4.2.3 Caractérisation de l'indice `"DONT_NEED"` 74
4.2.4 Caractérisation de l'indice `"WILL_NEED"` 74
4.3 Évaluation expérimentale 75

4.3.1 Ressources matérielles et logicielles . . . 75
4.3.2 Visualisation des résultats expérimentaux . . . 76
4.3.3 Caractérisation du système . . . 76
4.3.4 Caractérisation des indices de motifs d'E/S . . . 78
4.3.5 Caractérisation de l'indice "DONT_NEED" . . . 80
4.3.6 Caractérisation de l'indice "WILL_NEED" . . . 81
4.4 Proposition de consignes d'utilisation des indices d'usage de données . . . 83
4.5 Conclusion . . . 85

5 Contribution 3 : Étude de l'effet de la variation de la fréquence et de la tension sur des applications HPC 87
5.1 Motivation . . . 87
5.2 Proposition : méthodologie d'évaluation de l'effet du DVFS sur des applications haute performance . . . 88
5.2.1 Élaboration d'applications synthétiques pour la caractérisation du DVFS . . . 88
5.2.2 Caractérisation du DVFS durant les états actifs du processeur . . . 91
5.2.3 Caractérisation du DVFS durant les états inactifs du processeur . . . 91
5.3 Évaluation expérimentale . . . 92
5.3.1 Ressources matérielles et logicielles . . . 92
5.3.2 Effet du DVFS durant les états actifs du processeur . . . 93
5.3.3 Effet du DVFS en fonction des C-states . . . 96
5.4 Proposition de consignes d'utilisation du DVFS pendant les phases d'E/S . . . 98
5.5 Conclusion . . . 100

6 Conclusion 103
6.1 Résumé des contributions . . . 103
6.2 Perspective . . . 105
6.2.1 Court terme . . . 105
6.2.2 Long terme . . . 106
6.3 Réflexions finales . . . 106

Bibliographie 109

Remerciements

Je tiens tout d'abord à remercier chaleureusement mon directeur de thèse, Jalil Boukhobza, pour son encadrement, sa confiance et son soutien tout au long de cette thèse. Merci pour sa disponibilité, ses conseils, son exigence scientifique et sa présence sans faille durant ces années de doctorat.

Je tiens également à exprimer ma profonde gratitude aux rapporteurs et aux membres du jury pour l'attention portée à mon travail, pour leurs lectures approfondies, leurs remarques constructives et les échanges enrichissants lors de la soutenance. Je remercie aussi toutes les personnes qui ont participé à cette journée, par leur présence, leurs questions, leurs encouragements ou leur soutien, et qui ont contribué à en faire un moment important.

Je remercie par ailleurs l'ensemble des doctorants, stagiaires, enseignants-chercheurs et personnels de ENSTA Bretagne pour l'ambiance de travail stimulante et conviviale qui a accompagné tous mes séjours à Brest. Un merci particulier à tous ceux avec qui j'ai pu échanger, collaborer, débattre, ou même partager des moments de pause autour d'un café.

Mes remerciements vont aussi à mes collègues de l'équipe Data Management de Bull, et plus particulièrement à Philippe Couvée et Salim Mimouni pour leur accompagnement et leurs conseils tout au long de cette thèse. J'ai eu le plaisir de travailler et d'apprendre au quotidien à leurs côtés. Leur expérience, leur très grande bienveillance ainsi que les nombreuses discussions techniques ont beaucoup compté à la réussite de cette thèse.

Je souhaite également remercier mes amis de Brest comme de Grenoble pour tous les moments partagés, pour m'avoir supporté lorsque je parlais de mes « contribs' » à la coloc comme sur le téléski, pour les discussions, les repas improvisés, les soirées, et pour avoir rendu cette période aussi mémorable qu'agréable.

Enfin, un immense merci à mes parents ainsi que toute ma famille pour leur soutien constant et leurs encouragements pendant cette thèse.

PUBLICATIONS

Article de journal

— Louis-Marie Nicolas, Salim Mimouni, Philippe Couvée, Jalil Boukhobza ; *I/O patterns modeling of HPC applications with call stacks for predictive prefetch*, **Future Generation Computer Systems** (FGCS), Volume 175, Article 108034, 2026.

Conférences et ateliers internationaux

— Louis-Marie Nicolas, Salim Mimouni, Philippe Couvée, Jalil Boukhobza ; *Investigating the Use of File Advisory Hints on Lustre and GPFS*, Papier court, **Proceedings of the 37th International Conference on Scalable Scientific Data Management** (SSDBM), Article n°18, Pages 1 - 6, 2025, Columbus, États-Unis.

— Louis-Marie Nicolas, Salim Mimouni, Philippe Couvée, Jalil Boukhobza ; *Characterizing the use of DVFS for HPC I/O optimization : a Microbenchmarking Approach*, **Proceedings of the 5th Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems** (CHEOPS), Pages 1 - 7, 2025, Rotterdam, Pays-Bas.

— Louis-Marie Nicolas, Salim Mimouni, Philippe Couvée, Jalil Boukhobza ; *GrIOt : Graph-based Modeling of HPC Application I/O Call Stacks for Predictive Prefetch*, **Proceedings of the SC'23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis** (PDSW), Pages 1195 - 1201, 2023, Denver, États-Unis.

Autres publications

— Louis-Marie Nicolas, Luis Thomas, Yassine Hadjadj-Aoul, Jalil Boukhobza ; *SLRL : A Simple Least Remaining Lifetime File Eviction policy for HPC multi-tier storage*

systems, **ACM SIGOPS : Operating Systems Review, Volume 56**, Pages 70 - 76, 2022.

— Louis-Marie Nicolas, Luis Thomas, Yassine Hadjadj-Aoul, Jalil Boukhobza ; *SLRL : A Simple Least Remaining Lifetime File Eviction policy for HPC multi-tier storage systems*, **Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems** (CHEOPS), Pages 33 - 39, 2022, Rennes, France.

GLOSSAIRE

1. Matériel et bas niveau

CPU *Central Processing Unit* — Processeur central.

DRAM *Dynamic Random-Access Memory* — Mémoire principale volatile.

NVMe *Non-Volatile Memory Express* — Interface haute performance pour SSD.

NVRAM *Non-Volatile Random-Access Memory* — Mémoire non-volatile à faible latence.

HDD *Hard Disk Drive* — Disque dur magnétique.

SSD *Solid State Drive* — Disque électronique à mémoire flash.

BMC *Baseboard Management Controller* — Contrôleur de gestion hors bande.

RDMA *Remote Direct Memory Access* — Accès direct à la mémoire distante.

P-states *Performance States* — États actifs de fréquence/tension du processeur.

C-states *Idle States* — États d'inactivité du processeur.

2. Systèmes d'exploitation et interfaces

POSIX *Portable Operating System Interface* — Standard d'API pour systèmes Unix.

VFS *Virtual File System* — Couche d'abstraction des systèmes de fichiers sous Linux.

CPUfreq Sous-système Linux de gestion de la fréquence CPU.

CPUidle Sous-système Linux de gestion des états inactifs du CPU.

LD_PRELOAD Mécanisme Unix permettant d'interposer des bibliothèques partagées.

LibC Bibliothèque standard du langage C.

libunwind Bibliothèque permettant d'obtenir et manipuler la pile d'appels.

DWARF Format standard de métadonnées de débogage.

3. Stockage et systèmes de fichiers

E/S Entrées/Sorties — Traduction francophone de *I/O*.

GPFS *General Parallel File System* (IBM Storage Scale) — Système de fichiers parallèle distribué.

Lustre Système de fichiers parallèle pour HPC.

PFS *Parallel File System* — Système de fichiers parallèle.

4. Applications, bibliothèques et outils

MPI *Message Passing Interface* — Standard de communication pour applications parallèles.

HPC *High Performance Computing* — Calcul haute performance.

GrIOt Modèle de prédiction d'E/S basé sur les piles d'appels (contribution de cette thèse).

Omnisc'IO Modèle d'E/S basé sur des grammaires de piles d'appels.

MFU *Most Frequently Used* — Heuristique d'éviction de cache.

MRU *Most Recently Used* — Heuristique d'éviction de cache.

IOR Outil de mesure de la performance d'E/S en HPC.

FIO *Flexible I/O Tester* — Outil générique de génération de charges d'E/S.

CSV *Comma-Separated Values* — Format de données tabulaire.

CMake Outil de configuration et de compilation multi-plateforme.

STDIO *Standard I/O* — Interface standard d'E/S bufferisées du langage C.

Chapitre 1

Introduction

1.1 Contexte scientifique

Nous assistons à une explosion exponentielle de la demande en calcul, traitement et stockage des données numériques [23] pour un vaste spectre d'acteurs, tels que Google, Facebook, Netflix mais aussi des banques [6] et des programmes scientifiques [36].

En parallèle avec cette augmentation de la génération des données numériques, la performance des processeurs a continuellement augmenté par la fabrication de composants toujours plus denses et d'architectures toujours plus efficaces. Cependant, malgré une performance par Watt en nette hausse [6], la consommation énergétique des supercalculateurs diminue peu, en raison de leur taille croissante. Ce phénomène illustre un *effet rebond* : plus l'on optimise les systèmes, plus la taille et le coût énergétique des supercalculateurs deviennent soutenables, ce qui en augmente l'utilisation. Ainsi, la puissance totale cumulée des 10 supercalculateurs les plus puissants est passée de 63 MW en novembre 2014 [70] à 156 MW en novembre 2024 [71]. La consommation d'un supercalculateur récent peut ainsi atteindre celle d'une petite ville, ce qui rend la question de l'efficacité énergétique incontournable. L'informatique "verte" est par conséquent devenue un enjeu scientifique et industriel majeur.

Bien que les périphériques de stockage consomment relativement peu d'énergie par rapport aux processeurs et à leur refroidissement, des goulets d'étranglement liés au stockage peuvent allonger la durée d'exécution des applications, entraînant une augmentation indirecte de la consommation énergétique des calculateurs. Cela s'explique par la nature des E/S des applications HPC, souvent concentrées et sporadiques [88, 100], et par l'écart de plusieurs ordres de grandeur qui sépare la performance de la mémoire de travail volatile de celle du stockage (latence d'environ 20 ns pour le cache "L3", 80 ns pour de la mémoire "DRAM", et 20 µs pour du stockage "SSD" [59, 13]).

L'émergence de nouvelles technologies de stockage offre toutefois des leviers pour réduire cet écart de performance entre stockage et mémoire de travail mais aussi pour réduire

la consommation énergétique. Ces innovations, qu'elles concernent le support de stockage (comme 3D XPoint [38] jusqu'en 2022), son interface (comme NVMe) ou sa gestion logicielle (par exemple, le stockage objet [34, 64, 4]), sont progressivement intégrées dans des systèmes de stockage qui combinent de multiples technologies de stockage et sont dits hétérogènes. Ceux-ci offrent un compromis entre coût et performance, mais nécessitent l'élaboration de stratégies de placement de données pour que les données les plus utilisées soient toujours placées sur le stockage le plus performant.

L'optimisation des performances et de la consommation énergétique dans les systèmes de stockage hétérogènes a donné lieu à un grand nombre de travaux, portant aussi bien sur la conception et l'étude d'architectures matérielles [38, 45, 65, 13, 12, 72] que sur la gestion logicielle des données[76, 34, 64, 4, 51]. Dans cet état de l'art, plusieurs approches complémentaires se distinguent : les optimisations au niveau matériel [38, 37, 24, 10] (couvrant notamment SSD, mémoires non volatiles, et interconnexions), les politiques de placement de données [18, 35, 29, 49, 89, 69, 95, 75, 81, 63, 44, 1, 60, 11, 67] (incluant politiques de préchargement et d'éviction), et les stratégies de modélisation des E/S des applications [16, 17, 97, 49, 26, 9, 92, 28]. Ces dernières constituent un axe de recherche particulièrement actif, car elles visent à relier le comportement observé d'une application à des décisions d'optimisation informées. Cependant, ces techniques ont des limitations, notamment de passage à l'échelle [9, 92, 49], ne sont pas conçues pour fonctionner sur tout type d'applications [27, 28], ou alors ne peuvent être utilisées directement pour réaliser des prédictions [16, 17, 97].

De plus, au-delà de la modélisation des E/S des applications, les leviers du placement de données ne sont pas tous entièrement caractérisés dans la littérature. Parmi ceux-ci figurent les fonctions d'indice d'usage de données [79, 57], qui peuvent être utilisées par une application pour interagir avec la politique de placement de données d'un système de fichiers parallèle. Ce levier a l'avantage d'être supporté par tout système de fichiers (fonction fadvise de la norme POSIX), et ne nécessite pas de modification du code source. Cependant, son effet n'est pas entièrement défini : la norme POSIX n'indique pas de quelle manière les systèmes de fichiers doivent réagir lorsqu'ils reçoivent des indices d'usage de données, ce qui en limite l'usage par les développeurs d'applications.

Enfin, parmi les leviers de l'optimisation énergétique se trouve la Variation Dynamique de la Tension et de la Fréquence (*Dynamic Voltage and Frequency Scaling*, DVFS). Il s'agit d'un outil commun qui peut être utilisé pour réduire la consommation d'énergie du processeur. Bien que l'effet du DVFS pendant des tâches de calcul soit bien documenté

dans la littérature [15, 101, 19, 65, 31, 21, 47], ce qui lui permet d'être utilisé pour en optimiser la consommation d'énergie, son effet pendant les E/S, et notamment les E/S HPC, ne l'est pas, ce qui représente une opportunité perdue.

En résumé, l'état de l'art met en évidence plusieurs manques persistants à l'intersection du calcul haute performance, du stockage et de l'efficacité énergétique [23, 6, 36, 88, 100, 59, 13]. Si les architectures matérielles et les systèmes de fichiers parallèles ont beaucoup évolué [38, 45, 65, 10, 76, 34, 64, 16, 37, 24], la compréhension et la caractérisation de certains leviers restent limitées. Ainsi, certains leviers d'optimisation tels que les indices d'usage de données [79, 57] ou le DVFS [15, 101, 19, 65, 31, 21, 47] sont encore peu étudiés dans le contexte spécifique des E/S en environnement HPC. De plus, les modèles prédictifs d'E/S de l'état de l'art [49, 26, 9, 92, 28, 27] qui pourraient être utilisés pour les actionner présentent d'importantes limitations, notamment de passage à l'échelle ou de généricité.Ces constats motivent les problématiques de recherche abordées dans cette thèse.

1.2 Problématiques de recherche

Ainsi, malgré les nombreuses avancées dans le domaine du calcul haute performance, plusieurs verrous persistent à l'intersection de la performance, du stockage et de l'efficacité énergétique. D'une part, la modélisation et la prédiction des comportements d'E/S des applications restent limitées en termes de passage à l'échelle et de généricité. D'autre part, les leviers d'optimisation que constituent les fonctions d'indice d'usage de données et le DVFS demeurent encore insuffisamment caractérisés, en particulier dans le contexte des E/S. Afin d'aborder ces limites et de proposer des pistes d'optimisation adaptées aux environnements de calcul intensif, nous formulons les trois problématiques de recherche suivantes :

1. **Comment produire à l'échelle de chaque nœud de calcul un modèle peu coûteux et réutilisable de description et de prédiction des E/S ?** Modéliser et prédire les E/S des applications est nécessaire pour pouvoir actionner des optimisations telles que les fonctions d'indice d'usage de données et le DVFS d'une manière adaptée à chaque application. Ces dernières optimisations ayant pour cible les nœuds de calcul, ces prédictions doivent être réalisées par un modèle directement sur les nœuds de calcul. La puissance de calcul et la mémoire de ces nœuds étant utilisées par les applications, l'entraînement et l'usage de ce modèle

doivent être peu coûteux. Cependant, les propositions de l'état de l'art ont des limitations de passage à l'échelle [9, 92, 49], ou ne sont pas conçues pour fonctionner sur tout type d'applications [27, 28]. D'où le besoin d'un modèle de description et de prédiction des E/S précis et peu coûteux.

2. **Quel est l'effet des fonctions d'indice d'usage de données sur le placement de données ?** Le placement des données est nécessaire pour exploiter pleinement la performance d'un système de stockage hétérogène, et a un effet sur la consommation énergétique durant les phases d'E/S. L'effet des fonctions d'indice d'usage de données n'est pas clairement défini dans la plupart des systèmes de fichiers, ce qui en limite l'usage par les développeurs d'applications. D'où le besoin d'une étude de l'effet de ces fonctions sur le placement des données.

3. **Quel est l'effet du DVFS sur l'énergie et la performance des E/S ?** Le DVFS permet de réduire la puissance consommée par le processeur en sacrifiant sa puissance de calcul. Comprendre l'effet du DVFS sur les différents motifs d'E/S est nécessaire afin de pouvoir l'actionner uniquement pendant les E/S où le gain en énergie est positif et le coût en performance acceptable.

La problématique de la modélisation et de la prédiction des E/S est traitée dans le chapitre 3. Les problématiques portant sur le placement des données et l'optimisation énergétique sont traitées dans les chapitres 4 et 5, respectivement. Une contribution scientifique est présentée par problématique.

1.3 Contributions

Les trois problématiques identifiées structurent les contributions de cette thèse. Chacune vise à lever un verrou identifié dans l'état de l'art : la première concerne la modélisation frugale et généralisable des E/S, la seconde l'évaluation du levier que représentent les indices d'usage de données pour le placement dans les systèmes de fichiers parallèles, et la troisième l'étude du DVFS comme outil d'optimisation énergétique pendant les phases d'E/S. Les contributions correspondant à ces trois axes sont détaillées dans les sections suivantes.

1.3.1 Contribution 1 : Modélisation et prédiction d'E/S à l'aide de la pile d'appels et de graphes

Pour répondre à la problématique de la modélisation et de la prédiction des E/S, nous proposons GrIOt. GrIOt utilise des graphes orientés pour modéliser et prédire les E/S. Dans un graphe créé par GrIOt, un nœud correspond à une fonction d'E/S identifiée par sa pile d'appels. GrIOt peut être utilisé pour prédire les E/S au niveau du fichier, permettant un placement des données adapté.

Le modèle satisfait les contraintes identifiées dans la problématique :

— **Frugalité.** Mettre à jour le modèle revient à ajouter un arc ou un sommet dans le graphe, et prédire la prochaine E/S revient à sélectionner un arc, ce qui peut être réalisé en temps presque constant grâce à une table de hachage, entraînant un ralentissement très faible sur l'application.
— **Réutilisabilité.** Les piles d'appels étant liées au binaire de l'application, le modèle reste valable d'une exécution à l'autre.
— **Généricité.** Les graphes produits localement sur différents nœuds peuvent être fusionnés arête par arête pour obtenir un modèle unique par application, pour tout type d'application.

Contrairement aux approches existantes de modélisation des E/S, qui sont limitées aux applications à E/S régulières ou déterministes [28, 27] ou qui ont des limitations de passage à l'échelle [9, 92, 49], GrIOt s'applique aussi bien aux applications déterministes que non déterministes, et permet la mise à jour du modèle et la prédiction des E/S en ralentissant faiblement l'application.

GrIOt prédit correctement un plus grand volume d'E/S que les solutions existantes comme Omnisc'IO sur tout type d'application, avec un ralentissement de l'application au minimum 30% plus faible que l'état de l'art pour la plupart des applications, une taille de modèle en moyenne 20x plus petite, et une précision de prédiction similaire à l'état de l'art.

GrIOt répondant à la première problématique, la seconde contribution traite de la seconde problématique, l'étude des fonctions d'indice d'usage de données.

1.3.2 Contribution 2 : Étude de l'effet des indices d'usage de données sur des systèmes de fichiers parallèles

Un placement judicieux des données est nécessaire pour pouvoir profiter de toute la performance d'un système de stockage hétérogène. Il est possible pour une application exécutée sur un nœud de calcul d'interagir avec la politique de placement de données d'un système de fichiers parallèle en utilisant des indices d'usage de données, ce qui permet d'optimiser performance et énergie.

Cependant, bien que les créateurs d'outils applicatifs aient des attentes concernant l'effet de chaque indice d'utilisation de données, le comportement des systèmes de fichiers de l'état de l'art lorsqu'ils reçoivent de tels indices est mal documenté, y compris pour des systèmes de fichiers parallèles très populaires.

Afin de répondre à la deuxième problématique de recherche, et de permettre aux développeurs applicatifs une utilisation éclairée de ces indices, une méthodologie d'évaluation de l'effet des indices d'usage de données sur des systèmes de fichiers parallèles est proposée. Cette méthodologie est mise en œuvre dans un outil compatible avec tout système de fichiers, et dont le code source est rendu disponible publiquement. Cet outil est utilisé pour évaluer l'effet d'indices sur deux systèmes de fichiers parallèles très utilisés, Lustre et GPFS, que nous introduisons avec plus de détails dans le second chapitre. Des lignes de conduite pour l'utilisation des indices d'usage de données par les développeurs applicatifs sont tirées de ces résultats.

1.3.3 Contribution 3 : Étude de l'effet de la variation de la fréquence et de la tension sur des applications haute performance

Malgré une efficacité énergétique en hausse, la puissance totale des supercalculateurs augmente [70, 71]. Le coût et l'empreinte écologique de cette énergie justifient la recherche d'optimisations de la consommation énergétique.

Le DVFS est un outil commun déjà utilisé pour optimiser la consommation d'énergie du processeur. Cependant, son effet pour des charges intensives en E/S n'est pas décrit dans l'état de l'art, ce qui limite son utilisation pendant les E/S.

Pour obtenir cette connaissance, une méthodologie d'évaluation de l'effet du DVFS utilisant des micro-applications synthétiques est proposée, et appliquée sur un nœud de

calcul HPC. Les résultats produits sont analysés, et des consignes pour l'utilisation efficace du DVFS dans un contexte HPC sont proposées.

1.4 Cadre des travaux

Cette thèse s'inscrit dans le cadre d'un contrat CIFRE entre l'ENSTA|IP Paris et Atos/Eviden.

Les intervenants académiques impliqués dans cette thèse sont membres de l'équipe SHAKER (Software/HArdware and unKnown Environment inteRactions), qui est intégrée au pôle SHARP (Software, Hardware, Architecture and Processes) du laboratoire Lab-STICC CNRS UMR 6285 (Laboratoire en sciences et techniques de l'information, de la communication et de la connaissance), au site de Brest de l'ENSTA|IP Paris.

Les intervenants non-académiques de cette thèse sont membres de l'équipe "Data Management" du département Recherche et Développement HPC d'Atos/Eviden. Cette équipe est spécialisée dans le développement d'outils d'analyse et d'accélération des E/S pour les grands supercalculateurs. Cette équipe pluridisciplinaire de 18 ingénieurs et chercheurs est impliquée dans de nombreuses collaborations nationales (CEA, INRIA, Université Paris/Saclay...) et internationales (projets européens ASPIDE, LEXIS, SAGE2, IO-SEA,...).

1.5 Plan de la thèse

Dans le chapitre 2, les connaissances de base nécessaires à la compréhension des contributions sont introduites, avec notamment une description des spécificités du stockage HPC et l'état de l'art relatif aux trois problématiques de recherche.

Dans les chapitres 3 à 5, les trois contributions annoncées précédemment dans la Section 1.3.2 sont détaillées, et évaluées sur des applications HPC sur un supercalculateur mis à disposition par le département Recherche et Développement d'Atos/Eviden.

Enfin, le chapitre 6 résume les contributions de cette thèse, identifie les limites des solutions et études réalisées, et met en perspective des possibles travaux futurs dans la continuité des travaux réalisés.

Chapitre 2

Connaissances de base et état de l'art

2.1 Pile logicielle de gestion de fichiers et d'objets dans le HPC

Les infrastructures de calcul haute performance (HPC) reposent sur une pile logicielle complexe, intégrant de nombreux composants dédiés à la gestion efficace des données. Cette pile logicielle permet de relier les applications scientifiques aux ressources matérielles de stockage, tout en assurant la cohérence, la performance et la fiabilité des opérations d'entrée/sortie. L'objet de cette section est de décrire cette pile logicielle. Dans les sections suivantes, il s'agira d'en décrire les usages applicatifs, les possibles leviers d'optimisation, et l'état de l'art des modèles pouvant actionner ces leviers d'utilisation.

2.1.1 Système de fichiers Linux

Le classement des 500 plus puissants supercalculateurs (Top500) met en évidence la prépondérance de GNU/Linux dans les supercalculateurs les plus puissants du monde [71]. C'est pourquoi la première brique logicielle que nous présentons est le système d'accès au stockage de GNU/Linux.

Historiquement, les données sous Linux sont stockées dans des systèmes de fichiers, c'est-à-dire à ce qu'elles sont agencées dans des structures similaires à des arbres contenant des nœuds nommés dossiers, et des feuilles nommées fichiers et contenant les données. Un programme d'un utilisateur peut demander à accéder à ces fichiers, c'est-à-dire faire une E/S, en réalisant un appel système qui est alors exécuté dans le noyau.

Le détail de l'exécution d'une E/S est ainsi illustré sur la Figure 2.1.

Ainsi, dans la pile logicielle Linux, une E/S passe par :

— **A** : L'application utilisateur

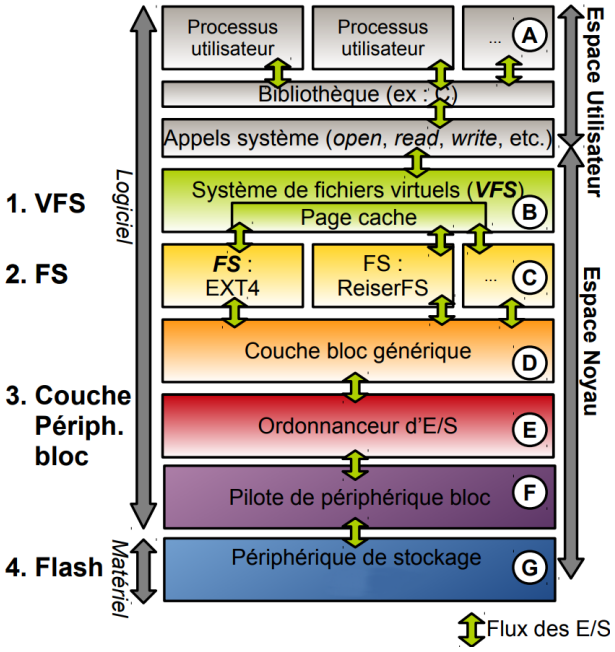


FIGURE 2.1 – La pile logicielle de gestion des E/S sous Linux [10]

— **Entre A et B** : L'interface d'appel système. Les programmes exécutés par l'utilisateur peuvent réaliser des appels système, directement ou en passant par des bibliothèques telles que la Lib-C. On sort alors de l'espace utilisateur pour rentrer dans l'espace noyau, c'est-à-dire que l'exécution passe du mode utilisateur au mode privilégié.

— **B** : Le système de fichiers virtuel (VFS) expose une interface générique pour les applications, et masque les différences de mises en œuvre des différents systèmes de fichiers. Il applique également des optimisations génériques à tous les systèmes de fichiers comme la lecture anticipée, et comprend un cache nommé *page cache*.

— **C** : Le ou les systèmes de fichiers ont pour seule contrainte de devoir respecter l'interface définie par le VFS pour recevoir les requêtes d'E/S, et l'interface de la couche bloc générique pour pouvoir les exécuter.

— **D** : Les systèmes de fichiers n'ont pas à gérer les pilotes de périphériques : la couche bloc générique expose une interface générique utilisable par les systèmes de fichiers pour commander les périphériques de stockage.

— **E** : Les supports de stockage historiques - Bandes magnétiques, HDD - étant plus efficaces en séquentiel (c'est-à-dire quand des données adjacentes dans le stockage sont lues ou écrites les unes à la suite des autres), l'ordonnanceur d'E/S essaye

d'augmenter la séquentialité des E/S en les réordonnant (d'où le terme d'ordonnanceur).

— **F** : Le pilote de périphérique s'interface avec l'ordonnanceur d'E/S pour transmettre les requêtes au périphérique de stockage de manière à ce qu'il les comprenne.

— **G** : Le support de stockage. Il peut avoir sa propre logique de contrôle, à l'image des SSD qui gèrent généralement eux-mêmes le placement des données.

On note tout particulièrement l'existence du *page cache*. Un cache, par définition, est un stockage volatile ou non volatile dans lequel il est possible de copier une ou plusieurs données, de manière à pouvoir y accéder plus rapidement. Le *page cache* est alloué dans la mémoire principale des nœuds de calcul. La mémoire principale étant plus rapide que les supports de stockage [58], si une donnée est dans le cache, alors sa lecture est accélérée.

2.1.2 Systèmes de fichiers parallèles et distribués

Avec un système de fichiers classique tel qu'ext4 et en l'absence de technologie RAID, un fichier unique et ses métadonnées dans le système de stockage Linux sont stockés dans un périphérique de stockage unique. Ce périphérique peut être connecté physiquement à la machine ou accessible par le réseau.

Cependant, ce type de stockage ne propose pas d'outils de synchronisation permettant un grand nombre d'utilisateurs simultanés, et il n'est pas conçu avec comme objectif la résilience aux pannes.

Le **stockage distribué** pallie une partie de ces problèmes : un ou plusieurs nœuds hébergent des serveurs de stockage, chacun connecté à plusieurs périphériques de stockage. Ces serveurs de stockage peuvent alors se charger de la synchronisation et des fonctionnalités de résilience [83], voire même de la gestion des permissions. Les nœuds de calcul peuvent alors se connecter à ces serveurs de stockage de manière transparente, c'est-à-dire comme s'il s'agissait de périphériques de stockage comme les autres, par le réseau.

Toutefois, dans ce cas de figure le débit en lecture ou en écriture reste celui d'un seul support de stockage. Comme illustré sur le tableau 2.2, ce débit est jusqu'à 7 fois plus faible comparé à celui du *page cache* qui se trouve dans la mémoire de travail (2.5 GB/s vs 17 GB/s), ce qui peut représenter un point de ralentissement en calcul haute performance. Pour résoudre ce problème, les systèmes de **stockage parallèle** stockent les fichiers en petits morceaux dits *stripe*. Cela permet alors de lire ou d'écrire une donnée sur plusieurs supports de stockage en parallèle, multipliant ainsi le débit effectif par le

Technologie	Latence	Débit Lecture/écriture	IOPS	Capacité	Coût ~$/GB	Puissance ~W/GB	Endurance DWPD	Durée de Rétention
Registres	< 1 ns	-	-	32/64 bits	-	-	∞	heures
L1 Cache	~ 5 ns	-	-	32+32 KB	-	-	∞	heures
L2 Cache	~ 10 ns	-	-	< 1024 KB	-	-	∞	heures
L3 Cache	~ 20 ns	-	-	8–12 MB	-	-	∞	heures
DRAM	~ 80 ns	17/17 GB/s	-	< 64 GiB	5.000	0.1500	∞	~ 5 ms
NVRAM	~ 5 ţs	2.5/2.5 GB/s	4.6M	< 480 GB	1.200	0.0300	($\sim$1000)	($\sim$10 y)
SSD (NVMe)	~ 20 ţs	8.0/5.0 GB/s	1.2M	< 32 TB	0.200	0.0007	$\sim$ 10-25	> 10 y
SSD	~ 100 ţs	2.1/2.0 GB/s	0.8M	< 8 TB	0.100	0.0018	$\sim$ 10-25	> 10 y
HDD	~ 10 ms	250/240 MB/s	< 500	< 14 TB	0.030	0.0004	-	> 10 y
Bande Magnétique	> 20 s	315/315 MB/s	-	< 15 TB	0.001	-	-	> 30 y

FIGURE 2.2 – Comparaison des technologies de stockage et de mémoire, par J. Luttgau et al. [59]

nombre de supports de stockage lus ou écrits en parallèle, et facilitant l'écriture sur un même fichier depuis plusieurs nœuds.

Pour ces raisons, la plupart des systèmes de stockage utilisés en calcul haute performance sont ainsi parallèles et distribués. Ils sont introduits dans le tableau 2.1.

Lustre [84] est un système de fichiers parallèle (PFS) open-source couramment utilisé dans le calcul haute performance. Comme indiqué dans le tableau 2.1, il est en novembre 2025 répertorié 79 fois dans le classement IO-500 [71], un classement des systèmes de stockage les plus performants ayant exécuté l'outil de mesure de performance éponyme et partagé le résultat. On note que parmi ces 79 occurrences de Lustre dans ce classement, 25 étaient sous le nom Exascaler, qui est une variante propriétaire de Lustre développée par DDN. De plus, on note qu'à cette date, le classement IO-500 ne compte que 284 entrées. Lustre a pour particularité de déployer des services spécialisés, typiquement un par nœud : les nœuds de métadonnées rendent accessibles les métadonnées des fichiers, les nœuds de données rendent accessible le contenu des fichiers, et les nœuds de gestion sont entre autres responsables de la configuration du système de stockage et garantissent que celui-ci ne se fragmente pas en cas de problème réseau.

GPFS [83] est un système de fichiers parallèle (PFS) propriétaire développé par IBM. Des variantes de ce système de fichiers sont aussi connues sous le nom d'IBM Spectrum Scale depuis 2014, et d'IBM Storage Scale depuis 2022. Il figure 25 fois dans le top 500 du classement IO-500. Bien que très similaire à Lustre, l'architecture de GPFS ne repose pas

TABLE 2.1 – Comparaison de différents systèmes de fichiers parallèles et distribués

Nom	Caractéristiques principales	Présence IO-500, sur 284 entrées (Oct. 2024)	Licence
Lustre	Système de fichiers parallèle open-source, architecture modulaire avec services dédiés (métadonnées/MDS, données/OSS, management/MGS), hautement intégré au noyau Linux.	79 occurrences (dont 25 Exascaler)	GPLv2
GPFS / Storage Scale	Système propriétaire IBM basé sur une architecture distribuée sans rôles statiques. Les nœuds peuvent réassigner dynamiquement les fonctions de stockage ou de métadonnées en cas de défaillance.	25 occurrences	Propriétaire
BeeGFS	Système de fichiers parallèle open-source développé par Fraunhofer ITWM et maintenu par ThinkParQ. Fonctionne principalement en espace utilisateur, avec une configuration flexible et un découplage clair entre services de métadonnées et de données.	15 occurrences	GPLv2 (avec extensions commerciales)
GekkoFS	Système distribué éphémère en espace utilisateur, sans stockage persistant partagé. Agrège le stockage local des nœuds de calcul pour créer un espace de travail temporaire à haut débit, minimisant la contention et les coûts d'accès aux PFS globaux.	9 occurrences	MIT

sur des services spécialisés assignés à chacun des nœuds : ceux-ci peuvent se réassigner dynamiquement des rôles (stockage de données, de métadonnées) en cas de défaillance matérielle ou réseau, conférant ainsi au système une résilience particulièrement élevée.

BeeGFS [40] est un système de fichiers parallèle développé initialement par le Fraunhofer Institute for Industrial Mathematics (ITWM). Il est conçu pour offrir une configuration flexible et une administration simplifiée, tout en maintenant des performances élevées. Similairement à Lustre, BeeGFS répartit les données et les métadonnées sur des services distincts. Cependant, contrairement à Lustre, BeeGFS n'est pas profondément intégré au noyau Linux : la majorité de ses composants s'exécutent en espace utilisateur et communiquent via TCP/IP ou RDMA (InfiniBand, Omni-Path). Cela simplifie la maintenance et le déploiement. En novembre 2025, BeeGFS est mentionné 15 fois dans le classement IO-500 [71].

GekkoFS [93] GekkoFS est un système de fichiers distribué et éphémère orienté vers les applications HPC et les workflows scientifiques à forte intensité d'E/S temporaires. Il repose sur une architecture entièrement en espace utilisateur. GekkoFS vise à fournir un

espace de noms local et temporaire à chaque application, réduisant ainsi la contention et les coûts d'accès aux systèmes de fichiers parallèles globaux tels que Lustre ou GPFS. Il n'utilise pas de système de stockage persistant partagé, mais agrège l'espace disque local des nœuds de calcul pour créer un espace de travail distribué à haut débit. GekkoFS est mentionné 9 fois dans le classement IO-500 de novembre 2025.

Des systèmes de fichiers Lustre et GPFS sont disponibles dans le calculateur haute performance du département Recherche et Développement d'Atos/Eviden. Comme observé dans le tableau 2.1, ces deux systèmes de fichiers sont très utilisés. À eux seuls, ils sont mentionnés dans plus du tiers des entrées dans le classement IO-500. Ils sont utilisés dans cette thèse pour évaluer l'impact des contributions sur la performance et l'optimisation énergétique des E/S sur les systèmes de fichiers HPC.

2.1.3 Stockage objet

Le **stockage objet** propose une organisation différente de celle des systèmes de fichiers parallèles (PFS). Dans un PFS, les données sont stockées sous forme de blocs agencés en fichiers et répertoires, selon une hiérarchie exposée directement aux utilisateurs et aux applications. Dans un système de stockage d'objets (*Object Store* en anglais, abrégé en OS), les données sont encapsulées dans des *objets*, qui sont stockés de manière plate dans un espace de noms global. Ces objets sont identifiés par un identifiant unique qui permet de localiser l'objet, et peuvent être associés à un ensemble de métadonnées, décrivant par exemple la taille, la date de création ou encore des attributs applicatifs spécifiques [98, 56, 14].

Avec le stockage objet, les applications exécutées sur les serveurs de calcul n'écrivent pas dans des fichiers exposés par une interface POSIX, mais interagissent avec des identifiants d'objets au travers d'une interface de programmation, qui peut être propre au système de stockage [98], ou standardisée comme par exemple l'interface S3 [74]. Des algorithmes distribués sont chargés de maintenir l'espace de noms global des objets, garantissant que chaque objet soit associé à un identifiant d'objet unique, et inversement. Ne pas utiliser une arborescence, et donner aux clients la possibilité d'adresser directement les données, permet une latence réduite [98].

Cette organisation présente plusieurs différences notables par rapport aux PFS :

— **Accès** : les PFS exposent une API POSIX (fichiers, répertoires), tandis que les systèmes de stockage d'objets exposent généralement une API de type clé-valeur avec un espace de nommage plat.

— **Localisation** : dans un PFS, le système de fichiers gère l'arborescence et l'emplacement des blocs ; dans un *objet store*, c'est un algorithme de distribution (e.g. *Controlled Replication Under Scalable Hashing*, CRUSH [98]) qui détermine le placement.
— **Granularité** : le PFS est optimisé pour des flux séquentiels de blocs, tandis que le stockage objet est pensé pour de nombreux objets accédés indépendants [74, 98].

Plusieurs systèmes de stockage objet sont aujourd'hui disponibles et utilisés dans des environnements HPC. Parmi eux, Ceph et DAOS. **Ceph** [98] est un système de stockage d'objet qui repose sur un algorithme nommé CRUSH pour répartir les objets sur les serveurs de stockage sans nécessiter de serveurs de métadonnées. **DAOS** [56, 14] (Distributed Asynchronous Object Store), développé par Intel, est un autre système de stockage objet conçu spécifiquement pour tirer parti des supports de stockage modernes, notamment les SSD NVMe et les disques Optane [43] (jusqu'à l'arrêt de leur commercialisation).

Enfin, il convient de noter que certains systèmes de fichiers parallèles utilisent eux-mêmes une organisation interne en objets. C'est le cas de Lustre, qui stocke les données dans des *objets* au sein de serveurs de stockage d'objets (*Object Storage Targets, abrégés en OSTs)*, mais qui stocke en parallèle les métadonnées et l'arborescence de ces fichiers dans des serveurs de métadonnées (*Metadata Targets*, abrégés en MDTs). Cela permet aux utilisateurs d'accéder aux données via une interface de type système de fichiers [84, 57].

Ainsi, PFS et systèmes de stockage d'objets ont des caractéristiques propres. Ils sont utilisés par des applications aux profils différents, que nous détaillons dans la section suivante.

2.2 Usages du stockage par les applications HPC

La section précédente a présenté les différentes technologies de stockage et la pile logicielle qui permet leur exploitation dans un contexte de calcul haute performance. Pour cerner les problématiques liées au stockage HPC, il est également nécessaire d'en comprendre les usages. Ainsi, dans cette section, l'attention est portée non plus sur les systèmes de stockage, mais sur les **usages de ces systèmes de stockage par les applications HPC**.

2.2.1 Motifs d'E/S historiques : flux séquentiels dominés par l'écriture

Les applications HPC historiques, principalement issues de la simulation numérique (mécanique des fluides [50], dynamique moléculaire [78, 77, 90], simulation météo [85], etc.), ont été conçues pour exécuter des calculs intensifs et produire des volumes massifs de résultats.

Ces applications de calcul haute performance peuvent s'exécuter pendant plusieurs heures, voire plusieurs jours, sur des centaines voire des milliers de nœuds de calcul. Dans ce contexte, la probabilité d'une défaillance matérielle ou logicielle au cours de l'exécution est non négligeable [82, 30]. Pour limiter la perte de travail en cas de panne, les applications réalisent régulièrement des **points de reprise** (*checkpointing* en anglais).

Le principe consiste à enregistrer, à intervalles réguliers, l'état complet ou partiel de l'application (données, variables, matrices en mémoire) sur le système de stockage. En cas de panne, l'exécution peut être redémarrée à partir du dernier point de reprise, sans avoir à reprendre depuis le début.

Cette opération est coûteuse en E/S, car elle implique l'écriture d'un volume de données proportionnel à la taille mémoire utilisée par l'application, multiplié par le nombre de processus. Pour des applications massivement parallèles, cela peut représenter plusieurs téraoctets écrits simultanément sur le système de fichiers, durant laquelle les machines restent allumées.

Ainsi, du fait de ces contraintes, ces applications ont généralement des motifs d'E/S similaires :

— **Lecture initiale** : les données d'entrée sont lues au début de l'exécution (paramètres de simulation, maillages, conditions initiales du système à simuler).
— **Écritures périodiques** : la majorité des opérations d'E/S consiste en des écritures régulières, soit pour enregistrer des résultats destinés à l'analyse scientifique et à la visualisation, soit pour réaliser les points de reprise.

Ce schéma entraîne des flux séquentiels dominés par l'écriture, où le système de stockage est sollicité de manière ponctuelle mais intensive.

2.2.2 Motifs d'E/S émergents : accès variés et non structurés

L'évolution des usages du calcul intensif, en particulier avec l'intégration croissante de l'intelligence artificielle et de l'analyse massive de données, a conduit à l'émergence

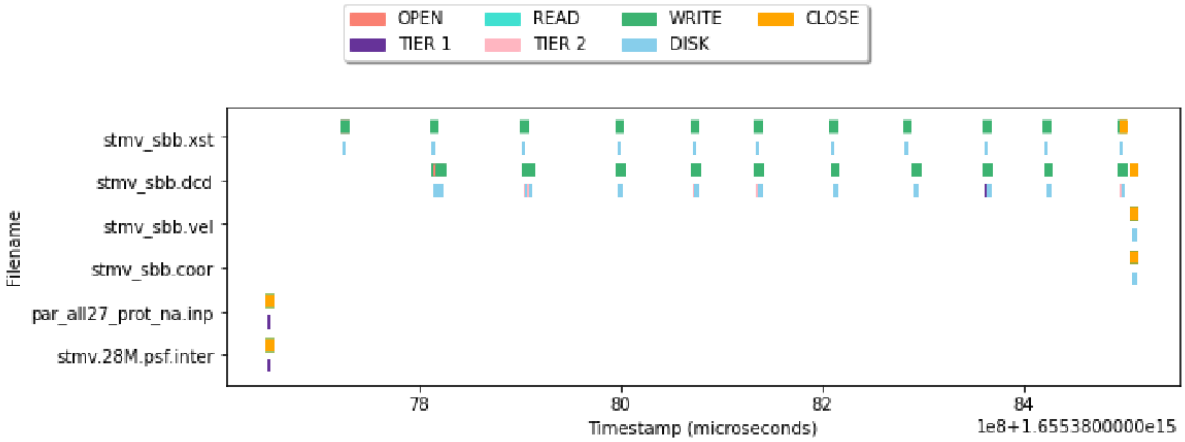


FIGURE 2.3 – Exemple du profil d'E/S de l'application HPC NAMD [78, 77] : une lecture initiale suivie de phases d'écritures séquentielles pour chaque point de reprise. Outil de visualisation FiLiP [46].

de nouveaux motifs d'E/S [8, 25]. Ces charges de travail présentent des caractéristiques différentes de celles des applications de simulation traditionnelles [8, 25, 2, 48] :

— **Multiplicité et variabilité des accès** : au lieu de flux séquentiels longs, on observe une grande variété de motifs différents, y compris des accès aléatoires.
— **Importance des lectures** : l'entraînement de modèles d'apprentissage automatique repose sur l'accès intensif à des ensembles massifs de données existantes, parfois sous forme de petites unités (images, étiquettes, texte).
— **Non-structuration des données** : les données, et notamment les données d'entraînement utilisées en apprentissage profond, ne sont plus nécessairement organisées sous la forme d'une arborescence

Ces nouveaux motifs d'E/S coexistent aujourd'hui avec les usages historiques. Les systèmes de stockage et les optimisations faites sur ces systèmes doivent donc être capables de répondre à cette diversité, allant de l'écriture séquentielle massive pour les simulations classiques aux accès distribués et non structurés, voire non déterministes, des applications d'intelligence artificielle.

Dans ce contexte, de nombreuses optimisations sont proposées pour améliorer la performance des systèmes de stockage.

2.3 Leviers pour l'optimisation des E/S et de leur énergie dans un contexte HPC

L'optimisation de la durée et de la consommation énergétique des applications scientifiques est liée à celle des E/S. En effet, la performance des supercalculateurs dépend non seulement de la puissance de calcul, mais également de la rapidité et de l'efficacité des échanges de données entre les différents niveaux de la hiérarchie mémoire et de stockage. Pour ce faire, la littérature a identifié des leviers permettant de réduire les temps d'accès, de limiter la consommation énergétique et d'adapter dynamiquement le comportement des systèmes aux besoins réels des applications. Parmi ces leviers figurent le placement intelligent des données et le DVFS, qui sont présentés dans cette section.

2.3.1 Placement de données, préchargement et éviction

Pour expliquer le problème du placement de données et son effet sur la performance, il est nécessaire d'introduire les différentes technologies de stockage. Dans la Figure 2.2, on s'intéresse ainsi à la performance et au prix des différentes technologies de stockage volatiles et non volatiles [13, 10, 12, 59] :

1. La mémoire volatile (i.e. la DRAM) offre la meilleure performance, mais a le coût le plus élevé (5$/GB vs 0.03$/GB pour un disque dur).
2. La mémoire flash (SSD) et la mémoire non volatile (NVRAM) offrent la meilleure performance, avec jusqu'à 8 GB/s en lecture séquentielle pour un SSD NVMe.
3. Les disques durs (HDD) ont une performance en lecture séquentielle plus faible que les SSD (250 MB/s vs 8 GB/s pour un SSD NVMe), mais leur coût est jusqu'à 7 fois plus faible à volume équivalent (0.03 $/GB vs 0.2 $/GB pour un SSD NVMe).
4. Les bandes magnétiques offrent un débit très similaire aux HDD, mais avec une latence très importante (>20 s) et à un très faible coût (0.001 $/GB vs 0.03 $/GB pour un HDD).

Ainsi, pour allier coût et performance, les systèmes de stockage HPC combinent plusieurs technologies organisées en tiers de capacité croissante et de performances décroissantes, formant une pyramide [10]. Ces systèmes de stockage sont dits hiérarchiques et hétérogènes (ou hybrides). D'où la problématique du placement de données : sur quel tiers de stockage placer chaque donnée ?

	WILL_NEED	DONT_NEED	SEQUENTIAL	RANDOM
POSIX fadvise	POSIX_FADV_WILLNEED	POSIX_FADV_DONTNEED	POSIX_FADV_SEQUENTIAL	POSIX_FADV_RANDOM
Lustre ladvise	LU_LADVISE_WILLNEED	LU_LADVISE_DONTNEED	NOT SUPPORTED	NOT SUPPORTED

TABLE 2.2 – Fonctions d'indice d'usage de données

L'intuition est de déplacer les fichiers qui seront les plus utilisés dans le futur vers les tiers de stockage plus performants, et inversement. Ainsi dans ces systèmes de stockage, on identifie deux types de mouvements de données :

— Les mouvements d'un tiers de stockage capacitif mais peu performant vers un tiers de stockage plus performant mais de plus faible capacité sont appelés préchargements.
— Les mouvements d'un tiers de stockage plus performant mais de faible capacité vers un tiers de stockage capacitif mais peu performant sont appelés évictions.

Les politiques de préchargement sont basées sur la connaissance des motifs d'E/S et leur prédiction, comme le préchargeur du noyau Linux qui essaye de prédire les E/S séquentielles à l'intérieur des fichiers [12, 13]. Quant à elles, les politiques d'éviction entrent dans deux catégories : les algorithmes d'éviction de cache utilisant des heuristiques, et ceux encore une fois basés sur la connaissance des motifs d'E/S et leur prédiction. Les politiques d'éviction basées sur des heuristiques peuvent combiner la fréquence et/ou récence de l'utilisation des données [20, 32, 52, 44], avec parfois de l'apprentissage pour le choix des métriques à prioriser [81, 95]. À l'inverse, les politiques d'éviction basées sur la connaissance des motifs d'E/S et leur prédiction peuvent directement réaliser un calcul de bénéfice/coût pour le placement de chaque donnée [94].

Au sommet de ces systèmes de stockage hiérarchiques et hétérogènes, dans la mémoire volatile des nœuds de calcul et des nœuds de données, se trouve le page cache Linux. Pour permettre des optimisations, et notamment un meilleur placement des données dans ce cache, Linux expose aux utilisateurs des mécanismes permettant de fournir des indications au PFS sur le comportement attendu des E/S, lui permettant éventuellement d'adapter ses politiques d'éviction et de préchargement. Nous en identifions deux dans le tableau 2.2 : POSIX fadvise et Lustre ladvise. Nous n'incluons pas l'appel système *readahead*, qui permet de précharger des blocs dans le *page cache*, car nous considérons qu'il s'agit d'une fonction impérative et non pas d'une fonction d'indice d'usage de donnée.

POSIX fadvise est un appel système qui permet aux applications de fournir des conseils au noyau sur l'utilisation prévue des données d'un fichier. Il prend en charge

deux types d'indications : les indications sur les motifs d'E/S et les indications relatives au cache. Pour le premier type, un utilisateur peut spécifier que certaines régions d'un fichier seront accédées de manière séquentielle ou aléatoire à l'aide des indications `SEQUENTIAL` et `RANDOM`. Pour le second type, l'utilisateur peut assister la politique de cache du système de fichiers en indiquant que certaines régions d'un fichier seront utilisées ultérieurement avec `WILL_NEED`, qu'elles ne seront plus utilisées avec `DONT_NEED`, ou qu'elles seront utilisées ultérieurement mais une seule fois avec `NO_REUSE`.

Lustre ladvise étend le concept de `POSIX fadvise`, en permettant aux applications s'exécutant sur un nœud de calcul de transmettre des indications équivalentes aux nœuds de données Lustre. Cependant, comme illustré dans le Tableau 2.2, `Lustre ladvise` ne prend en charge que les indications relatives au cache, et ne supporte pas les indications liées aux schémas d'E/S [57].

Les développeurs s'attendent généralement à ce que les données marquées avec l'indice WILL_NEED soient préchargées par le système de fichiers [35], tandis que celles marquées avec DONT_NEED soient évincées du cache [54]. Ainsi, dans la littérature, ces indications sont principalement utilisées pour le préchargement [35], pour l'éviction des données [53], ou encore pour aider le système de fichiers à gérer le cache — par exemple, en évitant que les fichiers journaux en cours d'écriture ne soient conservés en cache [54]. Les indices de motifs d'E/S, quant à eux, peuvent servir à désactiver certaines optimisations telles que le *read-ahead* [35]. De manière générale, ces indications sont régulièrement utilisées dans des applications réelles intensives en E/S, telles que *RocksDB* [33], *GridGraph* [102] ou *SQLite* [86].

Le placement des données et plus spécifiquement les indices d'usage de données ont ainsi un effet sur la consommation énergétique parce que non seulement accéder à différents supports de stockage ne se fait pas nécessairement avec la même consommation énergétique, mais aussi surtout parce que le CPU consomme de l'énergie même quand il ne réalise pas de calcul et qu'il attend des données. Il s'agit du premier levier d'optimisation qui est étudié et décrit dans ce manuscrit.

2.3.2 Variation dynamique de la tension et de la fréquence (DVFS)

Parmi les leviers d'optimisation de la consommation énergétique et de la performance du CPU se trouve la variation dynamique de la tension et de la fréquence (*Dynamic Voltage and Frequency Scaling*, DVFS). En réduisant la fréquence et la tension du CPU, le DVFS permet d'en réduire la puissance consommée. Cependant, cette réduction de la

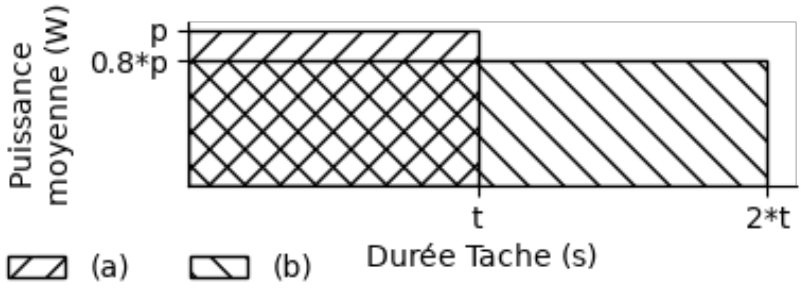


FIGURE 2.4 – Représentation de la durée (axe horizontal), de la puissance moyenne (axe vertical), et de la consommation d'énergie totale (surface hachurée) d'une même tâche exécutée avec deux fréquences CPU différentes (a) et (b)

puissance se fait au détriment de la performance, et entraîne donc une augmentation de la durée des applications.

Ainsi, du fait de la consommation passive des nœuds de calcul (mémoire, réseau, pertes...), la réduction de la fréquence CPU entraîne une diminution de la consommation énergétique si et seulement si la diminution de la puissance du CPU est suffisante pour contrebalancer l'augmentation de la durée de l'application et la consommation passive additionnelle.

Ainsi, dans le cas où la réduction de la puissance CPU n'est pas suffisante pour contrebalancer la consommation énergétique liée à l'augmentation de la durée d'exécution, on observe ainsi une augmentation de la consommation finale et une perte de performance.

Cette possibilité est illustrée Figure 2.4. La durée d'une tâche étant en abscisse et la puissance moyenne du CPU durant l'exécution de cette tâche en ordonnée, les surfaces hachurées représentent donc une quantité d'énergie. Dans cet exemple, il est ainsi visible graphiquement que la tâche exécutée avec la fréquence (b), du fait de l'importante augmentation de la durée de l'application et du maigre impact sur la puissance, consomme finalement plus d'énergie (plus grande surface hachurée) qu'avec la fréquence (a).

Dans la littérature, l'effet du DVFS sur les performances et la consommation d'énergie des tâches de calcul traditionnelles est bien documenté [15, 101, 19, 65]. Il est ainsi connu dans la littérature que la consommation énergétique des tâches dites limitées par le CPU ne peut pas être réduite en baissant la fréquence CPU. À l'inverse, il est possible de réduire la consommation énergétique des tâches dites limitées par la mémoire en baissant la fréquence CPU. Cependant, l'impact sur d'autres activités critiques telles que les opérations d'entrée/sortie (E/S) n'a soit pas été abordé dans la littérature, soit n'a

pas été entièrement caractérisé. Cela est problématique, car la performance et l'efficacité énergétique des E/S ont un impact direct sur la performance et l'efficacité énergétique des applications HPC [45]. C'est pourquoi il s'agit du deuxième levier d'optimisation de ce manuscrit.

2.4 Modélisation et prédiction des E/S

Ainsi, nous avons listé deux leviers de l'optimisation énergétique. Le premier levier, l'utilisation de fonctions d'indice d'usage de données, requiert une connaissance fine des E/S des applications pour que chaque indice d'usage de données envoyé soit correct temporellement et spatialement. Le deuxième levier, l'utilisation du DVFS, requiert également une connaissance fine des motifs d'E/S pour que le DVFS soit appliqué uniquement lorsqu'il est possible de gagner de l'énergie avec un impact sur la performance faible.

Pour obtenir cette connaissance de manière automatique, il existe plusieurs approches dans l'état de l'art. La première passe par une analyse et une modification du code source applicatif, la deuxième par l'interception des E/S, et la troisième par l'interception des E/S et l'instrumentation de leurs piles d'appels.

2.4.1 Approche boîte blanche : Modification du code source applicatif

La première approche permettant d'obtenir une connaissance fine des E/S d'une application repose sur l'analyse et la modification du code source applicatif. Parce que cette approche demande d'avoir accès au code source, cette approche est dite **boîte blanche**. Cette approche permet notamment après analyse du code de placer des primitives de préchargement (déplacement de données d'un stockage lent vers un stockage plus rapide) directement dans le code source, un peu avant un accès [26, 75].

Cette approche est limitée par le fait que le code source applicatif n'est pas toujours disponible. Dans ce cas de figure, il reste cependant possible d'intercepter les E/S. C'est ce qui est proposé avec la seconde approche.

2.4.2 Approche boîte noire : Interception des E/S

La deuxième approche, dite boîte noire, repose sur l'interception des E/S d'une application sans en connaître le code source. Cette approche nécessite l'utilisation de traceurs,

parfois coûteux, qui sont combinés à des modèles de prédiction. Ces modèles de prédiction peuvent être probabilistes [94, 49], ou encore basés sur des séries temporelles ou sur la similarité de motifs (*pattern matching*) [39, 9, 87, 55].

La principale limitation de cette approche est qu'elle permet difficilement de capter la logique de l'application, contrairement à l'approche boîte blanche qui a accès au code source. De plus, certaines approches, notamment de *pattern matching*, se heurtent à des limitations de passage à l'échelle car leur coût dépend de la taille de la fenêtre temporelle étudiée et du nombre de motifs d'E/S mémorisés. Ce qui justifie la troisième et dernière approche.

2.4.3 Approche boîte grise : Interception des E/S et de leur pile d'appels

Similairement à l'approche boîte noire, la dernière approche intercepte les E/S et ne requiert pas un accès au code source des applications. La différence se fait durant l'interception des E/S, où en plus des paramètres des E/S, la pile d'appel des E/S est instrumentée.

La pile d'appels (*call stack* en anglais) est une mémoire propre à chaque processus, dans lequel le système enregistre l'adresse de retour à chaque appel de fonction, selon une logique de type Dernier Entré, Premier Sorti (*Last In First Out*, LIFO). Lorsque l'exécution d'une fonction se termine, l'adresse de retour située au sommet de la pile d'appels est utilisée (puis retirée) afin de poursuivre l'exécution des instructions suivantes.

Ce mécanisme est illustré à la Fig. 2.5, où la pile d'appels obtenue lors de l'appel à une fonction d'instrumentation est représentée tout au long de l'exécution d'une application simple. Si on se concentre sur l'appel à la fonction *bar*, on observe qu'entre t=2 et t=3, pendant l'exécution de la fonction, une adresse de retour vers *bar* est comme décrit dans le paragraphe précédent ajoutée à la pile d'appels. Cette adresse de retour est ensuite retirée (et utilisée) entre t=3 et t=4, au moment du retour de la fonction *bar*, permettant ainsi de reprendre l'exécution de la fonction précédente.

Un instantané de la pile d'appels peut être obtenu à l'aide de la fonction $backtrace(void* buffer[.size], intsize)$, issue de l'extension GNU C, ou par l'utilisation de la bibliothèque *libunwind*. Dans les deux cas, la fonction d'instrumentation de la pile d'appels prend en paramètre un nombre maximal d'adresses de retour à extraire depuis le sommet de la pile. Il convient de noter que, puisque l'instrumentation de la pile d'appels s'effectue via un

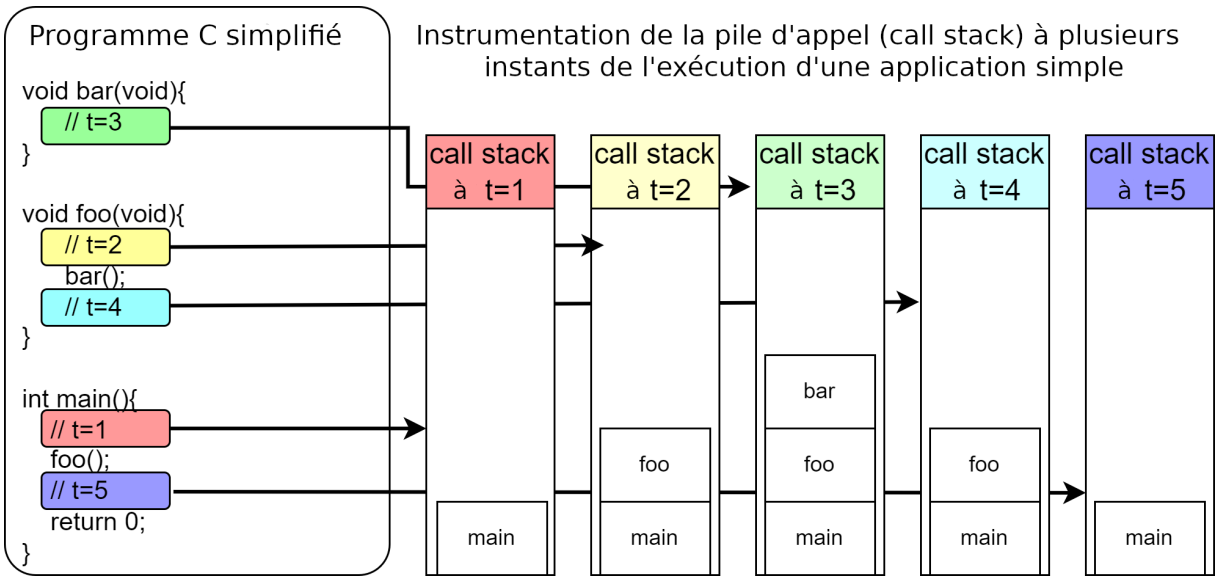


FIGURE 2.5 – Representing the call stacks at different times on a simple application

appel de fonction, l'adresse de retour de la fonction en cours d'exécution est temporairement ajoutée au sommet de la pile. C'est pour cette raison que la pile instrumentée à t=1 dans la Fig. 2.5 contient une adresse de retour vers *main*.

Parce que la pile d'appel contient des pointeurs vers l'ensemble des fonctions qui ont mené à l'exécution d'une E/S, la pile d'appel d'une E/S permet d'identifier de manière formelle chaque E/S dans le code qui l'exécute. Ainsi, l'approche boîte grise permet similairement à l'approche boîte blanche de comprendre dans une certaine limite la logique interne de l'application, sans nécessiter un accès au code source.

L'état de l'art de cette approche boîte grise est Omnisc'IO [27, 28]. Cependant, Omnisc'IO est conçu pour les applications aux E/S déterministes, et utilise une version modifiée de Sequitur [68] nommée StarSequitur [28] pour encoder son modèle. Il s'agit d'un algorithme de compression sans perte, ce qui rend le modèle créé par Omnisc'IO très précis, mais qui en augmente grandement la taille, augmente le coût de prédiction, et réduit la précision, quand les E/S de l'application modélisée sont non déterministes. Ainsi, Omnisc'IO n'est pas conçu pour être utilisé avec tout type d'application. C'est pourquoi dans le chapitre suivant, nous proposons un nouvel algorithme de modélisation et de prédiction des E/S qui utilise un graphe orienté (compression avec perte) plutôt que Sequitur (compression sans perte) pour sa modélisation, permettant une utilisation sur tout type d'applications, déterministes ou non déterministes.

Chapitre 3

Contribution 1 : Algorithme frugal de modélisation et de prédiction des E/S à l'aide de la pile d'appels et de graphes

3.1 Motivation

Dans la Section 1.2, deux leviers de l'optimisation de l'énergie des E/S ont été identifiés. Ces leviers sont les suivants :

— **Placement de données, par l'utilisation, notamment, des indices d'usage de données.** Pour minimiser le temps passé par le CPU à attendre des données, temps durant lequel le CPU continue de consommer de l'énergie, les fichiers qui seront utilisés dans un futur proche doivent être dans les supports de stockage les plus performants. Afin de pouvoir prédire quels seront ces fichiers très utilisés, des heuristiques ou une connaissance des E/S des applications sont nécessaires.

— **La réduction de la fréquence CPU durant certaines E/S.** Baisser la fréquence d'un cœur CPU en réduit la consommation énergétique mais aussi la performance. Des résultats préliminaires montrent que tous les motifs d'E/S ne sont pas affectés par une baisse de la fréquence du CPU de la même manière. Pour pouvoir baisser la fréquence CPU uniquement quand cela permet des gains d'énergie avec un impact faible sur la performance et la restaurer à sa valeur initiale juste après, la connaissance des motifs d'E/S des applications est nécessaire.

Après identification de ces leviers de l'optimisation de l'énergie des E/S, nous sommes arrivés à la conclusion qu'un modèle de description et de prédiction des E/S était nécessaire à leur bonne utilisation, d'où la problématique de recherche : **Comment produire à l'échelle de chaque nœud de calcul un modèle frugal, réutilisable et fédérable**

de description et de prédiction des E/S ?

Des modèles de description et de prédiction des E/S existent dans l'état de l'art [26, 75, 94, 49, 9, 87, 55, 92, 27, 28, 22]. Dans la Section 2.4.3, ils sont classés comme appartenant à une des trois approches boîte blanche, boîte noire, et boîte grise. L'approche boîte blanche nécessite l'accès au code source, l'approche boîte noire nécessite l'instrumentation des E/S, et l'approche boîte grise est une variante de l'approche boîte noire dans laquelle la pile d'appels des fonctions d'E/S est aussi instrumentée.

Cependant, comme analysé dans la Section 2.4.3, les mises en œuvre dans l'état de l'art de ces trois approches ont des limitations en termes de passage à l'échelle (complexité algorithmique), ou des contraintes liées aux applications (besoin d'un accès au code source ou d'E/S déterministes).

C'est pourquoi nous proposons GrIOt, une nouvelle approche boîte-grise permettant une modélisation et une prédiction peu coûteuses. Cette approche ne nécessite pas un accès au code source des applications et n'a pas de restrictions quant à la nature des E/S des applications.

3.2 Solution proposée : GrIOt, une approche utilisant des graphes et la pile d'appels

GrIOt (Graph-based Modeling of I/O call stacks for Predictive Prefetch) est un modèle "boîte-grise" qui partage de nombreux points communs avec l'état de l'art de cette approche. Notamment, GrIOt instrumente non seulement les E/S, mais aussi leur pile d'appels, ce qui permet de distinguer les E/S selon leur origine dans le code de l'application. De plus, la modélisation et la prédiction sont possibles pendant l'exécution d'une application, même sans connaissance préalable au sujet de l'application.

Sa spécificité par rapport à cet état de l'art de l'approche boîte-grise est l'utilisation de graphes orientés plutôt qu'un algorithme de compression sans perte tel que Sequitur [68] ou StarSequitur [28]. Ce modèle n'est pas sans perte, ce qui lui permet de dépasser les limites de l'état de l'art listées dans les sections 3.1 et 2.4.3 : GrIOt supporte tout type d'applications, que leurs E/S soient déterministes ou non déterministes, et ne nécessite pas un accès au code source des applications.

Comme illustré dans la figure 3.1, la stratégie GrIOt peut être décomposée en trois étapes majeures :

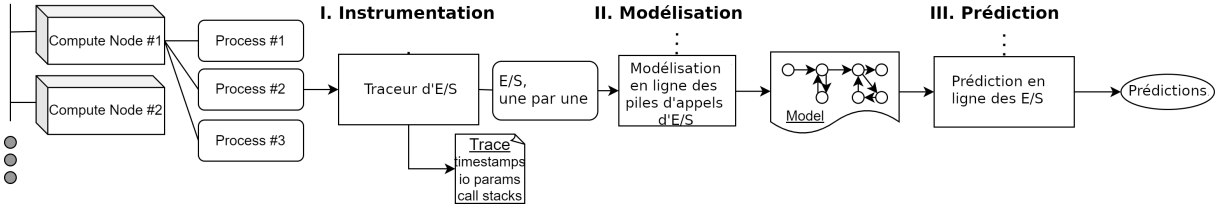


FIGURE 3.1 – Vue d'ensemble de GrIOt

— L'interception des E/S : Les appels de fonction d'E/S de chaque processus sont interceptés à l'aide d'un intercepteur d'E/S en espace utilisateur personnalisé (étape I).
— La modélisation des E/S : Les appels de fonction d'E/S et leurs piles d'appels correspondantes sont capturés dans un graphe orienté (étape II). Ce graphe est créé et mis à jour en ligne pendant l'exécution de l'application. Certaines métadonnées, telles que la fréquence et la récence des arêtes utilisées, sont enregistrées selon l'heuristique de prédiction en ligne utilisée.
— La prédiction en ligne : Les prédictions d'E/S sont effectuées à partir du graphe construit en suivant deux heuristiques [1] (étape III) : Arête la plus récemment utilisée (Most Recently Used, MRU) et Arête la plus fréquemment utilisée (Most Frequently Used, MFU).

Les 3 étapes de GrIOt sont effectuées en ligne (c'est-à-dire pendant l'exécution de l'application, dès sa première exécution) chaque fois que l'application effectue un appel de fonction E/S. L'Alg. 1 montre la séquence de ces étapes, il est référencé dans les explications qui suivent.

3.2.1 Instrumentation des E/S et de la pile d'appels

La première étape de GrIOt, comme représenté par le bloc de pseudocode (I) de la figure 3.1, est l'interception des appels des fonctions d'E/S POSIX et Lib-C qui sont liées dynamiquement, suivie par l'instrumentation de la pile d'appels. L'interception des fonctions d'E/S peut être réalisée avec LD_PRELOAD, une technique d'instrumentation très utilisée dans la littérature [97, 16, 17]. Parce que les outils couramment utilisés dans l'état de l'art tels que Recorder [97] ou Darshan [16, 17] interceptent les E/S mais

1. Le modèle pourrait être éventuellement adapté à des heuristiques hybrides ou utilisant l'apprentissage par renforcement telles que CACHEUS [81], ARC [63], ou LIRS [44]

Algorithme 1 Algorithme de modélisation et de prédiction GrIOt

function ON_IO_INTERCEPTED(*io*, *context_size*, *prediction_heuristic*)
    #### (I) Prétraitement des piles d'appels
    Obtenir, prétraiter et hacher la pile d'appels de l'E/S
    #### (II) Modélisation des E/S
    Mettre à jour le nouveau contexte
    Si nécessaire, créer un nœud ou une arête dans le graphe à partir du nouveau contexte
    Mettre à jour les métadonnées de récence ou de fréquence du graphe selon *prediction_heuristic*
    Mettre à jour la *table de hachage des E/S* pour le nouveau hachage de pile d'appels
    #### (III) Prédiction en ligne
    Si le nombre d'arêtes sortantes du nœud courant == 0 **Alors**
        Retourner *La pile d'appels du nœud courant*
    Sinon Si le nombre d'arêtes sortantes du nœud courant == 1 **Alors**
        Retourner *La pile d'appels du seul nœud suivant*
    Sinon
        Retourner *Pile d'appels sélectionnée selon prediction_heuristic*

```
#include <stdio.h>
#include <dlfcn.h>

// 0: Initialisation du traceur. Obtention d'un pointeur vers la fonction read originale
ssize_t (*iotracer_safe_read)(int fd, void *buf, size_t count)
    = dlsym(RTLD_NEXT, "read");

// 1: Interception des appels a la fonction read, avec tous les param tres
ssize_t read(int fd, void *buf, size_t count){

    // 2: Appel de la fonction read originale
    ssize_t return_value = iotracer_safe_read(fd, buf, count);

    // 3: Instrumentation de la pile d'appels de la fonction d'E/S, via libunwind ou backtrace
    uint32_t call_stack_hash = get_call_stack();

    // 4: La pile d'appels et les param tres de la fonction sont disponibles.
    //      Mettre a jour un mod le ou effectuer une prediction ici.
    fprintf(iotracer_trace_file, " crire  quelque chose, effectuer une action.");
    return return_value;
}
```

FIGURE 3.2 – Exemple d'interception de la fonction `read` à l'aide de `LD_PRELOAD` et *dlsym*

n'instrumentent pas la pile d'appels, nous avons créé notre propre outil.

Ainsi, ce nouvel outil a pour première étape l'interception des appels de fonction d'E/S. Ainsi, quand une application souhaite appeler une fonction d'E/S, une fonction avec un nom et des paramètres similaires à l'intérieur de notre traceur est appelée à la place. Dans cette fonction de notre traceur, la fonction d'E/S originale est appelée, et est suivie par un appel à la fonction C *backtrace* afin d'obtenir la pile d'appels. Comme préalablement décrit dans le chapitre précédent, cette pile d'appels consiste en la liste des adresses de retour des appels de fonction. Cette liste inclut l'adresse de l'appel à la fonction d'E/S. Dans le cas de l'exemple illustré figure 3.3, il s'agit d'un appel à la fonction *read* à la ligne 2 de la fonction *main*.

Un exemple de traceur codé en langage C est présenté dans la Figure 3.2. Dans (1), on voit que la fonction d'E/S *read* est redéfinie, pour que les appels à la fonction *read* (par exemple ceux réalisés par le code dans la Figure 3.3) soient interceptés, avec tous

leurs paramètres. Dans (2), la fonction d'E/S originale est appelée. En (3), la pile d'appels est instrumentée. Ainsi, en (4), la pile d'appels (obtenue à l'étape 3) et les paramètres de la fonction d'E/S (obtenus à l'étape 1) sont disponibles pour réaliser modélisation et prédiction des E/S.

Cependant, il doit être rappelé que les adresses de retour dans la pile d'appels ainsi obtenue ne sont pas les mêmes d'une exécution d'une application à une autre du fait de l'utilisation courante d'une technique nommée *address space layout randomization* [61] qui place les binaires d'une application et de ses dépendances à des endroits aléatoires de la mémoire pour minimiser les vulnérabilités liées à la mémoire.

De fait, les travaux existants utilisant la pile d'appels se concentrent sur la prédiction en ligne et n'étudient pas dans le détail la réutilisation de leurs modèles à base de pile d'appels [27, 28, 22]. Nous pensons que bien que le nombre d'applications uniques exécutées sur un supercalculateur dépende du domaine d'étude, il est commun d'observer un nombre restreint d'applications exécutées de manière régulière, par exemple dans le domaine de la prédiction météo. Cette observation justifie le besoin de disposer de modèles pouvant être réutilisés d'une exécution à une autre.

Pour permettre cela, nous introduisons une méthode permettant de rendre les piles d'appels des fonctions d'E/S identiques d'une exécution d'une application à une autre. Les adresses de retour contenues dans la pile d'appels sont rendues relatives aux adresses de chargement du binaire de l'application et de ceux de ses bibliothèques, ce qui les rend identiques d'une exécution d'une application à une autre tant que le binaire de l'application et de ses bibliothèques ne change pas. Ces adresses de chargement sont disponibles directement dans le fichier */proc/[pid]/maps* sous Linux. Cette pile d'appels modifiée, étant reproductible d'une exécution à une autre, permet la réutilisation du modèle de GrIOt.

Afin de pouvoir déterminer rapidement si deux instantanés de la pile d'appels sont identiques, et plus tard pour rendre la recherche dans le modèle de graphe de GrIOt plus rapide, GrIOt calcule enfin un identifiant unique pour chaque valeur unique de la pile d'appels en utilisant une fonction de hachage non cryptographique, une mise en œuvre 64-bits de Murmur 2 [66].

Dans les figures de ce chapitre, parce qu'un entier à 20 chiffres tel que ceux produits par la fonction de hachage est peu lisible, les identifiants produits à partir de la pile d'appels de chaque fonction d'E/S sont représentés par des lettres majuscules telles que "A", "B" ou "C", et ils sont appelés "identifiant de pile d'appels".

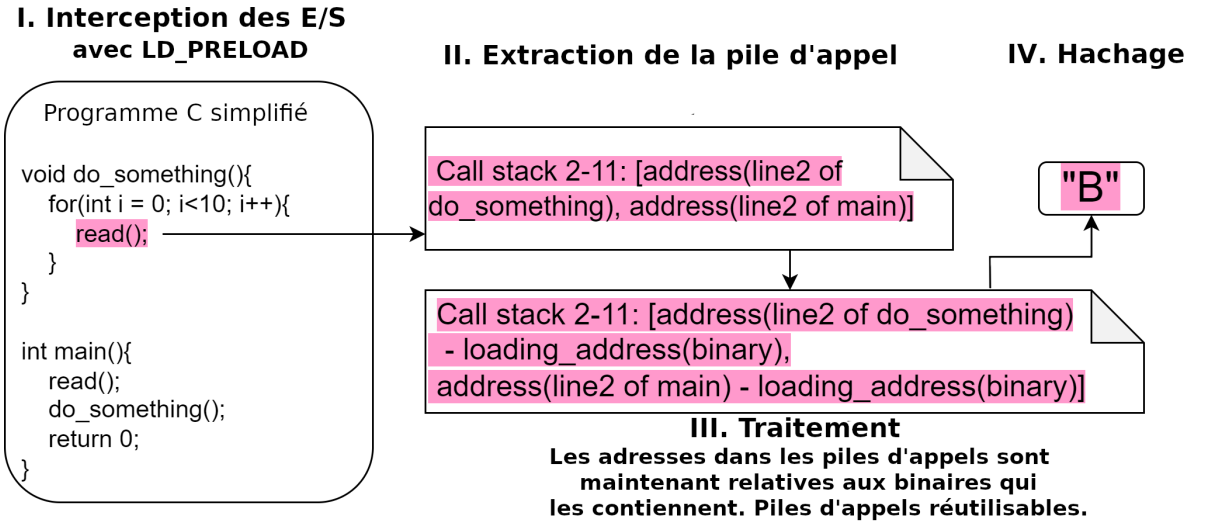


FIGURE 3.3 – Call stack preprocessing

Ainsi, après ce processus et ses 4 sous-étapes, le traceur de GrIOt possède non seulement les paramètres des fonctions d'E/S qui ont été interceptées (comme illustré dans la figure 3.2), mais aussi un identifiant relatif à la pile d'appels de chacune de ces fonctions. Ces informations, comme illustré dans la figure 3.1, peuvent être écrites en ligne dans un fichier de trace. Cependant, parce que GrIOt est un algorithme exécuté en ligne, cette fonctionnalité est désactivée par défaut et ces données sont transmises directement à l'algorithme de modélisation de GrIOt sans être stockées. Cependant, si à des fins d'analyse il était souhaitable de produire des traces d'E/S comportant des identifiants de pile d'appels d'E/S, GrIOt peut être configuré pour le faire.

3.2.2 Modélisation en ligne à l'aide de graphe

À la fin de l'étape I de la figure 3.1, à chaque E/S sont associés des paramètres d'E/S et un identifiant de pile d'appels. À l'étape II, qui est la phase de modélisation des E/S, ces informations sont utilisées pour créer un modèle composé de deux parties :

— Un graphe dirigé d'identifiants de piles d'appels d'E/S. Il est chargé de modéliser la séquence d'identifiants de pile d'appels d'E/S observée lors de l'exécution de l'application.

— Un tableau de paramètres d'E/S. Il est chargé d'associer des paramètres d'E/S tels que la taille et la position des E/S à chaque identifiant de pile d'appels d'E/S unique. Ces grandeurs sont nécessaires pour que le modèle soit utilisable pour la prédiction des paramètres d'E/S à utiliser pour le préchargement.

Ensemble, ces deux éléments peuvent être utilisés pour modéliser une séquence d'E/S. Ils sont détaillés dans les sous-sections ci-dessous.

De plus, cette modélisation des E/S peut être effectuée à plusieurs niveaux. Les travaux de l'état de l'art se concentrent sur la création d'un modèle d'E/S par processus. Cependant, cette granularité de modélisation ne permet que de prédire les E/S d'un processus de l'application, mais ne permet pas de prédire spécifiquement les prochaines E/S d'un fichier cible. Or la localité spatiale des E/S est importante pour le préchargement et plus généralement le placement des données. Afin de pouvoir prédire sélectivement les prochaines E/S d'un fichier cible, nous proposons deux granularités de modélisation alternatives, plus fines, par fichier. Les trois granularités de modélisation prises en charge par GrIOt sont donc les suivantes :

— Un modèle **par processus**. Un graphe de pile d'appels est créé par processus. Le modèle peut prédire les prochaines E/S du processus pour lequel il a été créé.
— Un modèle **par ouverture de fichier**. Un nouveau modèle est créé pour chaque nouveau descripteur de fichier. Contrairement au modèle précédent, cette granularité de modélisation permet de faire des prédictions locales à chaque fichier dans lequel un modèle a été créé. De plus, les métadonnées de prédiction telles que la récence et la fréquence sont également locales à chaque fichier ouvert, ce qui affecte la prédiction. Un nouveau modèle est créé dans un processus chaque fois qu'une fonction de type "ouverture" est appelée. Les modèles sont supprimés lorsque les descripteurs de fichiers sont fermés.
— Un modèle **par identifiant de pile d'appels d'ouverture de fichier**, raccourci en **par pile d'appels d'ouverture de fichier** dans ce chapitre. Cette granularité de modélisation, similaire à la précédente, permet une prédiction par fichier et des métadonnées par fichier. La principale différence est que le modèle est partagé entre les descripteurs de fichiers qui ont été ouverts avec un appel de fonction à la pile d'appels similaire, et que le modèle n'est pas supprimé avant la fin du processus. Le programme doit donc garder en mémoire un modèle pour chaque pile d'appels d'ouverture de fichier, mais peut réutiliser des modèles de fichiers.

La conception et la mise en œuvre du modèle GrIOt avec ces granularités de modélisation sont détaillées ci-après.

Graphe dirigé de piles d'appels d'E/S

Au cœur du modèle de GrIOt se trouve un graphe dirigé qui modélise la séquence des piles d'appels d'E/S.

L'approche naïve pour modéliser la séquence d'identifiants de pile d'appels d'E/S consiste à créer un graphe dirigé, dans lequel un nœud est un identifiant de pile d'appels unique (c'est-à-dire un appel de fonction d'E/S unique). Une arête est créée entre deux nœuds à l'exécution pour indiquer que deux fonctions d'E/S avec les identifiants de piles d'appels correspondants ont été appelées dans l'ordre. C'est la première solution de modélisation que nous considérons.

Ce modèle, bien que simple à construire, présente quelques limitations. Par exemple, si un identifiant de pile d'appels donné peut être suivi par deux ou plusieurs identifiants de pile d'appels différents dans le graphe (plusieurs arêtes sortantes), la prédiction est plus difficile à inférer.

Un tel cas est illustré dans la figure 3.4. Dans cet exemple, une séquence d'identifiants de pile d'appels est créée : "ABACAB", c'est-à-dire une E/S d'identifiant de pile d'appels notée "A", suivie d'une E/S d'identifiant de pile d'appels notée "B", et ainsi de suite. Lorsque le dernier identifiant de la séquence est atteint ("A", en vert dans la figure), le prédicteur n'a aucun indice pour savoir lequel des nœuds "B" ou "C" suivre car il y a deux arêtes sortantes. De plus, ce modèle ne prend pas en compte le fait que l'identifiant de pile d'appels de la prochaine E/S ne peut pas nécessairement être déterminé uniquement à partir de l'identifiant de pile d'appels de la dernière E/S.

Le problème précédent peut être résolu en créant des nœuds de graphe qui contiennent plus d'informations sur l'historique d'une E/S afin de faciliter le choix lorsqu'il existe plusieurs alternatives d'arêtes sortantes. C'est la deuxième solution de modélisation que nous considérons. Pour ce faire, le modèle peut stocker dans le nœud du graphe non seulement l'identifiant de pile d'appels de l'E/S capturée, mais aussi les quelques derniers qui la précèdent. L'identifiant de pile d'appels du nœud plus l'historique stocké est appelé le **contexte** et le nombre d'identifiants de pile d'appels par nœud la **taille du contexte**. Deux nœuds de graphe différents peuvent avoir le même identifiant de pile d'appels s'ils ont un historique différent.

Par exemple, dans la figure 3.5, une taille de contexte de deux est choisie, ce qui signifie qu'un nœud contient à la fois l'identifiant de pile d'appels de la fonction d'E/S actuelle plus ceux qui la précèdent. En raison de ce changement, le nœud qui était appelé "A" dans la figure 3.4 est divisé en deux nœuds "BA" et "CA", c'est-à-dire le nœud contenant l'E/S

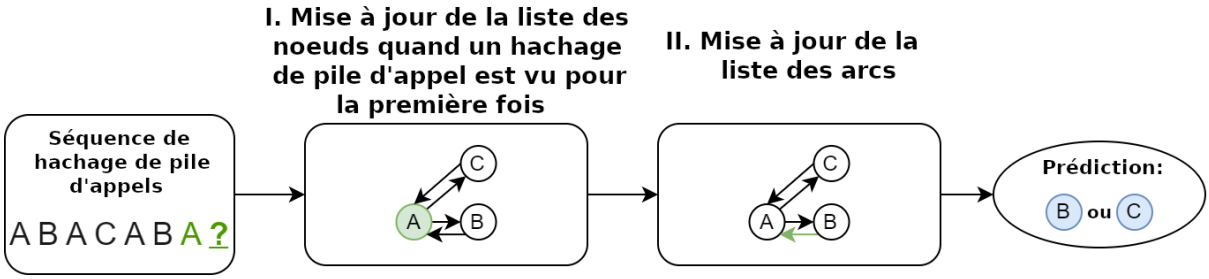


FIGURE 3.4 – Élaboration en ligne d'un graphe d'identifiants de pile d'appels avec une taille de contexte de 1

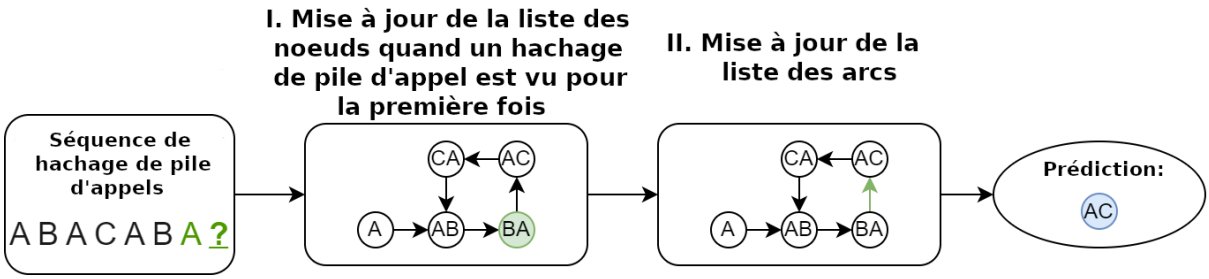


FIGURE 3.5 – Élaboration en ligne d'un graphe d'identifiants de pile d'appels avec une taille de contexte de 2

liée à la pile d'appels "A" qui était précédée par "C" et celle liée à la pile d'appels "A" qui était précédée par "B", respectivement (la dernière lettre représente le hachage actuel et celles qui le précèdent sont son historique). En raison de cela, une arête entre "BA" et "AC" signifie que la séquence de pile d'appels d'E/S "BAC" a été observée, ce qui signifie également qu'après un "A" qui était précédé par un "B", "C" est une prédiction possible.

En utilisant ce graphe, nous observons qu'il est maintenant possible de prédire la pile d'appels suivante sans ambiguïté. Plus la taille du contexte est grande, plus le modèle est précis. Cependant, ajouter de l'historique à chaque nœud a deux impacts : 1) la taille du nœud augmente car il doit stocker plus d'identifiants de pile d'appels, et 2) la taille du graphe augmente car il peut contenir plus de nœuds. En conséquence, il y a un compromis à trouver pour déterminer la meilleure taille de contexte pour une charge de travail donnée. Cela est évalué dans la partie expérimentale.

Tableau des paramètres d'E/S

Le graphe des identifiants de pile d'appels d'E/S modélise la séquence des piles d'appels d'E/S et permet la prédiction de la pile d'appels des E/S futures, mais ne permet pas de

déterminer les paramètres des E/S. Il s'agit du rôle du tableau des paramètres d'E/S.

Pour rappel, lors de l'instrumentation de chaque E/S, nous capturons non seulement la pile d'appels, mais aussi les paramètres d'E/S de la fonction d'E/S concernée. Ces paramètres sont stockés dans un tableau (une ligne par identifiant de pile d'appels) et contiennent toutes les métadonnées qui peuvent être extraites de l'appel de fonction intercepté, comme illustré dans la figure 3.2, c'est-à-dire : l'identifiant du fichier, le décalage, la taille des E/S et le temps d'inter-arrivée (par rapport à la fonction d'E/S précédente). Pour associer une ligne de paramètres à un nœud de graphe, nous avons créé une **table de hachage d'E/S** dont les clés sont des identifiants de pile d'appels et dont les valeurs sont les derniers paramètres d'E/S vus pour ces identifiants de pile d'appels. Cette table est mise à jour après chaque E/S.

Cette table de paramètres d'E/S repose sur l'hypothèse que les E/S avec la même pile d'appels auront souvent des paramètres d'E/S identiques. Cette hypothèse a été formulée pour la première fois dans Omnisc'IO [27, 28].

Granularité de modélisation et algorithme de mise à jour du modèle

Dans les approches "boîte-grise" de modélisation des E/S de l'état de l'art étudiées, un seul modèle est créé par processus. La raison de ce choix, outre la simplicité technique, est que cela permet d'identifier des motifs dans la suite des E/S de l'application. Cependant, comme introduit précédemment, nous avons proposé deux granularités de modélisation supplémentaires afin de pouvoir prédire sélectivement les prochaines E/S d'un fichier cible (par opposition à l'état de l'art qui prédit la prochaine E/S d'un processus). L'algorithme et la structure de données présentés précédemment dans cette section sont applicables aux trois niveaux de granularité, avec les différences suivantes dans le nombre de graphes, dans le tableau des paramètres, et des structures de données associées :

- un modèle **par processus** : il y a un seul graphe et un tableau de paramètres par processus. Aucune structure de données supplémentaire n'est nécessaire.
- un modèle **par ouverture** de fichiers : un modèle est créé par descripteur de fichier. Il y a une table de hachage associant un graphe et un tableau de paramètres à chaque descripteur de fichiers unique en un temps quasi constant. Deux descripteurs de fichiers différents ne peuvent pas partager un graphe ou un tableau de paramètres.
- un modèle **par identifiant de pile d'appels d'ouverture** de fichiers : il y a une table de hachage associant un graphe et un tableau de paramètres à chaque

descripteur de fichiers en un temps quasi constant. Plusieurs descripteurs de fichiers peuvent partager le même graphe. De plus, il y a une autre table de hachage associant de manière similaire un graphe et un tableau de paramètres à chaque identifiant de pile d'appels d'ouverture en un temps quasi constant. Deux piles d'appels d'ouverture différentes ne peuvent pas partager un graphe ni un tableau de paramètres.

En dehors de ces détails de mise en œuvre spécifiques aux trois granularités de modélisation, l'algorithme GrIOt tel qu'introduit plus tôt dans l'Alg. 1 est le même. Nous en détaillons ici les étapes de modélisation, à savoir la mise à jour de la pile de contexte, celle du graphe, puis celle du tableau des paramètres :

Mise à jour de la pile de contexte Lorsqu'une E/S est interceptée et que sa pile d'appels est traitée, une condition préalable à la création d'un nœud dans le graphe est la mise à jour du contexte, c'est-à-dire de la liste des dernières piles d'appels (ligne 5 de l'Alg. 1). Cette liste est dans notre cas une pile, c'est-à-dire un type de liste chaînée. Cette liste chaînée possède une taille égale à la taille du contexte.

Mise à jour du graphe Après la mise à jour de la pile de contexte, un nœud de graphe identifié par ce contexte est créé s'il n'existe pas déjà (voir ligne 6 de l'Alg. 1). Ensuite, si une arête entrante n'existe pas entre ce nœud et le nœud précédent, elle est ajoutée en conséquence. Afin de vérifier si un nœud existe déjà dans le graphe en un temps presque constant, une table de hachage est utilisée pour stocker tous les nœuds du graphe. De plus, afin de pouvoir vérifier si un nœud a une arête vers un autre nœud, chaque nœud contient dans un tableau ses arêtes sortantes. Bien que cette dernière opération nécessite de parcourir ce tableau ce qui pourrait en affecter la performance, un tableau est plus efficace en termes de mémoire et souvent même plus rapide qu'une table de hachage car la plupart des nœuds ont moins d'une dizaine d'arêtes sortantes.

Mise à jour des métadonnées des nœuds du graphe Après la mise à jour du graphe, les données de récence et de fréquence du nœud correspondant au contexte courant sont mises à jour. Leur ajout n'est pas trivial. Une mise en œuvre naïve ajouterait ces valeurs aux arêtes du graphe. Cependant, lors de l'utilisation de la granularité de modèle « par identifiant de pile d'appels d'ouverture », nous identifions un cas limite : plusieurs descripteurs de fichiers peuvent partager un graphe en même temps. Pour que la récence et

la fréquence soient utilisées efficacement, la localité est nécessaire. Avoir plusieurs sessions de fichiers mettant à jour les mêmes métadonnées dans le même graphe briserait cette localité. C'est pourquoi les métadonnées du graphe devraient être une structure de données distincte du graphe. De cette façon, lors de l'utilisation de la granularité de modèle « par identifiant de pile d'appels d'ouverture », la granularité des métadonnées utilisée pour la prédiction est *par fichier* plutôt que « par identifiant de pile d'appels d'ouverture », respectant ainsi la localité de récence et de fréquence.

Mise à jour du tableau des paramètres Comme introduit plus haut, un tableau appelé "tableau des paramètres" est chargé de relier les piles d'appels des E/S à leurs paramètres. Cette table est mise à jour avec les paramètres d'E/S relatifs à la fonction d'E/S interceptée concernée. Afin de garantir que la recherche des paramètres d'une E/S dans le tableau à partir d'un nœud du graphe peut être réalisée en temps presque constant, ce tableau est mis en œuvre sous la forme d'une table de hachage.

3.2.3 Prédiction en ligne des E/S futures

La dernière étape de GrIOt, présentée à l'étape III de la figure 3.1, est la prédiction en ligne des E/S. Elle peut être décomposée en deux sous-étapes : prédire la prochaine pile d'appels d'E/S, et utiliser cette pile d'appels d'E/S pour prédire les paramètres de la prochaine E/S. Ces deux sous-étapes sont détaillées ci-dessous.

3.2.4 Prédiction de la pile d'appels des E/S futures

La prédiction des E/S est la dernière étape de GrIOt, après les étapes d'interception et de modélisation des E/S. À cette étape, notre algorithme de prédiction utilise le modèle et la pile de contexte mise à jour. Ce contexte peut être associé au nœud actuel dans le graphe.

Comme illustré à la ligne 10 de l'Alg. 1, l'algorithme de prédiction GrIOt vérifie d'abord s'il existe des arêtes sortantes du nœud correspondant au contexte actuel.

Si aucune n'est trouvée, ce qui se produit notamment lorsqu'un identifiant de pile d'appels est vu pour la première fois, nous faisons l'hypothèse que l'identifiant de pile d'appels se répétera séquentiellement. De fait, cela veut dire que nous prédisons un appel de fonction identique, mais avec une mise à jour des paramètres correspondant à un accès

séquentiel avec la même taille d'E/S. C'est un cas extrême, car cela ne se produira que la première fois qu'une valeur de contexte est vue.

S'il y a une unique arête sortante et comme indiqué à la ligne 13 de l'Alg. 1, GrIOt prédit que le nœud suivant sera celui vers lequel l'arête pointe.

Finalement, si il y a plus d'une arête sortante et comme indiqué à la ligne 14 de l'Alg. 1, GrIOt utilise une heuristique pour prédire le nœud suivant.

Afin de tirer parti de la localité des E/S [49], nous proposons deux stratégies très simples :

— La première stratégie consiste à suivre l'arc le plus fréquemment utilisé (*Most Frequently Used*, MFU). Pour utiliser cette heuristique, GrIOt a besoin de sauvegarder des informations liées à la fréquence d'utilisation de chaque arc sortant dans chaque nœud. Pour ce faire, une valeur de fréquence est assignée à chaque arc. Cette valeur est par défaut de 1. GrIOt incrémente cette valeur de fréquence quand l'arc est traversé, tout en décrémentant celle des autres arcs du même nœud qui n'ont, de fait, pas été traversés.
— La seconde consiste à suivre l'arc utilisé le plus récemment (*Most Recently Used*, MRU). L'arc utilisé le plus récemment est sauvegardé pour chaque nœud.

GrIOt peut prédire de multiples nœuds en avance, et par extension de multiples piles d'appels d'E/S en avance en utilisant l'une de ces deux stratégies pour traverser le graphe à partir du nœud courant.

3.2.5 Prédiction des paramètres des E/S

À l'issue de la prédiction des piles d'appels d'E/S suivantes, il s'agit d'y associer des **paramètres d'E/S tels que la taille, la position, et le temps d'inter-arrivée des E/S**, c'est-à-dire le temps séparant une E/S de l'E/S précédente. Pour ce faire, nous utilisons le tableau des paramètres introduit dans la sous-section précédente, en suivant une méthode similaire à l'état de l'art [27, 28] : le tableau des paramètres étant stocké dans une table de hachage, il est possible, en temps théoriquement constant, d'associer la valeur de contexte (d'un nœud du graphe prédit dans la sous-section précédente) aux paramètres d'E/S stockés dans le passé pour cette valeur de contexte.

Application	Description	Nœuds	Proc -essus	Volume et nombre d'E/S	# piles d'appels uniques	# transitions de piles d'appels uniques	% de piles d'appels répétées
NAMD	Dynamique moléculaire, 1.1M atomes : STMV 210M	12	12	81,5 GiB 93,3 e3 E/S	371	718	9,32 %
LAMMPS	Dynamique moléculaire, 10 k atomes : 3NIR Crambin	14	896	13,1 GiB 18,5 e3 E/S	39	52	80,18 %
Incompact3d	Solveur de Navier–Stokes	10	640	13,8 GiB 911 e3 E/S	85	110	0,28 %
LQCD	Chromodynamique quantique	16	3072	73,0 GiB 5,11 e6 E/S	319	643	97,91 %
NEMO	Simulation océanique	8	256	22,8 GiB 143 e3 E/S	229	312	32,52 %

TABLE 3.1 – Liste des applications utilisées pour l'évaluation

3.3 Évaluation expérimentale

GrIOt a été programmé en C. CMake et des macros, c'est-à-dire du code exécutable à la compilation, sont utilisés pour créer un binaire de GrIOt par granularité de modélisation. Ces binaires peuvent exécuter en ligne chacune des 3 étapes de GrIOt, c'est-à-dire l'interception des E/S d'une application, la modélisation des E/S, et la prédiction des E/S. En plus de ces 3 étapes propres à GrIOt qui sont exécutées pour chaque E/S, une étape d'export des résultats est ajoutée à la fin de l'exécution de l'application durant laquelle le modèle de prédiction produit par GrIOt et des métriques d'évaluation telles que le nombre de prédictions correctes ou leur volume sont écrites sous forme de tableur CSV.

Ces métriques d'évaluation sont détaillées dans la sous-section 3.3.2. Des applications HPC réelles sont utilisées. Elles sont détaillées dans la sous-section 3.3.1.

3.3.1 Applications instrumentées

Comme introduit dans la table 3.1, cinq applications HPC scientifiques ont été instrumentées.

LAMMPS [90] est le nom d'un outil de simulation dynamique moléculaire. LAMMPS est un sigle pour "*Large-scale Atomic/Molecular Massively Parallel Simulator*", ce qui peut être traduit littéralement en français par *Simulateur Massivement Parallèle Atomique/-*

Moléculaire à Grande Échelle. Nous utilisons un fichier d'entrée disponible publiquement [2] pour simuler une *protéine Glutamine 1WDN* avec 1024 processus sur 14 nœuds de calcul.

NAMD [77] est également un outil de simulation dynamique moléculaire. Nous utilisons un autre fichier d'entrée disponible publiquement pour exécuter NAMD sur un virus STMV [3].

Incompact3d [99] est un solveur de Navier-Stokes. Nous avons utilisé un fichier d'entrée disponible publiquement, le fichier d'entrée de référence X3D [4].

Nemo est un outil pour la simulation océanique et les études climatiques. Nous avons utilisé un fichier d'entrée non public.

LQCD est une application HPC pour la simulation de la chromodynamique quantique. Nous avons utilisé un fichier d'entrée non public.

En plus d'une brève description des applications, la Table 1 contient également des métriques liées aux piles d'appels d'E/S des applications. Ces métriques peuvent être utilisées pour évaluer la complexité des motifs de piles d'appels d'E/S, et sont utilisées dans la discussion des résultats. En effet, une faible complexité des schémas de piles d'appels d'E/S peut se traduire par (1) un pourcentage élevé de piles d'appels répétées (jusqu'à 97.91% pour LQCD) ou (2) un nombre réduit de transitions uniques entre piles d'appels comparé au nombre total de piles d'appels uniques (seulement 110 transitions uniques pour un total de 85 piles d'appels uniques dans Incompact3d).

3.3.2 Métriques d'évaluation

Nous avons identifié cinq métriques d'évaluation. Tout d'abord, la qualité de prédiction du modèle est évaluée par deux métriques. La première est la précision de prédiction, qui est définie dans cette méthodologie comme le pourcentage de prédictions de la prochaine pile d'appels qui se sont avérées correctes. La deuxième métrique d'évaluation est le pourcentage du volume des E/S dont la pile d'appels a été correctement prédite. Ces deux métriques sont complémentaires : la première permet de savoir le nombre d'E/S correctement prédites, et la deuxième leur volume.

Le coût du modèle est évalué avec les trois métriques restantes : le ralentissement lié à l'instrumentation, le ralentissement lié au modèle, et l'empreinte mémoire du modèle. Le ralentissement lié au modèle et l'empreinte mémoire du modèle sont utilisés dans les

2. https ://www.hecbiosim.ac.uk/access-hpc/benchmarks
3. https ://www.ks.uiuc.edu/Research/namd/utilities/stmv/
4. https ://github.com/xcompact3d/X3D-benchmarking

groupes d'expériences 1 et 2, et le ralentissement lié à l'instrumentation est utilisé uniquement dans le groupe d'expériences 3. L'absence de cette dernière métrique dans les deux premiers groupes d'expériences est justifiée par le fait que l'état de l'art et nos binaires de GrIOt utilisent la même technique d'instrumentation, ce qui rend la comparaison inutile.

Précision de prédiction Une prédiction de pile d'appels est considérée correcte si elle correspond exactement à la pile d'appels de la prochaine E/S. La précision de prédiction est le pourcentage de prédiction correcte. Nous utilisons également dans cette évaluation la précision de prédiction pondérée, qui correspond au pourcentage du volume d'E/S dont l'E/S a été prédite correctement.

Ralentissement Nous avons mesuré la durée d'exécution des applications listées dans la Section 3.3.1 avec et sans GrIOt. L'écart entre ces deux durées est ce que nous appelons le ralentissement total.

Ensuite, nous avons mesuré la durée de ces applications avec un binaire de GrIOt qui ne réalise que l'étape d'instrumentation des E/S telle que décrite précédemment, mais pas les étapes de modélisation et de prédiction. L'écart entre cette durée et celle observée sans le binaire de GrIOt réalisant l'instrumentation est appelé le ralentissement d'instrumentation.

Le ralentissement de modélisation et de prédiction de GrIOt peut être obtenu en soustrayant le ralentissement d'instrumentation du ralentissement total.

Afin de pouvoir produire des métriques de ralentissement similairement détaillées pour GrIOt, nous exécutons le binaire *Omnilizer* d'Omnisc'IO [73] sur une trace de chaque application. Bien qu'il eût été préférable d'exécuter Omnisc'IO directement sur les applications, comme cela est fait avec GrIOt, nous n'avons pas été en mesure de faire fonctionner le module d'instrumentation d'Omnisc'IO. Comme Omnisc'IO utilise LD_PRELOAD similairement à GrIOt pour son instrumentation des E/S, nous considérons qu'ils ont tous les deux le même ralentissement d'instrumentation.

Empreinte mémoire du modèle L'empreinte mémoire du modèle fait partie des métriques complémentaires exportées automatiquement par GrIOt à la fin de l'exécution de chaque application. Parce que GrIOt est exécuté par processus, il ne s'agit pas d'une empreinte mémoire du modèle par nœud, mais bien d'une empreinte mémoire par processus. De plus, parce que la taille du modèle peut non seulement augmenter mais aussi diminuer

pendant le fil de l'exécution d'une application, il ne s'agit pas de l'empreinte mémoire moyenne mais bien de l'empreinte mémoire maximale du modèle observée sur la totalité de l'exécution de l'application.

Pour Omnisc'IO, Omnilyzer exporte la taille du modèle sous la forme d'un nombre de lignes de grammaire. Afin d'avoir des métriques comparables, nous avons modifié l'Omnilyzer pour qu'il exporte la taille en mémoire de son modèle, plutôt qu'un nombre de lignes de grammaire.

3.3.3 Méthodologie

Trois groupes d'expériences visant à évaluer l'efficacité de GrIOt sur ces applications sont proposés. Le premier compare GrIOt et l'état de l'art. Le second compare différentes configurations de GrIOt. Le troisième étudie avec plus de détail le coût - en performance - de l'instrumentation de la pile d'appels.

GrIOt est comparé à Omnisc'IO [27, 28, 73]. Omnisc'IO est choisi pour cette évaluation car il s'agit du dernier et du plus performant modèle de prédiction des E/S utilisant la pile d'appels dont le code source était disponible [5].

Un cluster de calcul comportant 32 nœuds de calcul X440-A5 BullSequana est utilisé. Chaque nœud comprend deux CPU AMD EPYC Rome 7282 (16 cœurs, 2.8GHz-120W) et 128GiB de RAM DDR4-3200MHz par noed.

3.4 Résultats expérimentaux et discussion

Trois groupes d'expériences ont été réalisés. Le premier compare GrIOt avec l'état de l'art. Le deuxième compare différentes configurations de GrIOt. Le troisième évalue en détail le coût d'instrumentation, et notamment le coût de l'instrumentation de la pile d'appels.

3.4.1 Comparaison de GrIOt à l'état de l'art

Le premier groupe d'expérience a été réalisé pour comparer notre modèle de prédiction avec l'état de l'art, Omnisc'IO.

5. https ://github.com/mdorier/omniscio

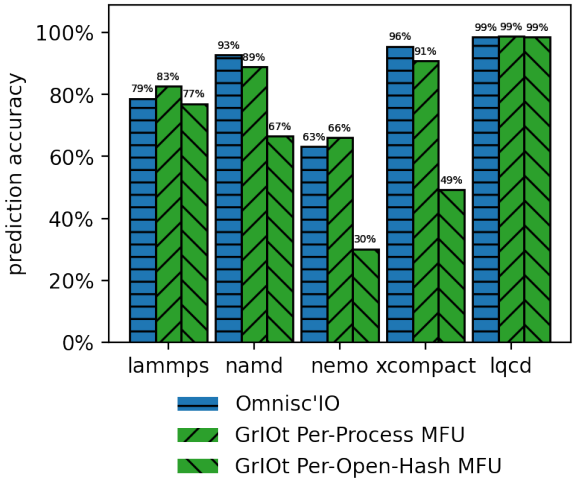


FIGURE 3.6 – Comparaison de la précision de prédiction de GrIOt et d'Omnisc'IO

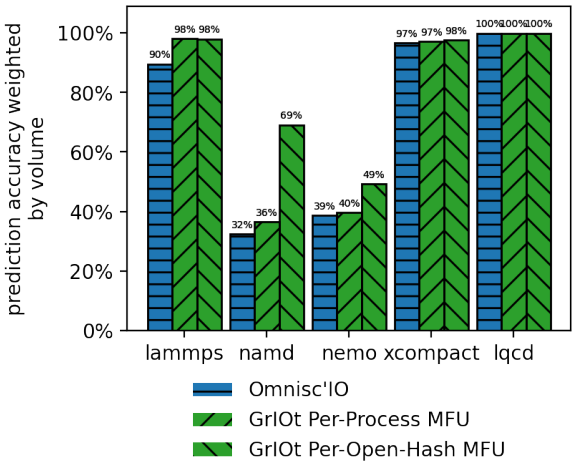


FIGURE 3.7 – Comparaison de la précision de prédiction de GrIOt et d'Omnisc'IO, pondérée par le volume d'E/S

Précision de prédiction

Dans la figure 3.6, la précision des prédictions de GrIOt et d'Omnisc'IO sont comparées. Deux configurations de GrIOt ont été utilisées, et sont les mêmes pour les quatre parties suivantes. La première utilise la granularité de modélisation « par processus » (notée "GrIOt Per-Process"), tandis que la seconde utilise la granularité de modélisation « par identifiant de pile d'appels d'ouverture ». Les deux utilisent MFU comme heuristique de prédiction, avec une taille de contexte de 2. On peut observer que GrIOt avec cette configuration de base a une précision de prédiction comparable à celle de l'état de l'art. GrIOt avec une granularité « par processus » a des performances très similaires à Omnisc'IO : il fait légèrement mieux sur 3 charges de travail et est légèrement en retard pour les trois autres. Cela valide notre approche : **la modélisation à l'aide de graphe de piles d'appels d'E/S de GrIOt offre une précision similaire voire supérieure à Omnisc'IO, sans demander un encodage sans perte de la séquence d'E/S comme le fait Omnisc'IO.** Dans le même temps, on voit que GrIOt avec une granularité de modélisation « par identifiant de pile d'appels d'ouverture » a une précision plus faible. Cela peut s'expliquer par le fait que disposer d'informations concernant tous les fichiers en même temps (modélisation par processus) peut être utile pour la prédiction, en particulier pour prédire les fichiers journaux qui représentent beaucoup de petites E/S. Cette information est disponible pour GrIOt avec la granularité « par processus », mais

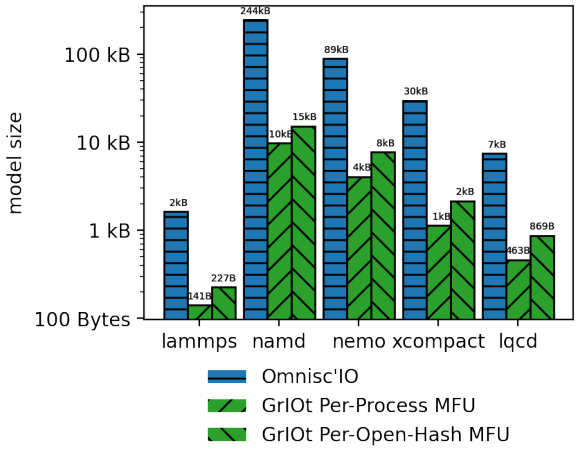


FIGURE 3.8 – Comparaison du ralentissement causé par les phases de modélisation et de prédiction de GrIOt et d'Omnisc'IO.

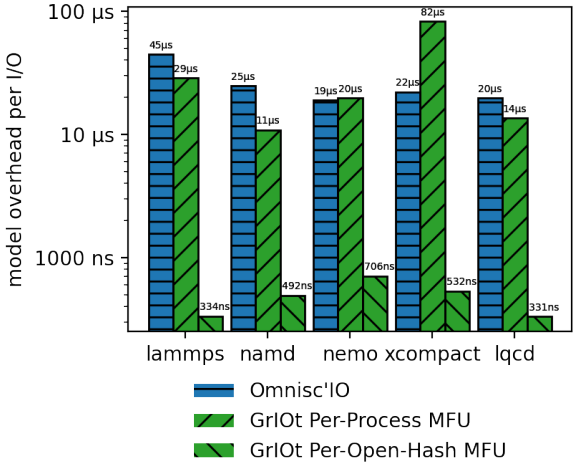


FIGURE 3.9 – Comparaison de l'empreinte mémoire de GrIOt et d'Omnisc'IO

elle n'est pas disponible avec la granularité « par identifiant de pile d'appels d'ouverture » puisque le modèle est construit par fichier.

Précision de la prédiction, pondérée par le volume

Dans la figure 3.7, la précision de la prédiction pondérée est utilisée comme métrique de précision alternative pour comparer GrIOt et OmniscIO. Dans cette figure, plutôt que de comparer le nombre d'E/S aux piles d'appels correctement prédites, nous évaluons ici le volume d'E/S correctement prédit. Nous pouvons voir des tendances différentes comparées à la figure 3.6. Cela s'explique par le fait que bien qu'Omnisc'IO et GrIOt prédisent correctement un certain nombre de piles d'appels d'E/S, le volume de ces E/S n'est pas nécessairement élevé. Plus particulièrement, l'écriture de fichiers journaux se fait souvent par de nombreuses très petites E/S. En raison de leur nombre élevé, ces E/S peuvent avoir un impact significatif sur la métrique de précision malgré leur volume d'E/S relativement faible. **Ainsi, nous pouvons voir que GrIOt avec la granularité de modélisation « par identifiant de pile d'appels d'ouverture » prédit correctement la pile d'appels d'un plus grand volume d'E/S qu'Omnisc'IO et que GrIOt avec la granularité de modélisation « par processus ».** Cela s'explique par le fait qu'en créant un modèle au niveau du fichier avec la granularité « par identifiant de pile d'appels d'ouverture », plusieurs modèles simples et faciles à prédire sont créés plutôt qu'un unique

modèle monolithique par processus avec des prédictions plus complexes. Ainsi, il s'avère qu'un plus grand volume d'E/S est prédit correctement.

Ralentissement causé par les phases de modélisation et de prédiction

Dans la figure 3.9, les ralentissements de prédiction et de modélisation par E/S de notre modèle et d'Omnisc'IO sont comparés. Cette métrique ne comprend pas le coût d'interception, qui est le même pour GrIOt et Omnisc'IO. **On peut observer que le modèle « par identifiant de pile d'appels d'ouverture » a un ralentissement par E/S beaucoup plus faible qu'Omnisc'IO, avec une réduction allant jusqu'à 120x sur LAMMPS.** Contrairement à un modèle de grammaire, qui nécessite un algorithme complexe pour la prédiction, faire une prédiction dans un graphe, étant donnés un nœud et une heuristique est simple et a une faible complexité. Dans le même temps, on voit que **GrIOt avec la granularité « par processus » souffre d'une performance légèrement moins bonne qu'avec la granularité « par identifiant de pile d'appels d'ouverture », mais généralement meilleure qu'Omnisc'IO.** Cela peut s'expliquer par le grand graphe unique qui doit être parcouru (plutôt que plusieurs petits graphes pour l'option par pile d'appels d'ouverture), et une implémentation de table de hachage médiocre. Nous pensons que cette surcharge plus élevée est due à des collisions de hachage dans les tables de GrIOt réduisant les performances. Ainsi, de meilleurs résultats pourraient être obtenus en augmentant la taille de la table de hachage, ou en essayant une fonction de hachage alternative.

Empreinte mémoire du modèle

Dans la figure 3.8, les empreintes mémoire de GrIOt et d'Omnisc'IO sont comparées. **L'empreinte mémoire de GrIOt est systématiquement inférieure à la taille du modèle d'Omnisc'IO, jusqu'à 100x inférieure sur LQCD.** Cela s'explique par le fait que contrairement à Omnisc'IO, GrIOt n'essaie pas d'encoder la séquence de la pile d'appels d'E/S sans perte. Nous notons que la granularité par pile d'appels d'ouverture a une taille de modèle légèrement supérieure à la granularité « par processus » : cela peut s'expliquer par le fait qu'elle maintient plusieurs petits graphes plutôt qu'un grand graphe unique, ce qui est plus coûteux en raison de la nécessité de structures de données supplémentaires.

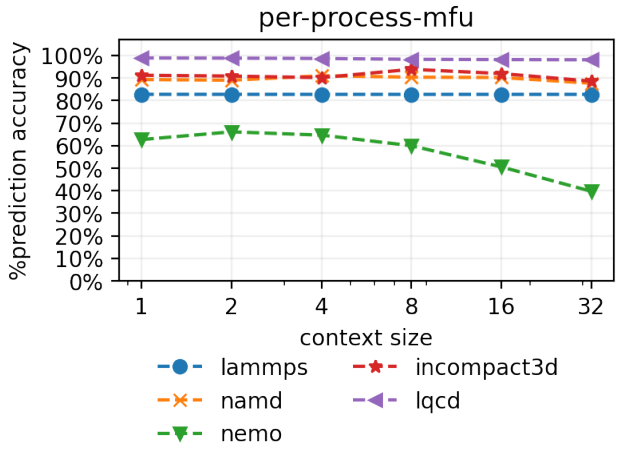


FIGURE 3.10 – Effet de la taille du contexte sur la précision de prédiction de GrIOt

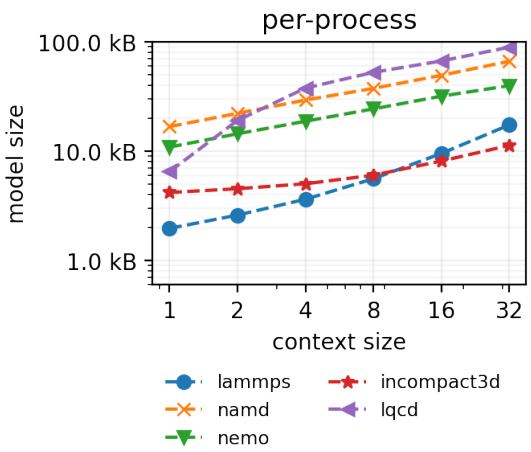


FIGURE 3.11 – Effect of context size on model size

3.4.2 Comparaison des différentes configurations de GrIOt

Dans le groupe d'expériences précédent, nous avons comparé GrIOt à l'état de l'art. Dans ce deuxième groupe d'expériences, nous avons évalué l'impact des paramètres de GrIOt sur la précision de prédiction, le ralentissement causé par les phases de modélisation et de prédiction, et l'empreinte mémoire du modèle. Ces paramètres sont la taille de contexte, la granularité de modélisation, et le choix de l'heuristique de prédiction.

Effet de la taille du contexte : La Fig. 3.10 montre la précision de prédiction de GrIOt en fonction de la taille du contexte. La granularité du modèle et l'heuristique de prédiction sont fixées à notre configuration de base, par processus avec MFU. On peut observer que pour certaines applications, en particulier NEMO, une taille de contexte plus élevée (par rapport à une taille de contexte de 1) a un effet positif sur la précision de la prédiction pour des valeurs faibles, mais un effet négatif pour des valeurs très élevées (supérieures à 8). Le premier point s'explique par le fait qu'une taille de contexte plus grande permet à notre modèle d'identifier les E/S de manière plus efficace, comme illustré précédemment dans les Fig. 3.4 et 3.5. Le second point peut être interprété par l'impact négatif de l'augmentation de la taille du contexte sur la vitesse d'apprentissage : une taille de contexte plus élevée signifie plus de nœuds dans le graphe, ce qui allonge le temps nécessaire à la construction et à la mise à jour du modèle complet de l'application. Le fait que la performance sur d'autres applications, en particulier LAMMPS, LQCD et Incompact3d, ne soit pas affectée par la taille du contexte suggère qu'elles présentent

	LAMMPS	NAMD	NEMO	Incompact3d	LQCD
Per-Process	141 bytes	9.8 kB	4.1 kB	1.1 kB	463 bytes
Per-Open-Hash	227 bytes	15.2 kB	7.8 kB	2.2 kB	870 bytes
Per-Open	281 bytes	2.4 kB	11.9 kB	383 bytes	283 bytes

TABLE 3.2 – Effet de la granularité de modélisation de GrIOt sur la taille du modèle

des schémas de pile d'appels E/S plus simples. Cela est confirmé dans le Tableau 1, où l'on peut observer que LAMMPS et LQCD possèdent effectivement le taux le plus élevé de répétition directe des piles d'appels. En outre, on constate qu'Incompact3d présente un nombre de transitions uniques de piles d'appels très proche du nombre total de piles d'appels uniques (110 contre 85), ce qui suggère également que l'application possède des schémas de pile d'appels E/S très simples. Toutefois, ce n'est pas le cas de toutes les applications HPC. En effet, NEMO et NAMD n'ont ni un faible nombre de piles d'appels uniques (229 pour NEMO, 371 pour NAMD), ni un pourcentage particulièrement élevé de répétitions de piles d'appels (32.52% pour NEMO, 9.32% pour NAMD). On note que NAMD utilise Charm++, un framework de programmation parallèle adaptatif, ce qui rend les piles d'appels E/S plus complexes et moins déterministes. De plus, au-delà de NEMO et NAMD, la littérature récente souligne que les charges de travail liées à l'intelligence artificielle sont de plus en plus fréquentes sur les plateformes HPC, et qu'elles présentent des schémas d'E/S encore différents [76].

Dans la Fig. 3.11, la taille du modèle est représentée en fonction de la taille du contexte. On observe qu'une taille de contexte plus élevée entraîne une augmentation de la taille du modèle. C'est la raison pour laquelle nous avons choisi une taille de contexte de 2 comme valeur de référence dans cette évaluation, afin de maximiser la précision de prédiction tout en maintenant la taille du modèle à un niveau raisonnable.

Effet de la granularité de modélisation : Dans la table 3.2, la taille du modèle est représentée en fonction de la granularité de modélisation pour les cinq applications étudiées. Toutes les granularités entraînent des tailles de modèle inférieures à 20 kB. La granularité "par identifiant de pile d'appels d'ouverture" entraîne systématiquement une taille de modèle plus élevée que la granularité "par processus". Cela s'explique par le fait qu'au cours de l'exécution de l'application, GrIOt génère alors plusieurs modèles au lieu d'un seul modèle par processus. La granularité de modélisation "par ouverture", quant à elle, présente soit la taille de modèle la plus élevée, soit la plus faible. En effet, elle crée un modèle par descripteur de fichiers ouvert, puis supprime ces modèles dès que les fichiers

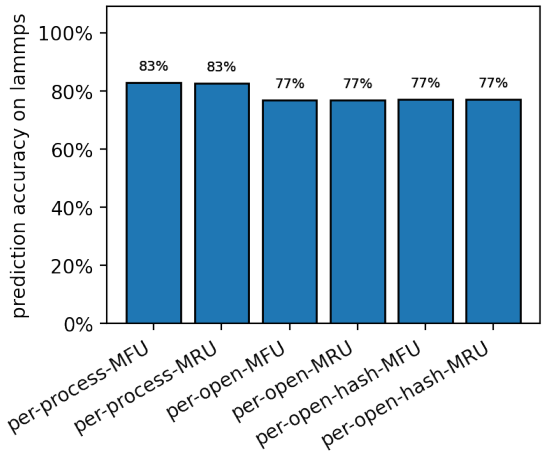


FIGURE 3.12 – Effet de la granularité de modélisation et de l'heuristique de précision sur la précision de prédiction pour LAMMPS

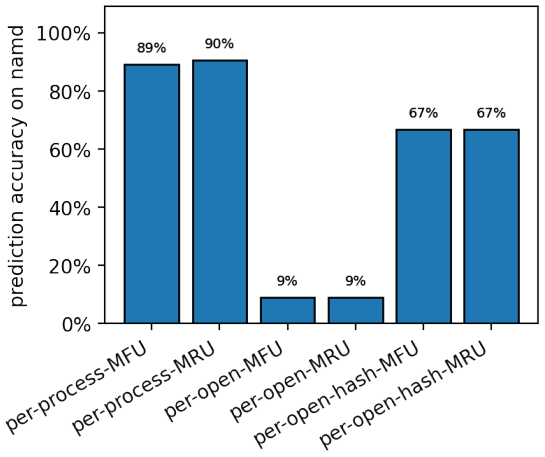


FIGURE 3.13 – Effet de la granularité de modélisation et de l'heuristique de précision sur la précision de prédiction pour NAMD

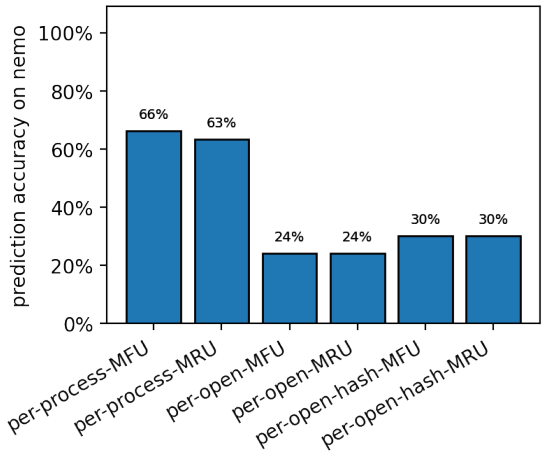


FIGURE 3.14 – Effet de la granularité de modélisation et de l'heuristique de précision sur la précision de prédiction pour NEMO

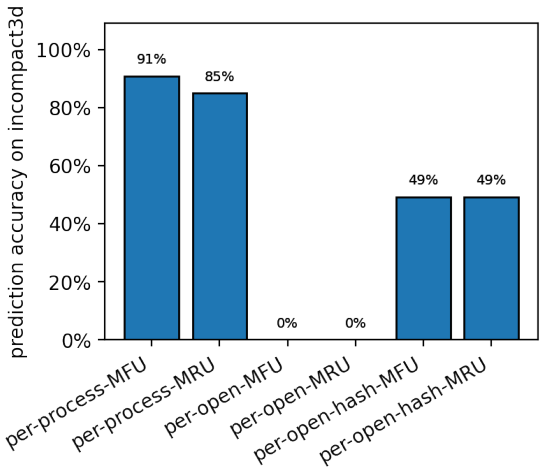


FIGURE 3.15 – Effet de la granularité de modélisation et de l'heuristique de précision sur la précision de prédiction pour INCOMPACT3D

sont fermés. Ainsi, la taille maximale observée du modèle est élevée si l'application ouvre de nombreux fichiers simultanément, et très faible si les fichiers sont ouverts de manière séquentielle.

Dans les figures 3.12 à 3.16, la précision du modèle est affichée en fonction de sa granularité pour les différentes applications. On observe que la granularité « par proces-

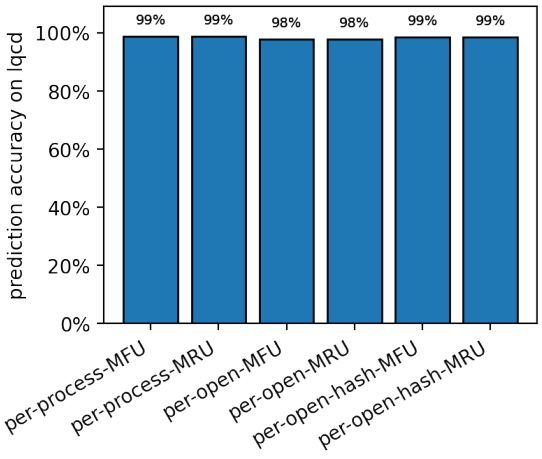


FIGURE 3.16 – Effet de la granularité de modélisation et de l'heuristique de précision sur la précision de prédiction pour LQCD

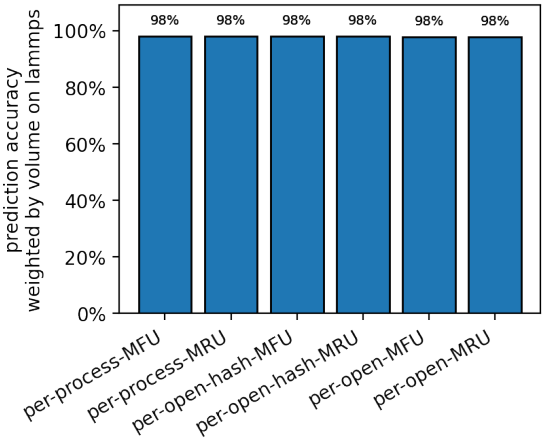


FIGURE 3.17 – Effet de la granularité de modélisation et de l'heuristique de précision sur la précision de prédiction pondérée pour LAMMPS

sus » offre la meilleure précision avec cette métrique d'évaluation. C'est pourquoi cette approche a été choisie comme configuration de référence. En revanche, dans les figures 3.17 à 3.21, où la métrique utilisée est la précision de prédiction pondérée, la conclusion diffère : la granularité « par identifiant de pile d'appels d'ouverture » obtient la meilleure précision pondérée. Cela s'explique par le fait que cette granularité permet de capturer des métadonnées de récence et de fréquence des fichiers, tout en réutilisant les modèles pour des fichiers ouverts avec des piles d'appels similaires, ce qui améliore la qualité des prédictions. La granularité « par ouverture de fichiers » offre également ces avantages, mais elle affiche de moins bonnes performances dans la plupart des applications. En effet, certaines applications effectuent moins d'une dizaine d'opérations d'E/S entre chaque ouverture et fermeture de fichier. Dans ces cas, une modélisation trop fine, sans réutilisation des modèles, devient un désavantage, car les modèles sont supprimés avant d'avoir appris le comportement des E/S sur ces fichiers.

Dans les figures 3.22 à 3.26, le ralentissement causé par les phases de modélisation et de prédiction par opération d'E/S est présenté en fonction de la granularité de modélisation pour les cinq applications. La granularité « par processus » entraîne le ralentissement le plus élevé dans tous les cas, atteignant jusqu'à 82 µs pour Xcompact3d. Toutefois, ce ralentissement ne représente que 3,5 % du temps total d'une opération d'E/S, car la durée moyenne des E/S dans Xcompact3d est relativement élevée. En comparaison, pour

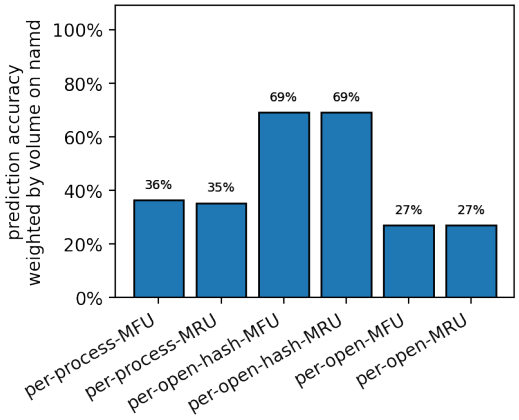


FIGURE 3.18 – Effet de la granularité de modélisation et de l'heuristique de précision sur la précision de prédiction pondérée pour NAMD

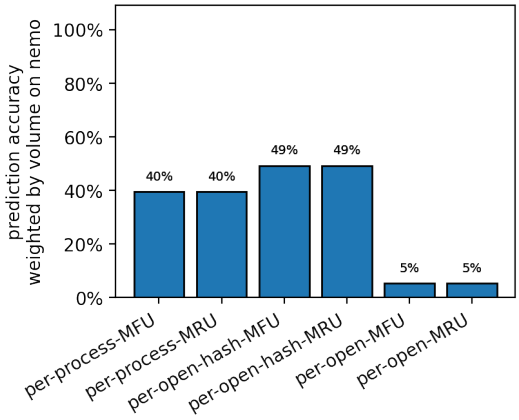


FIGURE 3.19 – Effet de la granularité de modélisation et de l'heuristique de précision sur la précision de prédiction pondérée pour NEMO

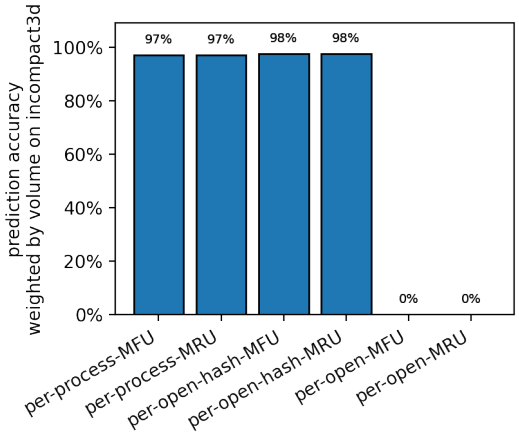


FIGURE 3.20 – Effet de la granularité de modélisation et de l'heuristique de précision sur la précision de prédiction pondérée pour INCOMPACT3D

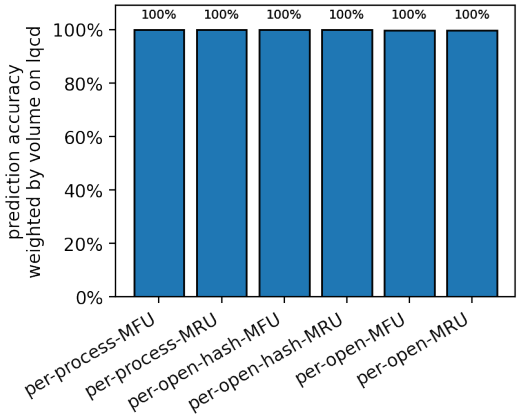


FIGURE 3.21 – Effet de la granularité de modélisation et de l'heuristique de précision sur la précision de prédiction pondérée pour LQCD

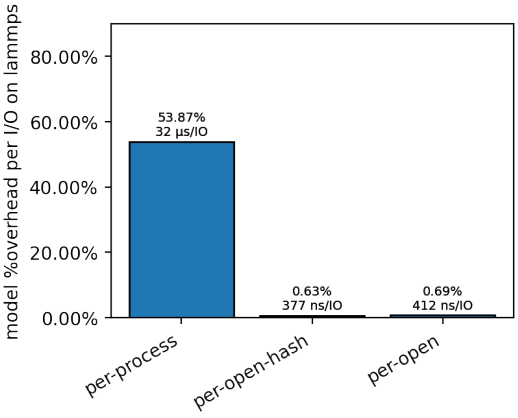


FIGURE 3.22 – Effet de la granularité de modélisation sur le ralentissement par E/S, pour LAMMPS

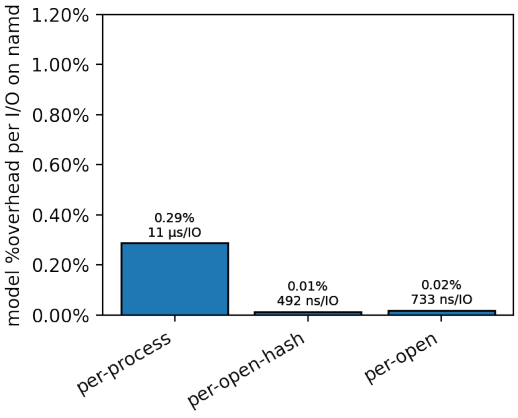


FIGURE 3.23 – Effet de la granularité de modélisation sur le ralentissement par E/S, pour NAMD

LAMMPS, le ralentissement lié à la modélisation au niveau du processus est de seulement 32 µs, mais cela représente 54 % du temps total d'une opération d'E/S. Cette forte proportion s'explique par la durée moyenne des E/S dans LAMMPS, qui est beaucoup plus faible. En comparaison, les deux autres granularités de modélisation présentent un ralentissement nettement inférieur à celui du niveau processus, avec un impact inférieur à 0,1 % pour les cinq applications. De plus, combinant le ralentissement par opération d'E/S présenté dans les Figures 3.22 à 3.26 avec le nombre d'opérations d'E/S indiqué dans le Tableau 1, il est possible pour chaque application d'estimer le surcoût total en E/S induit pour chaque application. Il est à noter que le surcoût maximal observé est de 75 secondes pour Incompact3d. Étant donné que l'exécution de chacune des applications évaluées varie de 2 à 6 heures (dans le cas d'une configuration avec un point de sauvegarde par heure), ce surcoût peut être considéré comme négligeable. En outre, ce surcoût est réparti sur les 10 nœuds de calcul et les 640 processus parallèles utilisés par Incompact3d, ce qui atténue encore davantage son impact réel sur la durée de l'application.

Effet de l'heuristique de prédiction : Dans les figures 3.12 à 3.21, on observe que l'heuristique de prédiction influence la précision des prédictions. L'heuristique « MFU » a tendance à offrir une précision légèrement supérieure, quelle que soit la métrique de précision utilisée. L'impact du choix de l'heuristique sur la taille du modèle n'a pas été évalué, car les deux heuristiques sont prises en charge par les trois implémentations du modèle. Cependant, l'heuristique « MFU » nécessite légèrement plus de mémoire que l'heuristique « MRU », car elle doit stocker une valeur de fréquence par arête, tandis que

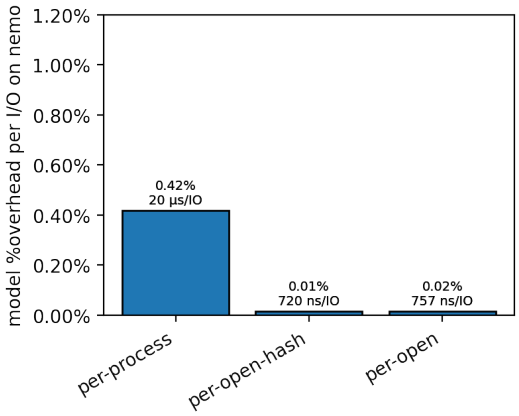


FIGURE 3.24 – Effet de la granularité de modélisation sur le ralentissement par E/S, pour NEMO

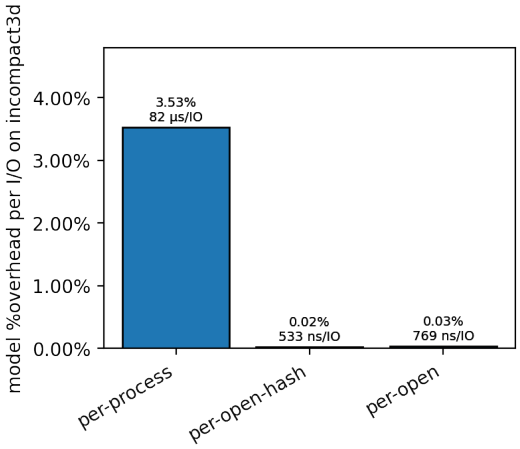


FIGURE 3.25 – Effet de la granularité de modélisation sur le ralentissement par E/S, pour INCOMPACT3D

l'heuristique « MRU » ne stocke qu'une seule valeur par nœud. Or, le nombre d'arêtes est le plus souvent supérieur ou égal au nombre de nœuds en raison de la répétition des piles d'appels.

3.4.3 Caractérisation de l'instrumentation de la pile d'appels

Comme détaillé précédemment, l'instrumentation de la pile d'appels peut être réalisée à l'aide de la fonction *backtrace* ou via *libunwind*. Dans les deux cas, la profondeur d'instrumentation de la pile d'appels est un paramètre configurable.

Effet de la profondeur d'instrumentation de la pile d'appels : Dans la figure 3.27, le nombre total de piles d'appels distinctes est représenté en fonction de la profondeur d'instrumentation pour les cinq applications étudiées. Cette figure montre qu'une profondeur plus faible ne permet pas de différencier toutes les piles d'appels. Les figures 3.28 et 3.29 illustrent la surcharge induite par l'instrumentation en fonction de la profondeur d'instrumentation. Il peut être observé qu'augmenter la profondeur d'instrumentation de la pile d'appels entraîne une augmentation du ralentissement lié à l'instrumentation. De plus, on observe que cette augmentation du ralentissement suit de près l'évolution du nombre total de piles d'appels différenciables. Par conséquent, il semble que la seule manière de réduire cette surcharge soit de limiter le nombre de piles différenciables, ce qui pourrait nuire à l'utilité des prédictions. C'est pourquoi nous avons choisi d'adopter une profondeur d'instrumentation élevée de 32 afin de garantir la différenciation des piles

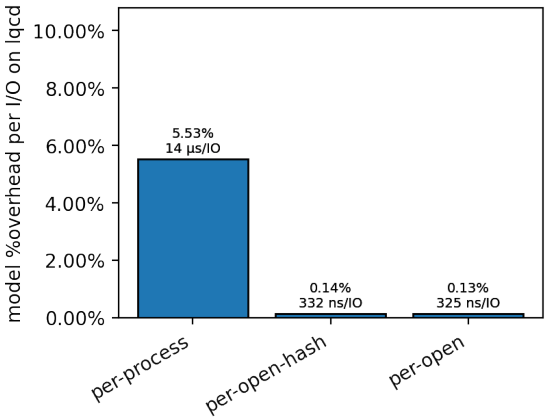


FIGURE 3.26 – Effet de la granularité de modélisation sur le ralentissement par E/S, pour LQCD

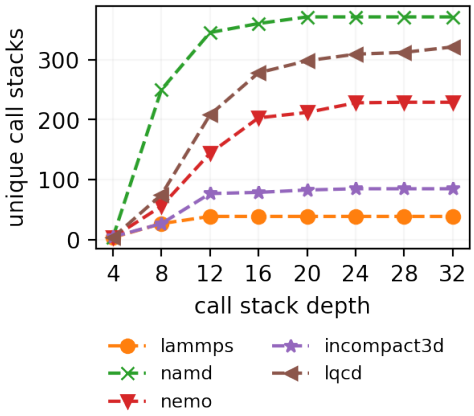


FIGURE 3.27 – Effet de la profondeur maximale d'instrumentation de la pile d'appel sur le taux de différenciation des piles d'appels

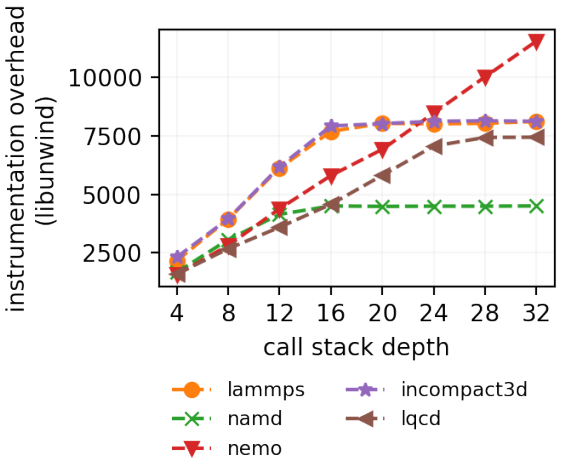


FIGURE 3.28 – Effet de la profondeur maximale d'instrumentation de la pile d'appel sur le coût d'instrumentation avec *libunwind*

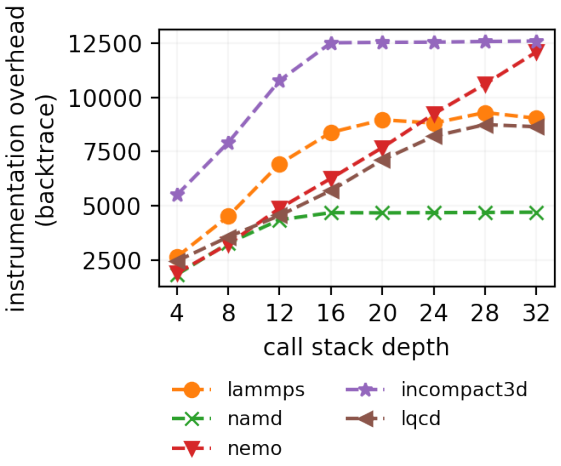


FIGURE 3.29 – Effet de la profondeur maximale d'instrumentation de la pile d'appel sur le coût d'instrumentation avec *backtrace*

d'appels.

Comparaison entre *backtrace* et *libunwind* : Dans les figures 3.28 et 3.29, nous comparons la surcharge d'instrumentation de *backtrace* et *libunwind*. Nous observons que, pour toutes les applications sauf une, le surcoût associé à *libunwind* est similaire mais légèrement inférieur à celui de *backtrace*, avec des gains allant jusqu'à 10%. Pour Incompact3d en revanche, *libunwind* présente un surcoût jusqu'à 50% inférieur à celui de *backtrace* (2.5 µs contre 5 µs pour une profondeur de pile d'appels de 4, 8 µs contre 12.5 µs pour une profondeur de 32). Nous formulons l'hypothèse que cet écart de performance s'explique par une meilleure prise en charge des trames DWARF par *libunwind* en l'absence de pointeurs de trame [5]. Enfin, il est important de noter que *libunwind* est une bibliothèque externe, liée dynamiquement. Son intégration introduit une complexité supplémentaire pour les administrateurs système en termes d'installation et de maintenance.

3.5 Conclusion

Dans ce chapitre, nous avons présenté **GrIOt**, une nouvelle stratégie de modélisation et de prédiction des entrées/sorties (E/S) pour les applications HPC. Cette approche se distingue des modèles à base de grammaires, comme Omnisc'IO, en utilisant des graphes de piles d'appels et un algorithme de prédiction plus simple basé sur des heuristiques telles que MRU ou MFU. GrIOt vise à prédire les motifs d'E/S aussi bien pour des applications à comportement déterministe que non déterministe, tout en réduisant le coût de modélisation et de prédiction.

L'évaluation expérimentale menée sur cinq applications HPC a permis de dégager plusieurs résultats. Tout d'abord, **la précision de prédiction de GrIOt est comparable, voire supérieure à celle de l'état de l'art Omnisc'IO**. En particulier, lorsqu'on considère la précision pondérée par le volume d'E/S, GrIOt parvient à prédire correctement un plus grand volume d'opérations d'E/S que les approches existantes. Cela démontre que la modélisation par graphes peut être pertinente, et qu'elle permet de capturer efficacement la structure des schémas d'E/S, même dans le cas d'applications présentant une forte variabilité ou des comportements d'E/S irréguliers, comme NAMD du fait de Charm++.

Ensuite, les résultats ont mis en évidence **un avantage significatif en termes de performances et d'efficacité mémoire**. Le coût de modélisation et de prédiction de GrIOt est jusqu'à 120 fois inférieur à celui d'Omnisc'IO (334 ns vs. 45 µs avec LAMMPS), tandis que son empreinte mémoire est 10x à 15x plus faible (463 B vs. 7 kB avec LQCD).

Ces résultats confirment que la représentation par graphes permet non seulement d'améliorer la rapidité des prédictions, mais aussi de réduire la consommation de ressources, rendant GrIOt plus adapté à un déploiement sur des systèmes de production.

L'analyse des différents paramètres du modèle a également permis d'identifier plusieurs leviers d'optimisation. **La granularité de modélisation** joue un rôle central : une granularité « par processus » maximise la précision simple, tandis qu'une granularité « par identifiant de pile d'appels d'ouverture » optimise la précision pondérée par le volume d'E/S et réduit le coût de modélisation. **La taille du contexte** influence la complexité et la capacité d'apprentissage du modèle : un contexte trop court réduit la différenciation des motifs d'E/S, tandis qu'un contexte trop long augmente la taille du modèle et son temps d'apprentissage, donc sa réactivité. Enfin, **le choix de l'heuristique de prédiction** (MFU ou MRU) permet d'adapter le comportement du modèle selon les caractéristiques de l'application. Tous ces paramètres font que le compromis entre précision, empreinte mémoire et ralentissement de l'application peut donc être ajusté en fonction des besoins de l'utilisateur et des caractéristiques de l'application.

Enfin, la caractérisation détaillée du coût d'instrumentation des piles d'appels a permis de mieux comprendre les limites pratiques de la méthode. Notamment, l'étude de la profondeur d'instrumentation et la comparaison entre les bibliothèques *backtrace* et *libunwind* ont montré que la différenciation des piles d'appels était fortement liée au coût d'instrumentation, ce qui ouvre la voie à des optimisations futures.

En résumé, les résultats démontrent que **GrIOt est une alternative robuste, performante et modulable** pour la modélisation et la prédiction d'E/S dans les environnements HPC. Grâce à son approche basée sur les graphes, il offre une meilleure compréhension des comportements d'E/S et un compromis intéressant entre précision, coût d'exécution et empreinte mémoire. Son adaptabilité lui permet d'être utilisé sur une large gamme d'applications, qu'elles soient déterministes ou non, et ouvre la voie à des optimisations futures des E/S.

Dans le chapitre suivant, nous nous penchons sur **l'utilisation des indices d'usage de données**. Il serait souhaitable de fournir aux PFS, grace à ces indices d'usage de données, les informations issues de la modélisation prédictive pour influencer le placement des fichiers et optimiser les performances globales et la consommation d'énergie. Cependant, l'effet réel de ces indices reste encore mal défini. Le chapitre suivant se concentre donc sur l'analyse expérimentale de leur impact sur le placement et les performances des PFS.

Chapitre 4

Contribution 2 : Étude de l'effet des indices d'usage de données sur des systèmes de fichiers parallèles

4.1 Motivation

Un placement efficace des données est nécessaire à la performance des applications HPC lorsqu'elles réalisent des E/S. Pour ce faire, et comme détaillé dans le premier chapitre de ce manuscrit, les PFS tels que Lustre [84] et GPFS [83] sont conçus pour gérer d'importants volumes de données. Cependant, en raison de la variabilité des motifs d'E/S des applications, l'optimisation de la performance de ces PFS reste un défi tant pour les utilisateurs que pour les développeurs [80].

Pour répondre à cette problématique, les systèmes de fichiers proposent des mécanismes d'optimisation sous la forme d'indices, associés à des fonctions telles que POSIX fadvise [79] et Lustre ladvise [57]. Leur utilisation permet aux développeurs applicatifs d'indiquer la manière dont ils comptent accéder aux données.

POSIX fadvise est un appel système qui permet aux applications de fournir des indices quant à l'utilisation prévue des données d'un fichier, ce qui permet d'optimiser les stratégies de mise en cache et de préchargement. Lustre ladvise étend le concept de POSIX fadvise, permettant aux applications d'un nœud de calcul de transmettre des indices directement aux nœuds de données Lustre. Ces deux fonctions et les indices qu'elles supportent sont détaillés dans le chapitre 2.

POSIX fadvise [79] indique que la fonction peut être utilisée pour transmettre des indices, entre autres au PFS, et que ces indices peuvent être utilisés pour optimiser la gestion des données, sans garanties. Similairement, la documentation de Lustre ladvise [57]

indique que les indices sont transmis aux serveurs de données de Lustre sans garantir la manière ni le moment de leur prise en compte. Ainsi, des actions peuvent être déclenchées à la réception des indices, ou pas, en fonction du type d'indice formulé ainsi que des décisions prises en temps réel par le PFS.

Dans la littérature, lorsque ces indices sont utilisés, les développeurs ont généralement des attentes vis-à-vis du PFS : par exemple, que les données marquées avec l'indice `WILL_NEED` soient préchargées par le système de fichiers [35], tandis que celles marquées avec `DONT_NEED` soient évacuées du cache [54]. Dans la littérature, ces indices sont principalement utilisés pour le préchargement [35] et l'éviction de données [53]. Les indices sur les motifs d'E/S tels que `RANDOM`, `SEQUENTIAL` (et plus rarement `NOREUSE`), quant à eux, peuvent également être utilisés avec comme intention la désactivation de certaines optimisations comme la lecture anticipée [35], ou pour éviter que les fichiers journaux en cours d'écriture ne restent inutilement en cache [54].

Ainsi, bien que les développeurs d'applications et les chercheurs aient des attentes claires quant au comportement d'un système de fichiers lorsqu'il reçoit des indices, la manière dont ces conseils sont réellement pris en compte n'est pas spécifiée dans la documentation de POSIX fadvise et de Lustre ladvise, ni dans celles des PFS. Cela peut conduire certains utilisateurs à ne pas exploiter ces mécanismes, au risque d'une dégradation de la performance des E/S, tandis que d'autres investissent du temps à intégrer des recommandations qui, en réalité, sont ignorées par le système de fichiers utilisé. Dans les cas les plus extrêmes, notre étude montre que des indices couramment utilisés, comme `WILL_NEED`, peuvent être complètement ignorés et ne déclenchent aucun préchargement, contrairement aux attentes. Cette gestion du cache inattendue entraîne alors une utilisation sous-optimale des ressources, et nous observons que cela peut provoquer une diminution du débit allant jusqu'à une division par trois.

C'est pourquoi dans ce chapitre, nous proposons de clarifier l'effet des indices d'usage de données sur les systèmes de fichiers parallèles ainsi que sur la performance des E/S. Nos contributions sont triples : (1) nous proposons une méthodologie permettant d'évaluer l'efficacité de ces indices sur différents PFS ; (2) nous présentons un banc d'essai synthétique simple mettant en œuvre cette méthodologie, afin de permettre aux utilisateurs d'évaluer si ces indices peuvent réellement améliorer la performance des E/S sur leur PFS ; et (3) nous analysons les résultats obtenus à l'aide de ce banc d'essai, afin de pouvoir fournir des recommandations pratiques aux développeurs

applicatifs.

4.2 Proposition : méthodologie d'évaluation de l'effet des indices d'usage de données

Afin de pouvoir caractériser automatiquement l'effet des indices d'E/S sur les PFS, nous proposons dans cette étude quatre expériences basées sur une application synthétique afin de caractériser l'effet des différents indices sur la performance d'E/S.

Une application synthétique est utilisée parce qu'elle permet une analyse approfondie de l'effet des indices d'usage de données en faisant varier la taille des E/S, les temps d'inter-arrivée, les motifs d'E/S et les tailles de fichiers, couvrant ainsi la plupart des comportements d'E/S rencontrés en pratique [91].

La principale métrique d'évaluation que nous utilisons pour décrire la performance des E/S est le débit. Le débit est une métrique courante de performance d'E/S [59], définie comme le volume d'E/S divisé par le temps d'E/S. Pour nos expériences, qui sont réalisées à l'aide d'une application synthétique, l'instrumentation du volume et du temps d'E/S est directement intégrée dans le code source.

Expérience 1 : Caractérisation du système de stockage : La première expérience synthétique que nous présentons dans cette méthodologie vise à caractériser le système de stockage. Nous mesurons une référence de la performance des lectures lorsque les données ne sont pas mises en cache (1), mises en cache dans les nœuds de calcul (2) ou mises en cache uniquement dans les nœuds de données (3). Nous nous attendons à observer la meilleure performance lorsque les données sont mises en cache dans les nœuds de calcul, la performance la plus faible lorsqu'elles ne sont pas mises en cache, et une performance intermédiaire lorsqu'elles sont mises en cache dans les nœuds de données.

Expérience 2 : Évaluation de l'effet des indices de motif d'E/S : Dans la deuxième expérience, nous évaluons l'effet des indices de motif d'E/S, à savoir `SEQUENTIAL` et `RANDOM`, sur la performance des lectures. Nous nous attendons à une amélioration de la performance grâce à un préchargement plus précis de la lecture et de la mise en cache lorsque l'indice correspond au motif d'E/S.

Expérience 3 : Évaluation de l'effet de l'indice `DONT_NEED` sur l'éviction du cache : Dans la troisième expérience synthétique, nous évaluons l'effet de l'indice `DONT_NEED` sur la performance des lectures. Nous nous attendons à ce que les données marquées avec cet indice soient supprimées du cache.

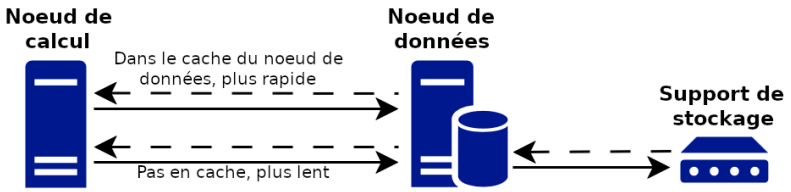


FIGURE 4.1 – Effet de la mise en cache sur le flot de données d'une E/S.

Expérience 4 : Évaluation de l'effet de l'indice `WILL_NEED` sur le préchargement du cache : Dans la quatrième expérience synthétique, nous évaluons l'effet de l'indice `WILL_NEED` sur la performance des lectures et comparons son effet avec un préchargement forcé des données effectué via des lectures asynchrones. Nous nous attendons à ce que cet indice entraîne un préchargement des données. Ainsi, si l'indice est fourni suffisamment tôt pour que le préchargement soit entièrement terminé avant le début des E/S, nous nous attendons à ce que les données soient entièrement mises en cache et que la performance soit maximisée.

Une attention particulière est portée à l'effacement des caches entre chaque mesure afin d'obtenir des métriques de performance précises et cohérentes.

Le cache de pages Linux sur les nœuds de calcul peut être vidé en écrivant dans */proc/sys/vm/drop_caches*. De plus, lors de l'utilisation du système de fichiers Lustre, il est possible d'interagir avec le cache de pages Linux sur les nœuds de données et de le vider en utilisant Lustre ladvise, ce que nous confirmons dans la première expérience. Enfin, GPFS n'utilise pas le cache de pages Linux et ne propose pas de méthode directe pour interagir avec son cache. Pour le vider, nous lisons un grand volume de données sans rapport avec celles utilisées par l'application synthétique afin de remplacer nos données dans son cache.

4.2.1 Caractérisation du système

Nous considérons les trois configurations de référence suivantes, menant à des performances distinctes :

(1) Lorsque les données ne sont **pas mises en cache**.

(2) Lorsque les données sont **mises en cache dans le nœud de calcul**.

(3) Lorsque les données ne sont pas mises en cache dans le nœud de calcul, mais sont **mises en cache dans la mémoire principale du nœud de données**.

Algorithme 2 Caractérisation du système de stockage

Entrée : Fichier cible F de taille S
Sortie : Débits de lecture pour les trois configurations de cache

1: **Pour tout** $C \in$ {non caché, cache nœud de calcul, cache nœud de données} **Faire**
2: **Si** C = non caché **Alors**
3: Exécuter `echo 3 > /proc/sys/vm/drop_caches` sur le nœud de calcul
4: **Si** PFS = Lustre **Alors**
5: Appeler `Lustre ladvise(DONT_NEED)` sur les nœuds de données
6: **Sinon Si** C = cache nœud de calcul **Alors**
7: Lire entièrement F pour le placer en cache local
8: **Sinon Si** C = cache nœud de données **Alors**
9: Lire entièrement F pour le placer dans le cache des nœuds de données
10: Vider le cache du nœud de calcul avec `echo 3 > /proc/sys/vm/drop_caches`
11: Lire à nouveau F, mesurer le volume et le temps d'E/S
12: Calculer le débit D = volume/temps
13: Comparer les débits obtenus pour les trois configurations

Ces trois scénarios sont illustrés dans la figure 4.1. Dans cette figure, les requêtes de données sont représentées par des flèches pleines, et le transfert de données associé par des flèches pointillées. Pour le premier scénario, lorsque les données ne sont pas mises en cache, nous voyons qu'il est nécessaire de les demander à un nœud de données, qui va lui-même les lire sur les supports de stockage. À l'inverse, pour le deuxième scénario, la donnée est déjà stockée dans la mémoire principale du nœud de stockage. Ainsi, un transfert de données coûteux en temps peut être évité, ce qui explique l'absence de flèche pour ce scénario dans la figure. Enfin, le troisième et dernier scénario est un intermédiaire : la donnée n'est pas mise en cache dans le nœud de calcul, mais elle l'est sur le nœud de stockage Lustre. C'est pour cela que nous voyons des flèches entre nœud de calcul et nœud de données, mais pas entre le nœud de données et le support de stockage. Grâce à cela, quand la donnée est demandée par un programme, le nœud de données est interrogé, mais celui-ci n'a pas besoin d'interroger le support de stockage à son tour, ce qui représente une économie de temps.

Le pseudo-code de ces scénarios est détaillé dans l'algorithme 2. On y voit ainsi que pour mesurer la performance de lecture lorsque les données ne sont pas mises en cache (lignes 2 à 5), nous vidons à la fois le cache du nœud de calcul et, dans le cas de Lustre, celui du nœud de données en utilisant la méthode décrite dans la section précédente. Nous procédons ensuite à la lecture d'un fichier cible préalablement créé, noté F, dont la taille

est configurable. Pendant cette opération, nous mesurons le temps et le volume d'E/S, ce qui nous permet de calculer le débit d'E/S obtenu lors de l'expérience (lignes 11 et 12).

Pour le deuxième scénario (lignes 6 à 7), nous cherchons à mesurer le débit lorsque les données sont mises en cache dans le nœud de calcul. Pour cela, il est nécessaire de lire une première fois le fichier cible afin que son contenu entre en cache dans les nœuds de calcul et de données. Ensuite, nous lisons à nouveau le fichier, la mesure du débit en cache étant réalisée lors de cette seconde lecture.

Pour le troisième scénario (lignes 8 à 10), nous visons à mesurer le débit lorsque les données sont mises en cache uniquement dans les nœuds de données. La procédure est similaire à la précédente, avec une étape supplémentaire de vidage du cache client avant la seconde lecture du fichier.

Chacune de ces trois configurations de référence est évaluée avec un motif d'accès séquentiel puis avec un motif d'accès aléatoire, car il est généralement attendu que la performance d'E/S varie en fonction du motif d'accès.

Nous nous attendons à obtenir une performance d'E/S du même ordre de grandeur que celle de la mémoire de travail lorsque les données sont mises en cache dans les nœuds de calcul, une performance du même ordre de grandeur que les supports de stockage lorsque les données ne sont pas mises en cache, et une performance intermédiaire lorsque les données sont mises en cache dans les nœuds de données Lustre.

Algorithme 3 Évaluation de l'effet des indices de motif d'E/S

Entrée : Fichier cible F, motif réel $M_{reel} \in$ {séquentiel, aléatoire}
Sortie : Débits de lecture pour chaque indice de motif appliqué

1: **Pour tout** $M_{indice} \in$ {`SEQUENTIAL`, `RANDOM`} **Faire**
2: Exécuter `echo 3 > /proc/sys/vm/drop_caches` sur le nœud de calcul
3: **Si** PFS = Lustre **Alors**
4: Vider le cache des nœuds de données avec `Lustre ladvise(DONT_NEED)`
5: Appeler `posix_fadvise(fd, 0, 0,` M_{indice}`)` sur F
6: Lire F selon le motif réel M_{reel}
7: Mesurer le temps et le volume d'E/S
8: Calculer le débit $D(M_{indice}, M_{reel})$ = volume/temps
9: Comparer $D(M_{indice}, M_{reel})$ avec la référence sans indice

4.2.2 Caractérisation des indices de motifs d'E/S

Les indices de motif d'E/S, à savoir `SEQUENTIAL` et `RANDOM`, permettent aux développeurs d'informer le système de fichiers parallèle (PFS) du motif d'accès qui sera utilisé pour lire certaines données. Nous notons que bien que ces deux indices de motif d'E/S soient pris en charge par POSIX fadvise, ils ne sont pas supportés par Lustre ladvise, comme illustré dans le tableau 2.2. Afin d'en évaluer l'impact sur la performance d'E/S, nous proposons l'algorithme 3, que nous détaillons ici.

La première étape de cette procédure consiste à vider le cache (lignes 2 à 4). Ensuite, l'indice de motif d'E/S est appliqué à l'ensemble du fichier à l'aide de fadvise (ligne 5). Enfin, nous procédons à la lecture du fichier (ligne 6) tout en mesurant le temps et le volume d'E/S, permettant ainsi de calculer le débit d'E/S (lignes 7 et 8).

Si le PFS prend en compte ces indices de motif d'E/S, nous nous attendons à observer une performance d'E/S similaire ou améliorée lorsque l'indice correspond au motif d'accès de l'application synthétique, et potentiellement une dégradation de la performance lorsque l'indice de motif d'E/S est incorrect.

Algorithme 4 Évaluation de l'effet de l'indice `DONT_NEED`

Entrée : Fichier cible F

Sortie : Débit de lecture après application de l'indice

1: Vider les caches avec `echo 3 > /proc/sys/vm/drop_caches` sur le nœud de calcul
2: **Si** PFS = Lustre **Alors**
3: Vider le cache des nœuds de données avec `Lustre ladvise(DONT_NEED)`
4: Lire entièrement F pour le mettre en cache
5: **Si** expérience POSIX **Alors**
6: Appeler `posix_fadvise(fd, 0, 0, POSIX_FADV_DONTNEED)`
7: **Sinon Si** expérience Lustre **Alors**
8: Appeler `Lustre ladvise(fd, 0, 0, LL_DONTNEED)`
9: Lire à nouveau F, mesurer le volume et le temps d'E/S
10: Calculer le débit D
11: **Si** D proche du débit sans cache **Alors**
12: Conclure à une éviction effective du cache
13: **Sinon**
14: Conclure à un indice ignoré ou une éviction différée

4.2.3 Caractérisation de l'indice "DONT_NEED"

Comme illustré dans le tableau 2.2, l'indice `DONT_NEED` est pris en charge à la fois par POSIX fadvise et Lustre ladvise. Il est utilisé pour informer le système de fichiers parallèle (PFS) que certaines données, potentiellement préalablement mises en cache, ne seront plus nécessaires et peuvent ainsi être supprimées du cache.

Pour évaluer l'effet de cet indice, nous commençons dans l'algorithme 4 par vider le cache afin d'obtenir des résultats reproductibles (lignes 1 à 3). Ensuite, nous lisons une première fois le fichier afin qu'il soit mis en cache (ligne 4), appliquons l'indice au fichier (lignes 5 à 8), puis le lisons à nouveau tout en instrumentant le volume et le temps d'E/S (lignes 9 à 10). En comparant le débit d'E/S obtenu avec celui de la configuration de référence, nous pouvons déterminer si les données, ou une partie d'entre elles, ont effectivement été supprimées du cache.

Si ce n'est pas le cas, cela peut signifier que le PFS a ignoré l'indice ou qu'il a marqué les données pour suppression sans agir immédiatement, pour diverses raisons, telles que l'absence de pression sur le cache.

Algorithme 5 Évaluation de l'effet de l'indice `WILL_NEED`

Entrée : Fichier cible F, délai t_d

Sortie : Débits de lecture après application de l'indice ou préchargement explicite

1: Vider les caches du nœud de calcul avec `echo 3 > /proc/sys/vm/drop_caches`
2: **Si** PFS = Lustre **Alors**
3: Vider le cache des nœuds de données avec `Lustre ladvise(DONT_NEED)`
4: **Si** expérience POSIX **Alors**
5: Appeler `posix_fadvise(fd, 0, 0, POSIX_FADV_WILLNEED)`
6: **Sinon Si** expérience Lustre **Alors**
7: Appeler `Lustre ladvise(fd, 0, 0, LL_WILLNEED)`
8: Attendre un délai t_d
9: Lire F, mesurer le temps et le volume d'E/S
10: Calculer le débit D_{hint} = volume/temps

4.2.4 Caractérisation de l'indice "WILL_NEED"

L'indice `WILL_NEED` est pris en charge à la fois par POSIX fadvise et Lustre ladvise. Il est utilisé pour informer le système de fichiers parallèle (PFS) que certaines données seront

nécessaires dans un futur proche. Ainsi, il est généralement attendu que la réception de cet indice par le PFS soit suivie d'un préchargement asynchrone de ces données.

Dans l'algorithme , pour évaluer l'effet de cet indice, il est appliqué à l'ensemble du fichier un certain temps avant le début de la lecture (lignes 4 à 7), en faisant varier les tailles de fichiers (de très petites à très grandes) ainsi que les délais entre son application et l'opération de lecture (ligne 8). On peut s'attendre à ce que plus l'application de l'indice sera anticipée, plus la quantité de données préchargées avant le début de la lecture sera importante, et meilleure sera la performance d'E/S.

Comme configuration de référence supplémentaire dans cette expérience, les indices `WILL_NEED` intégrés dans notre application synthétique lors des expériences précédentes peuvent être remplacés par des primitives de préchargement explicites. Dans notre cas, nous utilisons des lectures asynchrones POSIX, qui contrairement à l'indice `WILL_NEED` garantissent que les données seront effectivement lues et vraisemblablement mises en cache par le système de fichiers.

4.3 Évaluation expérimentale

Dans cette section, la méthodologie proposée est mise en œuvre. Les ressources matérielles et logicielles employées sont listées, et les résultats décrits et analysés.

4.3.1 Ressources matérielles et logicielles

La méthodologie est appliquée sur un cluster HPC de petite taille. Il est composé de 20 nœuds de calcul équipés de deux processeurs AMD EPYC 7282 dotés de 16 cœurs chacun.

Deux systèmes de fichiers parallèles ont été utilisés dans cette étude. Le premier est basé sur GPFS [83] version 5.1.8.0, avec 8 volumes de 50 To chacun, pour une capacité totale de 400 To. Nous notons que GPFS dispose de son propre cache, distinct du cache de pages de Linux. Ce pool de pages a une taille fixe de 4 Go. Le deuxième système de fichiers utilisé est basé sur Lustre, avec la version 2.12.5.0 pour le serveur et la version 2.12.9 pour le client, comprenant 8 OST de 5 To et 1 MDT de 500 Go, pour une capacité totale de 40 To.

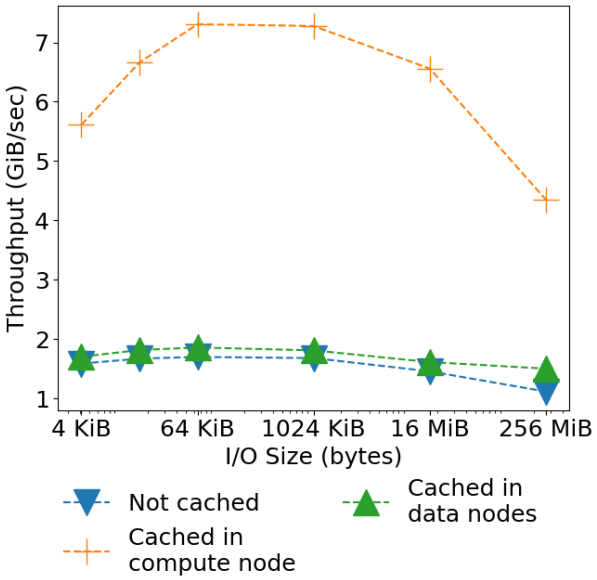


FIGURE 4.2 – Lecture séquentielle avec Lustre

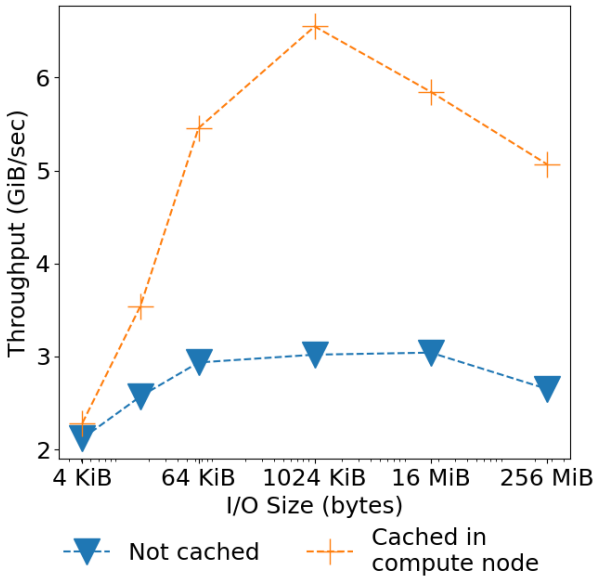


FIGURE 4.3 – Lecture séquentielle avec GPFS

4.3.2 Visualisation des résultats expérimentaux

Toutes les expériences décrites dans la méthodologie ont été réalisées à l'aide d'une application synthétique. Celle-ci est configurable, et a permis de faire varier les tailles d'E/S, les temps d'inter-arrivées, les motifs d'E/S ainsi que les tailles des fichiers lus. Compte tenu du grand nombre de paramètres impliqués, nous avons choisi, dans les visualisations ci-dessous, de fixer la taille des fichiers à 16 GiB et le temps d'inter-arrivée à 0. Dans le cas où la modification de ces valeurs aurait une influence significative sur les résultats, cela sera explicitement mentionné dans l'analyse.

4.3.3 Caractérisation du système

L'objectif de la première expérience est de mesurer les performances d'E/S de référence, c'est-à-dire le débit, dans les trois situations de base suivantes : lorsque les données ne sont pas en cache, lorsqu'elles sont en cache sur le nœud de calcul, et lorsqu'elles sont en cache sur les nœuds de données dans le cas de Lustre.

Dans les figures 4.2 et 4.3, le motif d'E/S est séquentiel. On observe sur Lustre une amélioration significative de la performance en lecture lorsque les données sont mises en cache dans la mémoire principale du nœud de calcul (jusqu'à 7 GiB/s), en comparaison

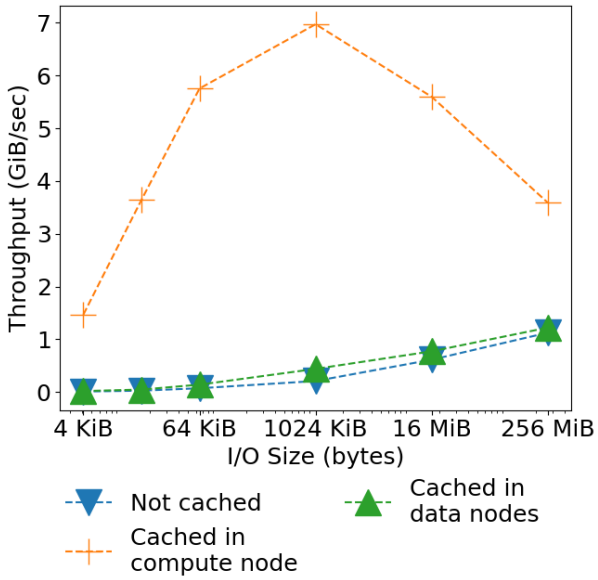


FIGURE 4.4 – Lecture aléatoire avec Lustre

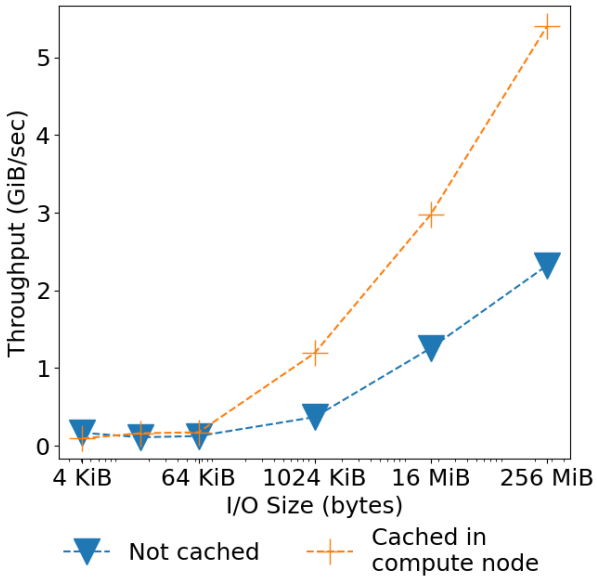


FIGURE 4.5 – Lecture aléatoire avec GPFS

avec lorsqu'elles sont en cache dans la mémoire des nœuds de données (jusqu'à 2 GiB/s) ou lorsqu'elles ne sont pas dans le cache (jusqu'à 1.7 GiB/s). Cette tendance se vérifie également avec GPFS, ainsi que pour les différentes tailles de fichiers et valeurs de temps d'inter-arrivée testées dans cette expérience, comme attendu : on retrouve bien une performance du même ordre de grandeur que la mémoire de travail lorsque les données sont mises en cache dans la mémoire principale du nœud de calcul, une performance du même ordre de grandeur que le support de stockage quand les données ne sont pas mises en cache, et une performance intermédiaire quand les données sont en cache dans un nœud de données. Cette baisse de performance s'explique par l'augmentation de la latence d'accès et des éventuelles limitations du réseau lorsque les données doivent être récupérées en dehors de la mémoire locale du nœud de calcul, ce qui limite le débit global par rapport à un accès entièrement en mémoire.

Dans les figures 4.4 et 4.5, lorsque le motif d'E/S est aléatoire, les performances mesurées diffèrent légèrement. Bien que l'accès à des données non mises en cache ne soit jamais plus rapide que l'accès à des données en cache, le gain en débit obtenu grâce à la mise en cache reste faible pour de très petites tailles d'E/S, et maximal pour des tailles plus élevées.

On remarque que, sur Lustre comme sur GPFS, le débit atteint lors de lectures séquentielles est proche de l'optimal pour une taille d'E/S de 1 MiB, et diminue pour des

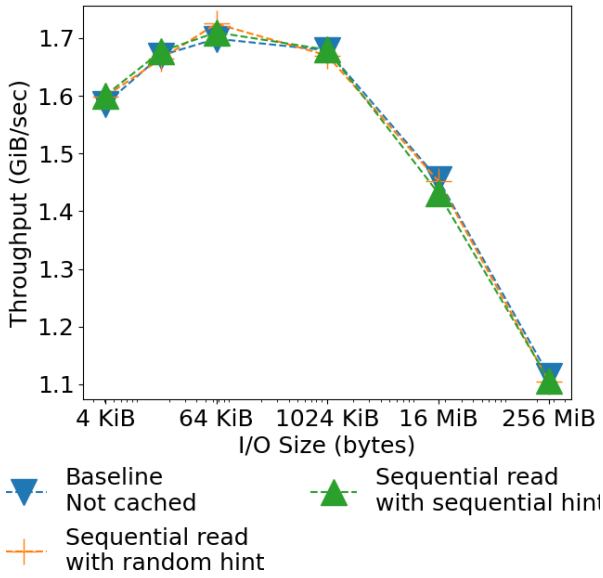


FIGURE 4.6 – Lecture séquentielle avec Lustre et des indices de motif d'E/S

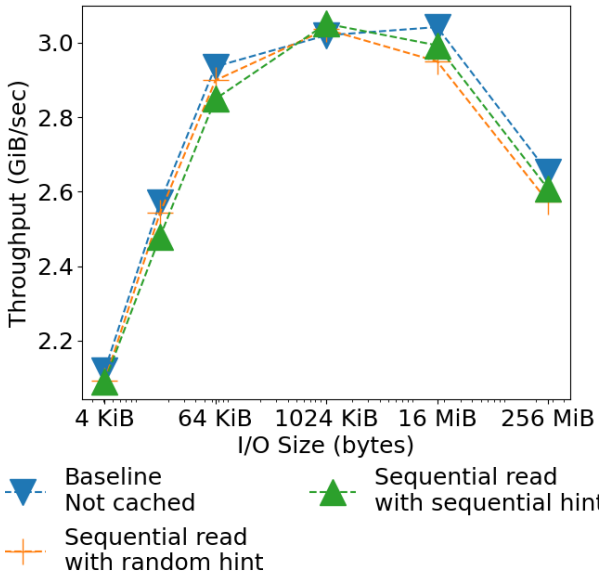


FIGURE 4.7 – Lecture séquentielle avec GPFS et des indices de motif d'E/S

tailles inférieures ou supérieures. Une observation similaire est faite sur Lustre pour les lectures aléatoires. Nous émettons l'hypothèse que cette baisse de performance peut être liée à des surcoûts réseau et pilote pour les petites tailles d'E/S, ainsi qu'à la configuration de *striping* pour les tailles plus grandes.

De manière générale, l'accès aux données mises en cache sur les nœuds de calcul est le plus performant, suivi par l'accès aux caches des nœuds de données, si disponible. Enfin, l'accès aux données non mises en cache est le plus lent. Les valeurs numériques obtenues sont utilisées dans la suite de cette étude comme valeurs de référence.

4.3.4 Caractérisation des indices de motifs d'E/S

L'objectif de la deuxième expérience est d'évaluer l'effet des indices de motif d'E/S `SEQUENTIAL` et `RANDOM`.

Dans les figures 4.6 et 4.7, différents indices de motif d'E/S ont été testés lors de lectures séquentielles. Dans ces graphes, en plus des mesures spécifiques à cette expérience, nous faisons référence à deux cas de référence issus de l'expérience précédente : les mesures réalisées sans mise en cache servent de référence sans indice de motif, tandis que celles effectuées avec des données en cache dans la mémoire du nœud de calcul représentent les

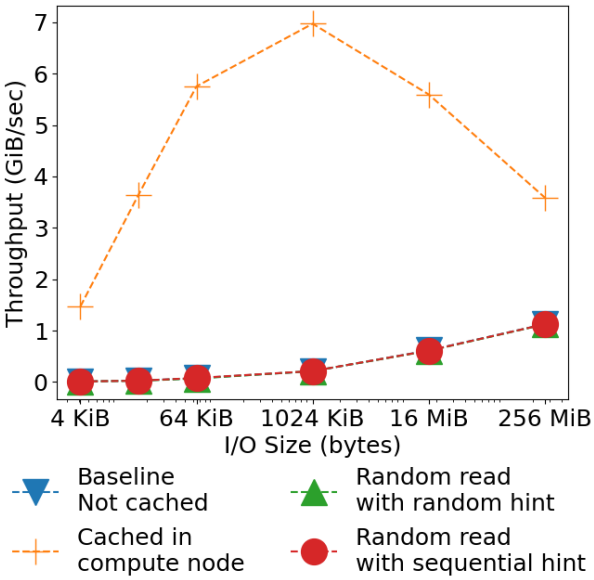


FIGURE 4.8 – Lecture aléatoire avec Lustre et des indices de motif d'E/S

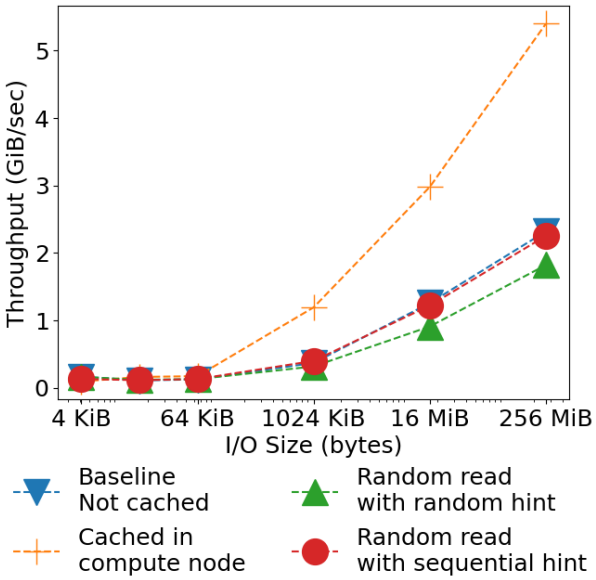


FIGURE 4.9 – Lecture aléatoire avec GPFS et des indices de motif d'E/S

performances attendues les plus élevées pour un motif et une taille d'E/S donnés.

On observe que l'ajout d'un indice de motif d'E/S n'a pas d'effet significatif sur les performances de lecture séquentielle, que ce soit sur GPFS ou Lustre. Ainsi sur Lustre, les performances avec indice sont équivalentes à celles de la référence sans cache et sans indice, avec une variation du débit pour une même taille d'E/S inférieure à 2% du débit (jusqu'à 50 KiB/s). Sur GPFS, une variation du débit inférieure à 5% du débit (jusqu'à 80 KiB/s) est observée selon l'indice utilisé. Aucune tendance claire ne se dégage. Ces chiffres sont les mêmes quel que soit le délai entre application des indices et E/S, ce qui consolide l'idée que les indices `SEQUENTIAL` sont ignorés.

Dans les figures 4.8 et 4.9, le motif d'E/S est aléatoire. Sur Lustre, les résultats confirment que les indices de motif d'E/S sont ignorés, car le débit reste identique, quel que soit l'indice fourni. En revanche, sur GPFS, bien que l'indice `SEQUENTIAL` n'entraîne aucune différence notable (et semble donc également ignoré), l'indice `RANDOM` a un effet légèrement négatif sur les performances (la courbe verte montre un débit inférieur dans la figure 4.9). Ce comportement peut s'expliquer par le fait que GPFS ne met pas en cache les données marquées comme aléatoires, alors que notre application synthétique réalise des E/S avec des positions aléatoires dans le fichier, ce qui signifie qu'il peut arriver que deux E/S soient proches et qu'il y ait une faible localité temporelle. Cela entraîne des

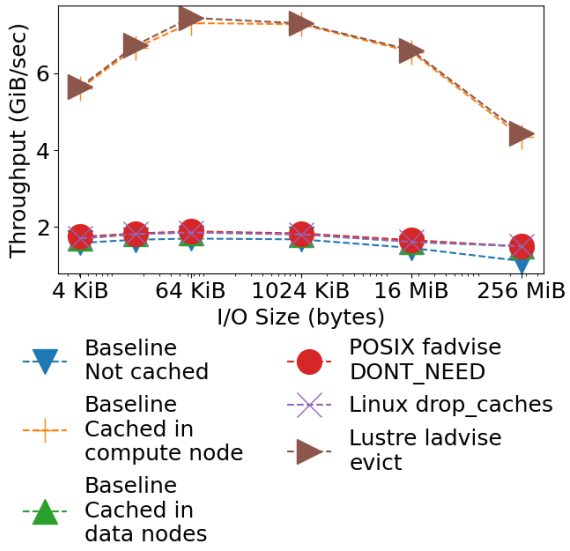


FIGURE 4.10 – Lecture séquentielle de données préalablement mises en cache puis marquées avec `DONT_NEED`, avec Lustre

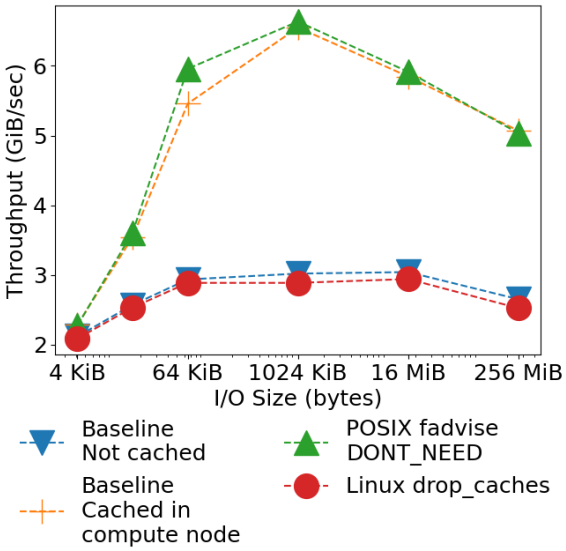


FIGURE 4.11 – Lecture séquentielle de données préalablement mises en cache puis marquées avec `DONT_NEED`, avec GPFS

ratés de cache et une baisse de débit.

De manière générale, l'indice `SEQUENTIAL` ne semble avoir aucun effet sur la performance d'E/S pour les deux systèmes de fichiers parallèles. Cela est attendu, dans la mesure où les lectures séquentielles constituent le motif dominant en calcul haute performance, bien que cette tendance évolue avec l'émergence de nouveaux usages tels que l'apprentissage automatique [10, 7]. **L'indice `RANDOM`, bien qu'ignoré sur Lustre, semble être pris en compte par GPFS.** Toutefois, son utilisation peut avoir un impact négatif sur la performance, probablement parce que GPFS s'attend à ce que les données accédées aléatoirement ne soient pas relues du fait de la très faible taille de cache par défaut sur ce PFS, et ne méritent donc pas d'être mises en cache — ce qui, dans notre cas de test, n'était pas toujours vrai.

4.3.5 Caractérisation de l'indice "`DONT_NEED`"

Lorsque l'indice `DONT_NEED` de *POSIX fadvise* est utilisé avec Lustre, on observe dans la figure 4.10 des performances équivalentes à celles de la situation de référence où les données sont présentes dans le cache des nœuds de données, mais absentes du cache du nœud de calcul. Cela peut s'interpréter comme une éviction immédiate des données du cache du nœud de calcul suite à la réception de l'indice, mais pas des nœuds de données,

où elles peuvent encore être accédées avec une meilleure performance que si elles n'étaient pas en cache du tout.

Dans la figure 4.11, on constate que l'utilisation du même indice sur GPFS n'a pas d'effet clair sur la performance d'E/S, en comparaison avec la référence où les données étaient en cache. Cela suggère que, contrairement à Lustre, GPFS n'évince pas directement les données suite à la réception de cet indice. Il est possible que GPFS procède à une éviction différée, ou bien qu'il ignore simplement l'indice.

Dans les figures 4.10 et 4.11, lorsque l'indice `DONT_NEED` de *Lustre ladvise* est utilisé, les performances sont similaires à celles observées lorsque les données sont en cache sur le nœud de calcul. Cela s'explique probablement par le fait que *ladvise* ne cible que le cache des nœuds de données, et non celui des nœuds de calcul.

De manière générale, l'indice `DONT_NEED` a des effets différents selon le système de fichiers parallèle utilisé. Sur Lustre, nous avons observé une éviction directe des données du cache du nœud de calcul lors de l'utilisation de *POSIX fadvise*, et du cache des nœuds de données avec *Lustre ladvise*. Sur GPFS, aucune éviction immédiate n'a été constatée, ce qui suggère que l'indice est soit ignoré, soit traité de manière différée.

4.3.6 Caractérisation de l'indice "`WILL_NEED`"

L'objectif de la quatrième expérience est d'évaluer l'effet de l'indice `WILL_NEED`. Cet indice est généralement utilisé dans l'intention d'effectuer un préchargement asynchrone des données, avec un léger décalage avant leur utilisation effective.

Nous avons évalué l'utilisation de l'indice `WILL_NEED` sur l'ensemble du fichier, en faisant varier le délai entre l'appel à la fonction de conseil et le début des lectures. Dans les figures 4.12 et 4.14, ce délai est fixé à une milliseconde, tandis que dans les figures 4.13 et 4.15, il est nul. Afin de définir une référence supplémentaire, nous avons également intégré une primitive de préchargement asynchrone (préchargement forcé), dont le débit est présenté dans les figures 4.12 et 4.14.

Lors de l'utilisation de l'indice `WILL_NEED` avec *POSIX fadvise*, aucune différence de performance n'est observée dans les figures 4.12 à 4.14 par rapport à la référence sans cache introduite dans la figure 4.2. Cela peut être interprété soit comme un indice ignoré, soit comme un délai insuffisant pour permettre au système de fichiers parallèle (PFS) de mettre en cache les données. En revanche, lorsqu'un préchargement est forcé via des E/S asynchrones avec un délai d'une milliseconde, un gain de performance significatif est observé : jusqu'à un facteur 3 sur Lustre et un facteur 2 sur GPFS. Cela suggère que

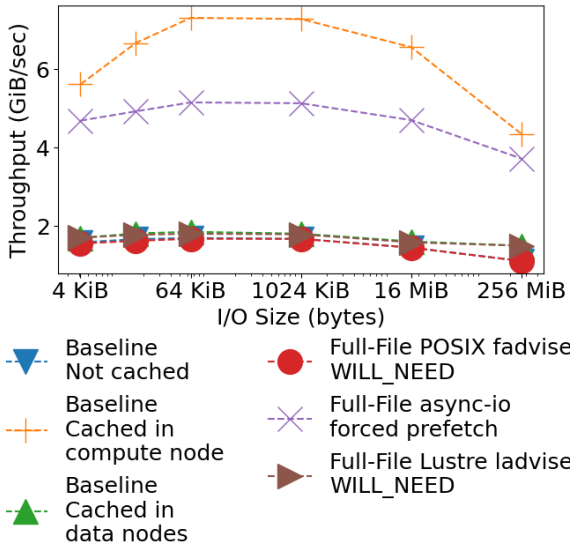


FIGURE 4.12 – Lecture séquentielle sur Lustre 1 ms après avoir appliqué un indice `WILL_NEED` sur la totalité du fichier cible

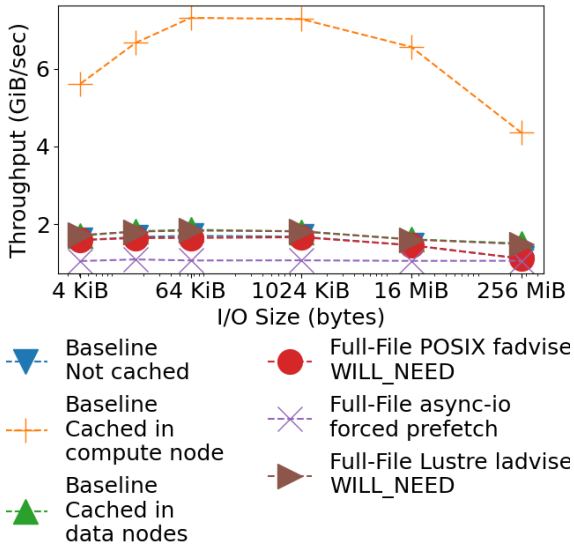


FIGURE 4.13 – Lecture séquentielle sur Lustre directement après avoir appliqué un indice `WILL_NEED` sur la totalité du fichier cible

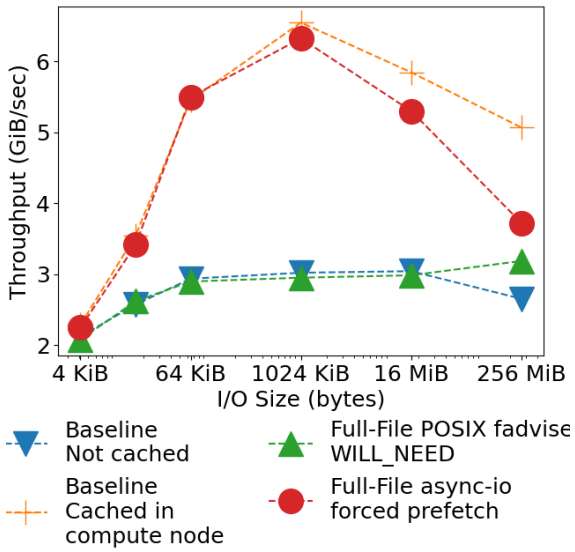


FIGURE 4.14 – Lecture séquentielle sur GPFS 1 ms après avoir appliqué un indice `WILL_NEED` sur la totalité du fichier cible

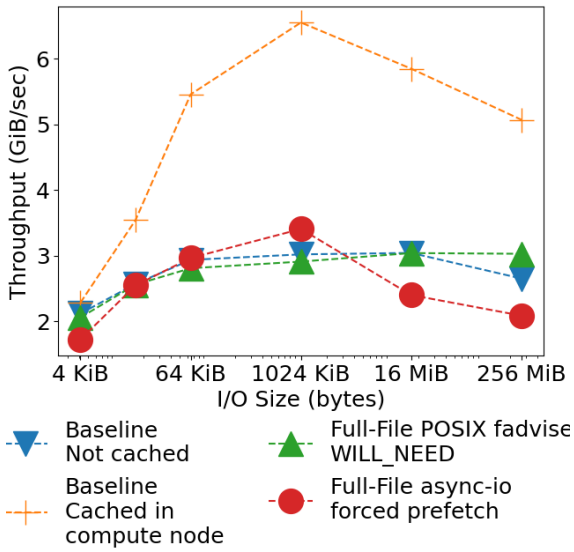


FIGURE 4.15 – Lecture séquentielle sur GPFS directement après avoir appliqué un indice `WILL_NEED` sur la totalité du fichier cible

l'indice `WILL_NEED` est probablement ignoré, ou qu'il n'est pas suffisamment réactif.

Dans ces mêmes figures, on constate que l'indice `WILL_NEED` de *Lustre ladvise* permet d'atteindre un niveau de performance équivalent à celui obtenu lorsque les données sont préchargées sur les nœuds de données. Ce niveau reste toutefois inférieur à celui observé lorsque les données sont en cache sur les nœuds de calcul.

Il convient de noter que, bien qu'un préchargement forcé constitue une bonne référence, il ne peut pas remplacer complètement l'indice `WILL_NEED`. En effet, dans le cas de Lustre, ce préchargement peut entraîner une réduction du débit d'un facteur 0.5 lorsque le préchargement et les E/S commencent simultanément (sans délai).

Par ailleurs, cette expérience montre que le temps d'inter-arrivée des E/S a un effet notable sur la performance. Un temps d'inter-arrivée plus élevé améliore le débit lors d'un préchargement forcé. Par exemple, sur Lustre, un préchargement forcé par E/S asynchrone sans délai, associé à un temps d'inter-arrivée nul, a eu un impact négatif sur la performance. En revanche, l'ajout d'un temps d'inter-arrivée de 10 µs a permis d'augmenter le débit jusqu'à un facteur 3. Cela peut s'expliquer par le fait que le PFS dispose alors de plus de temps pour effectuer le préchargement avant que les E/S ne le saturent.

Contrairement aux attentes, l'indication *POSIX fadvise* `WILL_NEED` ne garantit pas le préchargement effectif des données par le PFS. À la place, un préchargement forcé avec un léger délai (de l'ordre de 1 ms) avant le début des E/S peut améliorer la performance jusqu'à un facteur 3.

4.4 Proposition de consignes d'utilisation des indices d'usage de données

Dans cette section, nous fournissons des recommandations visant à aider les développeurs d'applications à maximiser la performance, et à identifier les limitations des fonctions d'indice d'usage des données dans un contexte HPC.

Utilisation des indices de motif d'E/S

D'après nos observations sur GPFS et Lustre, les indices de motifs d'E/S n'ont pas d'effet notable sur les performances lors d'opérations séquentielles. Cela est probablement dû au fait que le schéma d'E/S séquentiel reste le plus courant [10, 7]. La documentation de GPFS va jusqu'à indiquer explicitement que, par conception, le cache de pages de

GPFS (appelé *page pool*), avec sa taille par défaut de quelques GiB, ne peut accélérer que les E/S séquentielles [41]. Toutefois, lors de l'évaluation de l'effet de ces indices sur les lectures aléatoires, nous avons observé que les performances n'étaient pas affectées sur Lustre, mais étaient dégradées sur GPFS lors de l'utilisation de l'indice `RANDOM`. Cela est probablement dû au fait que, recevant cet indice et observant un motif de lecture aléatoire, GPFS cesse de mettre les données en cache pour éviter de gaspiller de l'espace dans son cache très limité, ce qui entraîne une légère baisse de performance, notre test de lecture aléatoire pouvant de manière rare et aléatoire relire certaines données plusieurs fois. **Ainsi, nous suggérons aux développeurs d'applications de ne pas se concentrer sur l'ajout d'indices pour les lectures séquentielles, car on peut supposer que les PFS les traiteront efficacement par défaut, et de n'ajouter des indices pour les lectures aléatoires qu'après avoir évalué leur PFS ou s'être assuré de l'absence de localité temporelle et spatiale.**

Utilisation de l'indice `DONT_NEED`

D'après nos observations, les données marquées avec cet indice via `POSIX fadvise` sur Lustre sont immédiatement évincées du cache de la mémoire des nœuds de calcul. Cela revient, en pratique, à une éviction de cache plus ciblée que l'usage de l'interface Linux */proc/sys/vm/drop_caches*. De plus, nous avons observé que l'utilisation de l'indice `DONT_NEED` via `Lustre ladvise` entraîne l'éviction des données du cache des nœuds de données, mais pas du cache des nœuds de calcul, ce qui constitue une fonctionnalité intéressante. En revanche, sur GPFS, nous n'avons constaté aucun effet sur les performances des E/S après application de cet indice, ce qui suggère que celle-ci est soit ignorée, soit que l'éviction n'est pas immédiate. **Nous montrons que, sur certains systèmes de fichiers tels que Lustre, l'utilisation de l'indice `DONT_NEED` permet de libérer de manière immédiate ou différée les données inutiles du cache. Sur d'autres, il est ignoré sans impacter la performance. Ainsi, nous suggérons aux développeurs d'utiliser cet indice pour les données qui ne seront pas réutilisées, telles que les journaux ou les sauvegardes, afin de réduire l'utilisation du cache.**

Utilisation de l'indice `WILL_NEED`

D'après nos observations, les données marquées avec cet indice via `POSIX fadvise` ne sont pas nécessairement préchargées sur les PFS testés. Cependant, nous avons constaté que `Lustre ladvise` permet effectivement d'augmenter le débit des E/S en préchargeant

les données dans le cache des nœuds de données. **Ainsi, nous suggérons aux développeurs d'applications ciblant Lustre de se concentrer sur l'ajout des indices `WILL_NEED` via `Lustre ladvise`. Pour les développeurs ne ciblant pas Lustre, nous suggérons d'envisager un préchargement asynchrone des données, tant qu'un délai existe entre le préchargement et l'accès aux données.**

4.5 Conclusion

Dans ce chapitre, nous avons proposé une méthodologie systématique pour évaluer l'efficacité et les effets réels des indices d'usage de données sur deux systèmes de fichiers parallèles largement utilisés dans le calcul haute performance : Lustre et GPFS. L'approche expérimentale mise en œuvre repose sur un ensemble de scénarios représentatifs, permettant d'isoler l'influence de chaque indice et de caractériser leur impact sur la hiérarchie des caches, à la fois au niveau des nœuds de calcul et des nœuds de données.

Nos résultats montrent que l'interprétation et l'efficacité des indices d'usage de données dépendent fortement de l'implémentation sous-jacente du système de fichiers parallèle. Ainsi, nous avons observé que :

- Les indices de motif d'E/S (`SEQUENTIAL` et `RANDOM`) n'ont qu'un effet marginal, voire nul, sur la performance. Leur impact est pratiquement inexistant sur Lustre, et légèrement négatif sur GPFS lorsque l'indice `RANDOM` est utilisé, probablement en raison de la désactivation complète du mécanisme de cache pour la donnée.
- L'indice `DONT_NEED` est traité différemment selon le PFS : sur Lustre, il provoque une éviction immédiate des données du cache du nœud de calcul (et, via `ladvise`, du cache des nœuds de données), tandis que sur GPFS, il semble être ignoré ou appliqué de manière différée. Ce comportement confirme la pertinence de cet indice pour optimiser la gestion de la mémoire sur Lustre, notamment dans les workflows où de grands volumes de données sont produits et consommés une seule fois.
- L'indice `WILL_NEED`, censé initier un préchargement asynchrone, n'a montré aucun effet mesurable sur la performance avec `POSIX fadvise`. En revanche, son application avec `Lustre ladvise` a permis d'améliorer le débit en préparant les données dans le cache des nœuds de données, bien que cet effet reste inférieur à celui d'un préchargement forcé explicitement implémenté au niveau applicatif.

Ces observations mettent en évidence une absence d'uniformité dans la prise en compte des indices d'usage entre différents systèmes de fichiers parallèles, ainsi qu'un écart im-

portant entre la sémantique théorique de POSIX et les comportements concrets observés. Elles suggèrent qu'en pratique, ces indices ne peuvent pas encore être utilisés de manière portable ni fiable dans les applications HPC sans validation préalable sur le système cible.

De manière plus générale, notre étude démontre que :

— Les effets de cache, souvent opaques pour l'utilisateur, jouent un rôle déterminant dans les performances d'E/S en environnement parallèle.

— Les fonctions d'indice d'usage peuvent être exploitées pour affiner la gestion de ces caches, mais uniquement lorsqu'elles sont effectivement implémentées et interprétées par le PFS.

— Une connaissance fine du comportement spécifique de chaque système de fichiers est indispensable pour exploiter ces mécanismes avec efficacité.

En conséquence, nous avons formulé un ensemble de recommandations pratiques à destination des développeurs d'applications HPC, visant à orienter l'usage de ces indices selon le contexte et le PFS utilisé.

Enfin, cette étude ouvre plusieurs perspectives de recherche. Il serait intéressant d'explorer l'impact de ces indices dans des scénarios multi-utilisateurs et en environnement fortement concurrent, leur interaction avec les optimisations spécifiques de bibliothèques d'E/S comme MPI-IO ou HDF5, et plus globalement d'étendre notre approche à d'autres PFS.

En somme, nos travaux contribuent à une meilleure compréhension du comportement réel des indices d'usage de données dans les PFS modernes, et posent les bases d'une approche plus rationnelle pour leur exploitation dans les applications HPC, en vue d'une amélioration de la performance et, indirectement, de leur consommation énergétique.

Dans le chapitre suivant, nous nous penchons sur **l'utilisation du DVFS**. Il serait souhaitable d'actionner le DVFS pendant les phases d'E/S. Cependant, l'effet réel du DVFS pendant les E/S dans un contexte reste encore mal défini. Le chapitre suivant se concentre donc sur l'analyse expérimentale de leur impact sur la performance et l'énergie des E/S.

Chapitre 5

Contribution 3 : Étude de l'effet de la variation de la fréquence et de la tension sur des applications HPC

5.1 Motivation

Comme introduit précédemment dans ce manuscrit, l'augmentation de l'échelle des clusters de calcul haute performance (HPC) entraîne des défis importants en matière de consommation énergétique et d'efficacité énergétique. La variation dynamique de la tension et de la fréquence (*Dynamic Voltage and Frequency Scaling*, DVFS) est une technique couramment utilisée pour réduire la consommation électrique des processeurs. **Toutefois, si l'effet du DVFS sur les tâches de calcul classiques a déjà été largement étudié, son impact sur les charges intensives en E/S, et tout spécialement dans un contexte HPC, reste très peu documenté dans la littérature.** Cela est problématique car la performance et la consommation énergétique des E/S se répercutent sur celles des applications HPC qui réalisent des E/S [45].

C'est pourquoi dans ce chapitre, nous proposons une méthodologie permettant l'étude de l'impact du DVFS. Cette méthodologie est appliquée sur un nœud de calcul HPC équipé de deux processeurs AMD EPYC Rome 7282. Nos contributions sont les suivantes : (1) nous présentons une méthodologie permettant d'évaluer, via l'utilisation d'applications synthétiques soigneusement sélectionnées, l'impact du DVFS sur la performance et la consommation énergétique d'un système HPC ; (2) nous produisons un ensemble de résultats et d'analyses concernant le comportement du DVFS ; et (3) nous proposons sur la base de nos résultats des recommandations pratiques pour l'utilisation du DVFS dans les applications HPC.

5.2 Proposition : méthodologie d'évaluation de l'effet du DVFS sur des applications haute performance

Comme présenté dans la Section 2.3.2, la gestion du DVFS sous Linux repose sur deux mécanismes complémentaires : *CPUfreq*, qui contrôle la fréquence du processeur pendant les états actifs (P-states), et *CPUidle*, qui gère les états d'inactivité (C-states). Notre méthodologie d'évaluation se décline donc en deux ensembles d'expériences distincts : le premier vise à caractériser l'impact du DVFS pendant les phases actives du calcul, tandis que le second s'intéresse à son effet durant les phases d'inactivité. Chaque ensemble est détaillé dans une sous-section dédiée.

Afin d'assurer la reproductibilité et la stabilité des mesures, nous utilisons des applications synthétiques. Leur comportement contrôlable et leur faible variabilité permettent d'isoler plus précisément l'influence du DVFS, ce qui serait difficile à obtenir avec des charges de travail réelles, plus hétérogènes [91].

Étant donné que le DVFS affecte à la fois les performances et la consommation énergétique, trois métriques principales sont retenues pour l'analyse :

1. **Durée d'exécution (s) :** temps total d'exécution de la charge synthétique pour une fréquence CPU donnée. Une durée plus longue indique une performance plus faible.
2. **Consommation énergétique (J) :** énergie totale consommée pendant l'exécution pour une configuration *CPUfreq* donnée.
3. **Puissance moyenne (J/s, W) :** puissance moyenne calculée à partir de la consommation énergétique et de la durée d'exécution.

La mesure de la durée d'exécution est triviale, mais celle de la consommation énergétique l'est moins. Nous faisons l'hypothèse que le nœud de calcul dispose d'un système de mesure fiable couvrant l'ensemble de sa consommation énergétique, avec une précision suffisante et un impact négligeable sur les performances.

5.2.1 Élaboration d'applications synthétiques pour la caractérisation du DVFS

Dans la Section 2.3.2, nous avons vu que l'effet du DVFS sur la performance et sur l'énergie dépendait de la nature de l'application. Afin de permettre un contrôle précis des

Algorithme 6 Charge de travail synthétique dont la performance est limitée par le processeur

```
1: random_state ← RANDOM_SEED
2: procedure NaivePseudoRandom( )
3:     random_state ← (MUL * random_state + INC)%MOD
4: procedure CPU-Fakeapp(N)
5:     ▷ Utilisation de MPI pour le parallélisme
6:     MPI_Init( ) ;
7:     ▷ Appel de NaivePseudoRandom N fois
8:     Pour iteration = 0, 1, 2, ..., N − 1 Faire
9:         NaivePseudoRandom( )
10:    MPI_Finalize( ) ;
```

Algorithme 7 Charge de travail synthétique dont la performance est limitée par la mémoire

```
1: procedure MemoryFakeapp(N, taille)
2:     ▷ Utilisation de MPI pour le parallélisme
3:     MPI_Init( ) ;
4:     ▷ Allocation d'un tampon de taille taille
5:     tampon ← Malloc(taille)
6:     ▷ Exécution de N opérations mémoire
7:     Pour itération = 0, 1, 2, ..., N − 1 Faire
8:         MEMSET(tampon, itération)
9:     MPI_Finalize( ) ;
```

caractéristiques des charges de travail synthétiques exécutées dans cette méthodologie, nous proposons trois charges de travail synthétiques :

1) Une tâche de calcul intensive en calcul (*CPU-bound*) : une charge de travail synthétique avec une forte utilisation du processeur est créée. Cette application synthétique est nommée "**CPU-Fakeapp**", et son pseudo-code est présenté dans l'Alg. 6. Elle est lancée avec un processus MPI par cœur physique, chaque processus générant des nombres pseudo-aléatoires (lignes 3 et 9 de l'Alg. 6). Cet algorithme est programmé en langage C, son binaire est de très petite taille et utilise peu de variables, ce qui garantit une utilisation majoritaire du cache du processeur plutôt que de la mémoire principale.

2) Une tâche de calcul intensive en mémoire (*memory-bound*) : une charge de travail synthétique exploitant pleinement le débit mémoire du processeur est conçue. Bien qu'un outil open-source tel que STREAM [62] aurait pu être utilisé, celui-ci n'est pas conçu pour une exécution parallèle et ne permet donc pas de représenter une tâche de calcul haute performance (HPC). Nous nommons notre application synthétique "**Memory-Fakeapp**", avec un processus MPI lancé par cœur physique. Son pseudo-code est donné dans l'Alg. 7. Cette charge de travail effectue des appels à la fonction UNIX *memset*

Algorithme 8 Charge de travail synthétique dont la performance est limitée par le stockage

```
1: procedure I/O FAKEAPP(N_E/S, taille_E/S)
2:     ▷ Utilisation de MPI pour le parallélisme
3:     MPI_INIT( ) ;
4:     ▷ Un fichier unique par processus
5:     fichier = OPEN( )
6:     Pour itération = 0, 1, 2, ..., N_E/S − 1 Faire
7:         WRITE(taille_E/S, fichier)
8:     CLOSE(fichier)
9:     MPI_FINALIZE( ) ;
```

Nom	Catégorie	% CPU inactif	% CPU attente des E/S	% CPU actif	Durée (2,8 GHz, tous les C-states)	Paramètres
CPU-Fakeapp	Tâche de calcul (*limitée par le CPU*)	2%	0%	98%	75 s	Taille de la séquence pseudo-aléatoire à générer = 20e9
Memory-Fakeapp	Tâche de calcul (*limitée par la mémoire*)	2%	0%	98%	115 s	Volume mémoire à accéder par processus = 400 GiB
I/O-Fakeapp	E/S sur fichiers Dépendance aux données (MPI-IO)	97%	2%	1%	100–500 s (selon la taille des E/S)	Taille des E/S, type d'E/S (en cache ou direct), nombre d'E/S = variable

FIGURE 5.1 – Applications synthétiques utilisées

(ligne 8 de l'Alg. 7), initialisant tous les octets d'une allocation mémoire cible à une valeur différente à chaque itération de boucle (ligne 5 de l'Alg. 7).

3) Une tâche intensive en E/S (*IO-bound*) : nous concevons une charge de travail synthétique parallèle dédiée aux entrées/sorties, appelée "**I/O-Fakeapp**", s'exécutant avec un processus par cœur physique. Son pseudo-code est présenté dans l'Alg. 8. Bien que cette application synthétique soit moins configurable que des outils existants comme IOR [42] ou FIO [3], son code volontairement simplifié permet de garantir que les métriques évaluées sont directement liées aux opérations d'E/S, et non à la logique interne de l'outil. Nous avons créé une version de cette charge synthétique pour les E/S POSIX en mode bufferisé (par défaut), ainsi qu'une autre version pour les E/S POSIX en mode direct.

Un résumé des caractéristiques de ces trois charges de travail synthétiques ainsi que de leurs paramètres est présenté dans le Tableau 5.1. On note que parmi les caractéristiques indiquées dans ce tableau se trouve notamment le pourcentage de temps passé dans un

état inactif, en attente d'E/S, et en exécution. Ces métriques ont été obtenues à l'aide de la commande *vmstat*, qui est une commande couramment utilisée pour obtenir des informations sur les processus, la mémoire, ainsi qu'une variété d'autres indicateurs [96].

5.2.2 Caractérisation du DVFS durant les états actifs du processeur

L'objectif de cette première série d'expériences est d'analyser l'effet de la variation de la fréquence du processeur durant ses états actifs (**P-states**) sur la consommation énergétique et la performance des applications HPC, en exécutant les mêmes charges de travail synthétiques avec différentes fréquences de processeur. Lors de ces expérimentations, la configuration par défaut de CPUidle est conservée, ce qui signifie que tous les C-states sont activés.

Pour ce faire, nous utilisons exclusivement le gouverneur *userspace* de CPUfreq, tel que préalablement introduit dans la Section 2.3.2, car nous nous concentrons sur l'effet du changement de fréquence et de tension (DVFS) sur la consommation énergétique et les performances des applications HPC, plutôt que sur les politiques (gouverneurs) permettant d'adapter dynamiquement cette variation. Le gouverneur *userspace* permet de régler manuellement la fréquence.

Les charges de travail synthétiques introduites dans la sous-section 5.2.1 sont exécutées une fois pour chaque fréquence de processeur disponible sur nos nœuds de calcul. Lors de ces exécutions, la consommation énergétique ainsi que la durée des tâches sont instrumentées.

Nous nous attendons globalement à ce que des fréquences de processeur plus élevées conduisent à une meilleures performances, mais aussi à une consommation énergétique plus importante, comparativement à des fréquences plus faibles. Comme présenté dans la Section 2.3.2, cela peut entraîner une consommation énergétique totale parfois supérieure, parfois inférieure, en fonction de la charge de travail. C'est afin de mieux comprendre cette dynamique que cette première campagne expérimentale est menée.

5.2.3 Caractérisation du DVFS durant les états inactifs du processeur

L'objectif de cette série d'expériences est d'analyser l'effet du DVFS sur la performance et la consommation énergétique d'applications HPC pendant les périodes d'inactivité du

processeur (**C-states**). Pour isoler spécifiquement l'impact de la gestion des états inactifs, la fréquence du processeur (P-state) est maintenue fixe à sa valeur maximale par défaut, de manière à éliminer toute interaction avec la modulation de fréquence.

Les charges de travail synthétiques présentées dans la Section 5.2.1 sont d'abord exécutées avec la configuration par défaut du noyau Linux, dans laquelle tous les C-states sont activés. Cette exécution sert de référence pour un fonctionnement standard du mécanisme CPUidle. Elles sont ensuite répétées avec deux configurations alternatives, chacune restreignant l'accès à certains C-states profonds. Cette approche permet d'évaluer comment la profondeur des états d'inactivité — et donc le compromis entre latence de réveil et économie d'énergie — influence les performances globales et l'efficacité énergétique des applications.

Dans la première configuration hors référence, le C-state le plus profond configurable, "C2", est désactivé. Dans la seconde configuration, les C-states "C2" et "C1" sont désactivés.

Pendant l'exécution des charges de travail synthétiques avec ces différentes configurations, la consommation énergétique ainsi que la durée des tâches sont instrumentées.

De manière générale, nous nous attendons à ce que la désactivation d'un plus grand nombre de C-states entraîne une consommation énergétique plus élevée, le processeur n'étant plus en mesure d'atteindre les états d'inactivité les plus profonds et les plus économes en énergie, mais aussi à une légère amélioration des performances en raison d'une latence réduite à la sortie de l'état d'inactivité.

5.3 Évaluation expérimentale

Dans cette Section, la méthodologie proposée est mise en œuvre. Les ressources matérielles et logicielles employées sont listées, et les résultats décrits et analysés.

5.3.1 Ressources matérielles et logicielles

La méthodologie présentée est appliquée sur un nœud de calcul HPC équipé de deux processeurs AMD EPYC 7282 dotés de 16 cœurs chacun. Chaque cœur de ces CPU supporte 3 fréquences CPU différentes : 1.5 GHz, 2.0 GHz, et 2.8 GHz.

Un système de fichiers parallèle GPFS [83] est utilisé pour les charges de travail d'E/S. Il s'agit de la version 5.1.8.0 de GPFS, configurée avec 8 volumes de 50 To chacun, pour

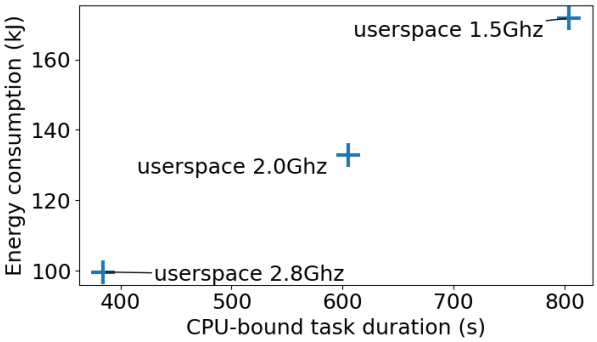


FIGURE 5.2 – Consommation énergétique et durée d'exécution d'une charge *CPU-bound* pour différentes fréquences CPU

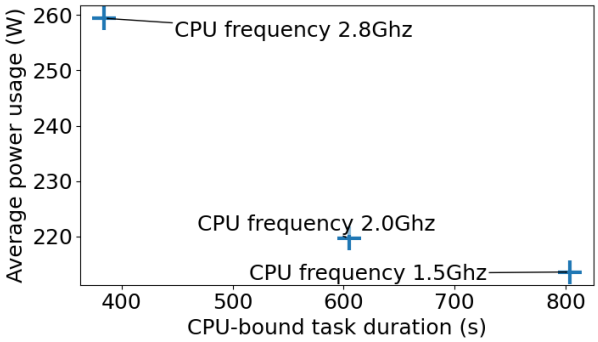


FIGURE 5.3 – Puissance moyenne et durée d'exécution d'une charge *CPU-bound* pour différentes fréquences de processeur

une capacité totale de 400 To.

Il est important de noter que GPFS dispose de son propre cache, distinct du cache de pages du noyau Linux. Ce cache a une taille fixe de 4 Go (valeur par défaut), et il est vidé entre chaque exécution de charge de travail d'E/S. Cette opération est réalisée en remplissant manuellement le page pool de GPFS avec des données sans rapport entre deux exécutions, et en utilisant l'interface */proc/sys/vm/drop_caches* du noyau Linux.

5.3.2 Effet du DVFS durant les états actifs du processeur

Comme introduit dans la Section 5.2.3, les charges de travail synthétiques sont exécutées avec, durant les états actifs du processeur, chacune des trois fréquences de processeur supportées par le processeur présenté précédemment. La consommation énergétique et les performances correspondant à ces différentes configurations sont comparées.

Afin de se concentrer exclusivement sur l'effet du DVFS, la configuration CPUidle est maintenue dans son état par défaut, avec tous les états d'inactivité (C-states) activés.

Nous nous intéressons dans un premier temps aux tâches de calcul, c'est-à-dire aux charges de travail synthétiques *CPU-bound* et *memory-bound*. Nous étudions ensuite plus en détail les charges de travail orientées E/S (*IO-bound*).

DVFS durant les P-states pour la charge de travail intensive en calculs

Dans les figures 5.2 et 5.3, les trois métriques d'évaluation pour la charge de travail synthétique limitée par le CPU sont présentées. La figure 5.2 affiche la consommation énergétique ainsi que la durée d'exécution.

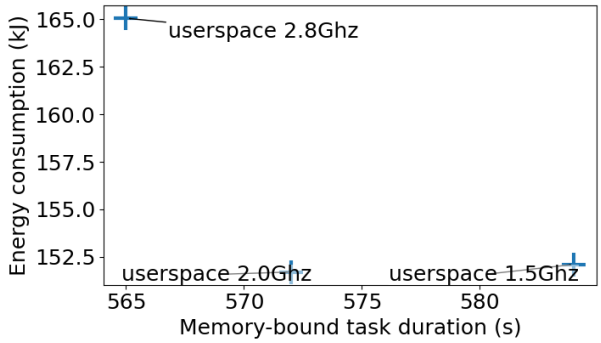


FIGURE 5.4 – Consommation énergétique et durée d'exécution d'une charge *memory-bound* pour différentes fréquences CPU

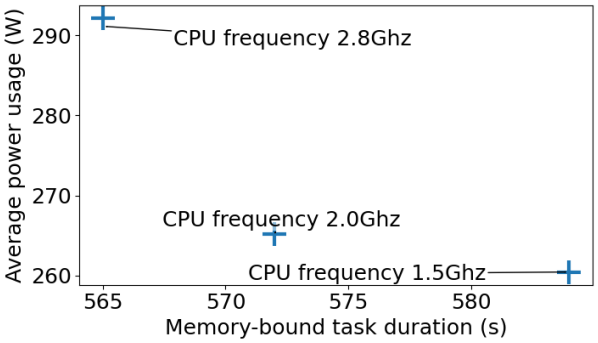


FIGURE 5.5 – Puissance moyenne et durée d'exécution d'une charge *memory-bound* pour différentes fréquences CPU

On observe que **pour cette charge de travail limitée par le CPU, une fréquence de processeur plus élevée conduit à de meilleures performances et à un coût énergétique total plus faible**. Cette observation est contre-intuitive, car on pourrait s'attendre à ce qu'une fréquence plus basse entraîne systématiquement une optimisation énergétique.

L'explication de ce comportement est donnée dans la figure 5.3. On y constate que, bien qu'une réduction de la fréquence du processeur de 2.8 GHz à 1.5 GHz diminue la consommation instantanée d'environ 15%, la durée d'exécution de la tâche double, ce qui entraîne une consommation énergétique totale plus élevée.

DVFS durant les P-states pour la charge de travail intensive en mémoire

Dans les figures 5.4 et 5.5, la charge de travail limitée par la mémoire est instrumentée de manière similaire. La figure 5.4 présente la consommation énergétique ainsi que la durée d'exécution.

Contrairement à une charge de travail synthétique limitée par le CPU, on observe que, pour une tâche limitée par la mémoire, une fréquence de processeur plus élevée améliore les performances, mais entraîne également un coût énergétique total plus élevé.

L'explication de cette observation est donnée dans la figure 5.5. On y constate que l'écart de performance entre la fréquence la plus basse et la plus élevée est inférieur à 30 secondes, ce qui correspond à un ralentissement d'environ 5 %, tandis que la consommation électrique est réduite de plus de 30 W, soit une diminution d'environ 18%. La réduction de la puissance étant ici bien plus importante que la perte de performance, il est observé que **pour cette charge de travail très consommatrice en mémoire, la réduction de**

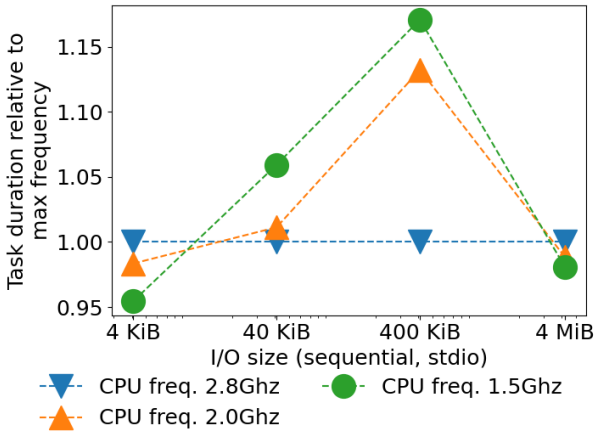


FIGURE 5.6 – Durée d'exécution d'une charge E/S bufferisées en fonction de la taille des E/S pour différentes fréquences CPU

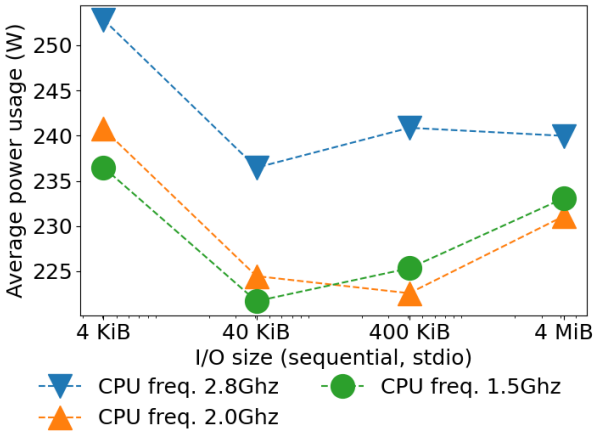


FIGURE 5.7 – Puissance moyenne d'une charge E/S bufferisées en fonction de la taille des E/S pour différentes fréquences CPU

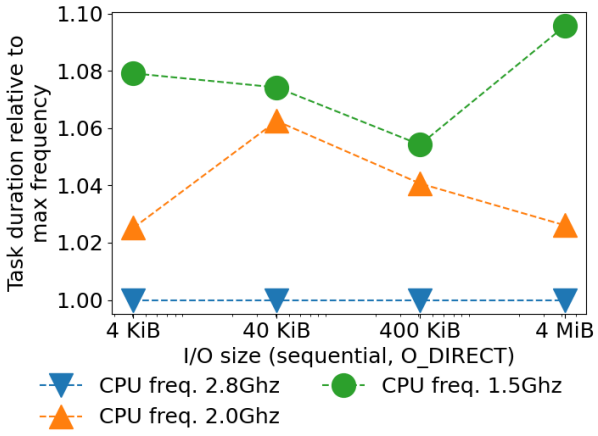


FIGURE 5.8 – Durée d'exécution d'une charge E/S directes en fonction de la taille des E/S pour différentes fréquences CPU

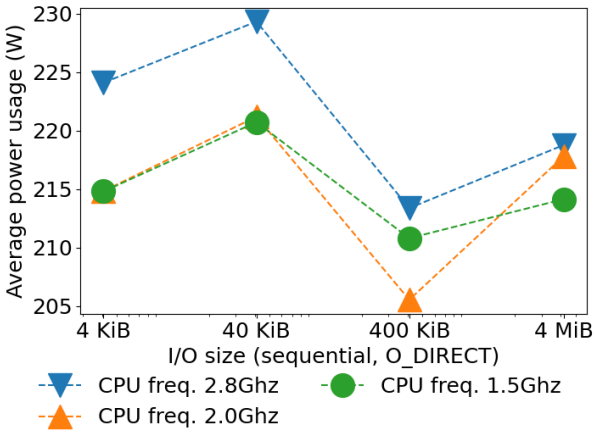


FIGURE 5.9 – Puissance moyenne d'une charge E/S directes en fonction de la taille des E/S pour différentes fréquences CPU

la fréquence du processeur permet de diminuer la consommation énergétique au prix d'une perte de performance modérée.

DVFS durant les P-states pour la charge de travail intensive en E/S

Dans les figures 5.6 à 5.9, l'effet du DVFS sur les charges de travail réalisant des E/S est évalué, à la fois pour les E/S bufferisées et les E/S directes, avec différentes tailles d'E/S.

Les figures 5.6 et 5.8 présentent la durée d'exécution de la charge d'E/S par rapport à une valeur de référence, en fonction de la taille des E/S et de la fréquence du processeur. Dans ces deux graphiques, la référence correspond à la durée obtenue avec la fréquence maximale disponible sur le processeur cible.

Pour les E/S bufferisées, bien que la réduction de puissance observée soit comprise entre 4% et 5% indépendamment de la taille des E/S (voir Fig. 5.7), l'impact sur la durée de la tâche dépend de cette taille.

La perte de performance causée par la réduction de fréquence lors des E/S bufferisées est observée comme étant minimale pour les petites tailles d'E/S ainsi que pour les E/S de 4 MiB, mais elle devient significative (jusqu'à 15 % de ralentissement) pour les autres tailles d'E/S.

Cela s'explique par le fait que le *page pool* de GPFS est spécialement conçu pour optimiser les petites E/S séquentielles [41], et que 4 Mio correspond à la taille de bloc de notre installation GPFS. Ainsi, le processeur passe plus de temps à traiter les données et moins de temps en attente pour ces deux tailles d'E/S, comparativement aux autres.

Pour les E/S directes, l'observation est plus simple : une fréquence de processeur plus faible entraîne systématiquement une baisse de performance (jusqu'à 15 % de perte), mais également une réduction de la consommation énergétique (jusqu'à 4.5%).

5.3.3 Effet du DVFS en fonction des C-states

Comme introduit dans la Section 5.2.3, les charges de travail synthétiques sont exécutées avec différentes configurations de *CPUidle* afin d'évaluer l'impact des états d'inactivité (**C-states**) sur les performances et la consommation énergétique. La fréquence du processeur pour les états actifs (**P-states**, via *CPUfreq*) est maintenue à sa valeur maximale, soit 2.8 GHz, afin d'isoler l'effet des transitions entre C-states sans interférence avec la modulation de fréquence.

Cette approche permet de quantifier précisément le compromis entre profondeur des C-states, latence de réveil et économie d'énergie, et ainsi de déterminer dans quelle mesure la gestion de l'inactivité influence l'efficacité globale du calcul.

Nous nous concentrons d'abord sur les tâches de calcul, à savoir les charges de travail synthétiques intenses en calcul et en mémoire. Ensuite, nous étudions la charge de travail réalisant des E/S.

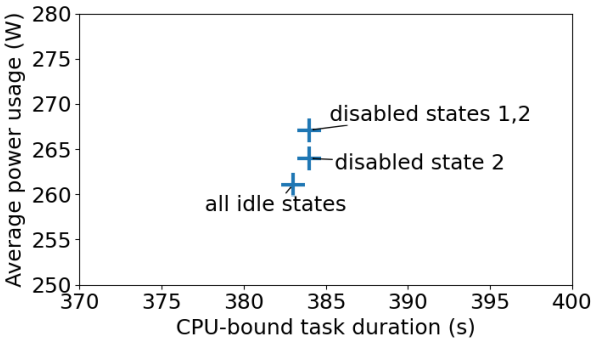


FIGURE 5.10 – Puissance moyenne et durée d'exécution d'une charge *CPU-bound* pour différentes configurations des états inactifs (C-states)

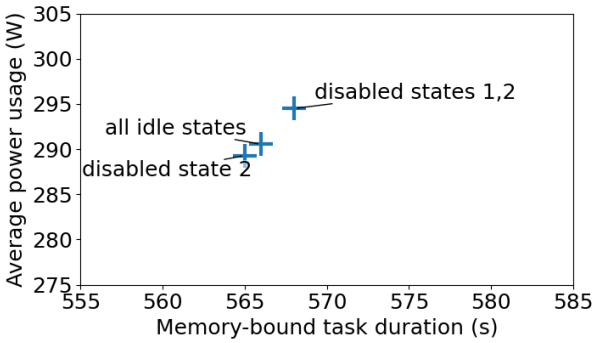


FIGURE 5.11 – Puissance moyenne et durée d'exécution d'une charge *memory-bound* pour différentes configurations des états inactifs (C-states)

DVFS en fonction des C-states pour les charges de travail intenses en calcul ou en mémoire

Les figures 5.10 et 5.11 présentent l'effet de quatre configurations de C-states sur la consommation énergétique et la durée d'exécution, respectivement pour les applications synthétiques limitées par le CPU et par la mémoire.

On observe que **sur nos charges de travail limitées par le CPU ou la mémoire, la désactivation des C-states entraîne un très faible effet sur la performance (<1%) ainsi que sur la puissance consommée (<2%)**. Cet effet très limité s'explique par le fait que ces charges de travail maintiennent le processeur actif la majeure partie du temps, l'empêchant ainsi d'entrer en état d'inactivité.

DVFS durant les C-states pour la charge de travail intensive en E/S

Les figures 5.12 à 5.15 présentent l'effet de quatre configurations de C-states sur la puissance consommée et la durée d'exécution pour les applications synthétiques limitées par les E/S, d'abord avec des E/S bufferisées, puis avec des E/S directes.

On observe dans les figures 5.13 et 5.15 que **la désactivation des C-states entraîne une augmentation de la puissance moyenne consommée**. Ce comportement s'explique par le fait que le processeur reste continuellement dans un état actif, sans possibilité d'entrer dans des modes de veille plus profonds pendant les périodes d'attente d'E/S. L'absence de transitions vers des C-states profonds empêche donc les économies d'énergie habituellement réalisées lorsque le CPU est inactif.

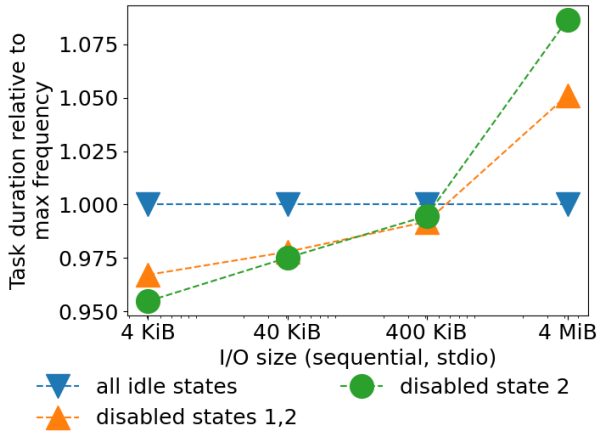


FIGURE 5.12 – Durée d'exécution d'une charge E/S bufferisées en fonction de la taille des E/S pour différentes configurations des états inactifs (C-states)

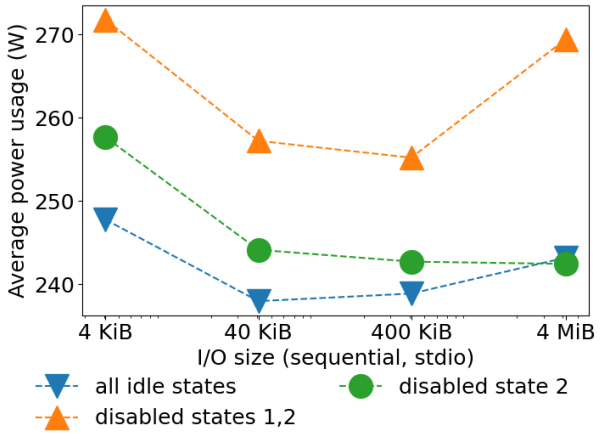


FIGURE 5.13 – Puissance moyenne d'une charge E/S bufferisées en fonction de la taille des E/S pour différentes configurations des états inactifs (C-states)

Dans le même temps, les figures 5.12 et 5.14 montrent qu'une telle désactivation peut entraîner **une légère amélioration des performances**. Cet effet s'explique par la suppression de la latence de sortie des C-states : le processeur reste immédiatement disponible pour traiter la reprise des opérations d'E/S. L'amélioration reste toutefois modérée, car les charges de travail considérées demeurent principalement limitées par la vitesse du stockage, et non par le temps de réaction du CPU.

Ainsi, le compromis observé entre consommation et performance illustre le rôle des C-states dans la gestion fine de l'énergie : plus la charge de travail présente de phases d'attente (comme les E/S), plus l'activation des C-states profonds permet de réduire la consommation au prix d'un léger surcoût temporel lié aux transitions. À l'inverse, leur désactivation favorise une réactivité légèrement accrue mais se traduit par une efficacité énergétique dégradée.

5.4 Proposition de consignes d'utilisation du DVFS pendant les phases d'E/S

L'analyse conjointe des résultats relatifs aux P-states et aux C-states met en évidence que l'impact du DVFS dépend fortement de la nature de la charge de travail, avec des comportements contrastés entre charges de calcul et charges d'E/S.

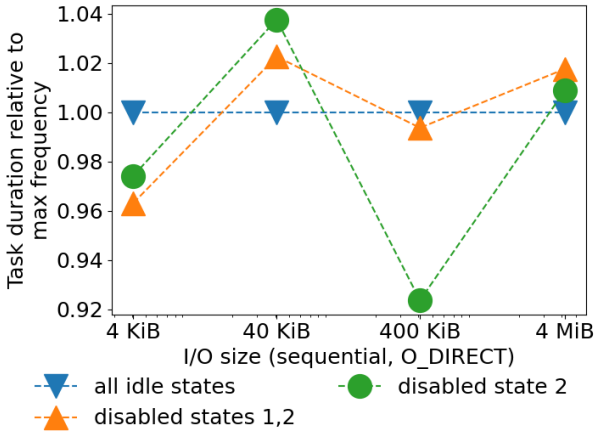


FIGURE 5.14 – Durée d'exécution d'une charge E/S directes en fonction de la taille des E/S pour différentes configurations des états inactifs (C-states)

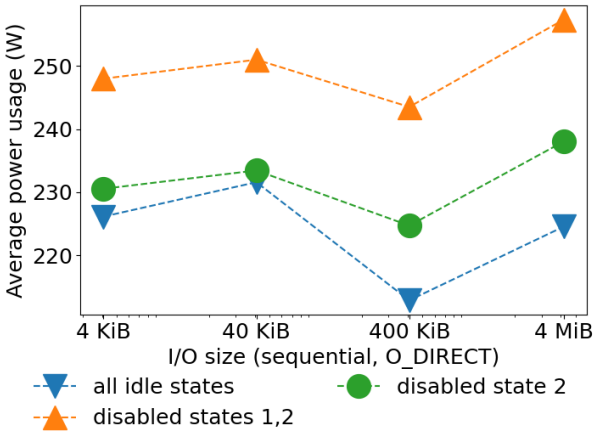


FIGURE 5.15 – Puissance moyenne d'une charge E/S directes en fonction de la taille des E/S pour différentes configurations des états inactifs (C-states)

Pour les charges limitées par le CPU (Fig. 5.2 et 5.3), la fréquence maximale (2,8 GHz) reste la configuration la plus efficace, combinant performance et consommation énergétique optimisée grâce à des temps d'exécution plus courts. Pour les charges limitées par la mémoire (Fig. 5.4 et 5.5), une réduction modérée de fréquence peut réduire la consommation énergétique avec une dégradation mineure des performances. **Ainsi, nous suggérons de maintenir la fréquence maximale pour les charges *CPU-bound* et d'envisager une réduction modérée (par exemple à 2,0 GHz) pour les charges *memory-bound*, lorsque la minimisation de la consommation prime sur l'optimisation stricte du temps d'exécution.**

En revanche, les charges orientées E/S révèlent des tendances plus nuancées. Les figures 5.6 à 5.9 montrent qu'avec des E/S bufferisées, certaines tailles d'E/S (notamment 4 Mio et les petits blocs séquentiels) tolèrent une baisse de fréquence CPU sans dégradation significative des performances, tout en réduisant la puissance moyenne consommée. Pour les E/S directes, une réduction de fréquence provoque généralement un ralentissement (jusqu'à 15

Ces observations soulignent que l'effet du DVFS sur les charges d'E/S ne suit pas une relation monotone simple entre fréquence, performance et énergie. L'efficacité d'une réduction de fréquence dépend à la fois du mode d'accès (bufferisé ou direct), de la taille des transferts et de la proportion de temps passée en attente d'E/S. Autrement dit, le potentiel de gain énergétique du DVFS sur les applications E/S-bound reste fortement

dépendant du profil d'accès et de la configuration du système de fichiers parallèle (PFS).

Ainsi, plutôt que de chercher une politique statique de réduction de fréquence pendant les E/S, il apparaît nécessaire d'adopter une approche adaptative, capable d'ajuster la fréquence CPU en fonction des motifs d'E/S. Une telle stratégie permettrait de peu affecter la performance tout en réalisant des économies d'énergie lorsque les E/S dominent le temps d'exécution. Ces résultats invitent plus largement à repenser l'usage du DVFS dans les environnements HPC : contrairement à l'approche classique se concentrant uniquement sur l'activité du processeur, la pleine efficacité énergétique ne pourrait en fait être atteinte qu'en prenant en compte les motifs d'accès aux données et les caractéristiques des systèmes de stockage.

S'agissant des C-states, les figures 5.10 à 5.15 confirment qu'ils ont peu d'impact pour les charges *CPU-bound* et *memory-bound*, où le processeur reste actif en continu. En revanche, pour les charges E/S-bound, leur désactivation induit une hausse sensible de la consommation électrique, liée à l'incapacité du processeur à entrer dans des états de repos lors des phases d'attente d'E/S. L'amélioration de performance observée reste marginale et ne justifie cette configuration que dans des scénarios où la latence est critique et la contrainte énergétique secondaire. **Ainsi, nous suggérons de maintenir les C-states activés pour toutes les charges de travail, à l'exception de cas spécifiques où une performance maximale est impérative et où une augmentation parfois significative de la consommation énergétique (jusqu'à 15% pour des E/S directes et séquentielles de 40 KiB) est acceptable.**

5.5 Conclusion

Dans ce chapitre, nous avons évalué l'impact du DVFS sur la performance et la consommation énergétique d'applications HPC. Nos résultats montrent que l'efficacité du DVFS dépend fortement du type de charge de travail. Pour les charges limitées par le CPU, la fréquence maximale offre le meilleur compromis entre performance et consommation totale, tandis que pour les charges limitées par la mémoire, une réduction de la fréquence peut réduire significativement la puissance moyenne sans perte de performance notable.

Pour les charges orientées E/S, les effets du DVFS sont plus variables, en fonction de la taille d'E/S et du type d'E/S : certaines tailles d'E/S bufferisées peuvent profiter d'une baisse de la fréquence CPU, mais ce n'est pas le cas pour les E/S directes, avec des pertes de performance pouvant atteindre 15%. Concernant les C-states, leur désactivation

n'apporte aucun gain pour les charges de calcul et augmente la consommation lors des phases d'E/S, confirmant l'intérêt de les maintenir activés par défaut.

Ainsi, cette étude met en évidence des opportunités d'optimisation, et montre que le DVFS ne pouvait être appliqué efficacement qu'en tenant compte du profil de la charge et du comportement d'E/S. Cependant, notre étude présente des limites importantes. Nous avons montré que le DVFS pouvait réduire la consommation énergétique dans certaines conditions, sans toutefois proposer un modèle exhaustif permettant de prédire son effet pour tout motif d'E/S, toute application MPI ou tout schéma de communication. Des travaux futurs pourraient donc viser à développer un modèle prédictif complet intégrant les interactions entre DVFS et comportement d'E/S, afin de permettre une adaptation automatique et généralisable du DVFS aux charges de travail HPC réelles.

Chapitre 6

Conclusion

6.1 Résumé des contributions

Le calcul haute performance (HPC) fait aujourd'hui face à une double évolution : la croissance rapide des volumes de données et des capacités de calcul, et l'augmentation de la consommation énergétique des supercalculateurs. Dans ce contexte, cette thèse s'intéresse à deux leviers susceptibles d'influer sur la performance et la consommation énergétique des applications HPC durant leurs opérations d'entrée/sortie (E/S) : (1) les fonctions d'indices d'usage de données, qui peuvent affecter le placement des données dans les systèmes de fichiers parallèles, et (2) le DVFS, qui permet d'ajuster dynamiquement la fréquence et la tension des processeurs. Afin d'actionner ces leviers de manière informée et efficace, nous avons présenté trois contributions principales : une approche de modélisation et de prédiction des E/S, une étude expérimentale des indices d'usage de données, et une évaluation de l'effet du DVFS sur les performances et la consommation énergétique des E/S.

(1) GrIOt — Modélisation et prédiction d'E/S par graphes de piles d'appels. La première contribution introduit **GrIOt**, une approche de modélisation et de prédiction des E/S d'applications HPC reposant sur des graphes de piles d'appels. GrIOt se distingue des modèles à base de grammaires, tels qu'Omnisc'IO, en proposant une représentation plus compacte, générique et adaptable, apte à modéliser aussi bien les comportements d'E/S déterministes que non déterministes. Les résultats expérimentaux obtenus sur cinq applications représentatives montrent que GrIOt atteint une précision de prédiction comparable à l'état de l'art, tout en réduisant fortement les coûts de modélisation et d'exécution (jusqu'à 120x plus rapide et de 10x à 15x moins coûteux en mémoire). Ces performances confirment la pertinence de la modélisation par graphes pour capturer la structure et la dynamique des motifs d'E/S. GrIOt constitue ainsi une première étape vers une exploitation prédictive des comportements d'E/S, ouvrant la possibilité d'actionner

automatiquement des leviers d'optimisation tels que les indices d'usage de données ou le DVFS.

(2) Analyse expérimentale des indices d'usage de données sur Lustre et GPFS. La deuxième contribution propose une **méthodologie expérimentale systématique** pour évaluer l'efficacité réelle des indices d'usage de données sur deux systèmes de fichiers parallèles majeurs : Lustre et GPFS. Les résultats obtenus montrent une forte dépendance du comportement de ces indices à l'implémentation du système de fichiers sous-jacent. Si les indices de motif d'accès (`SEQUENTIAL`, `RANDOM`) ont un impact faible, les indices de gestion de cache (`DONT_NEED`, `WILL_NEED`) se révèlent pertinents lorsqu'ils sont effectivement pris en compte par le PFS, comme nous l'avons vu notamment sur Lustre. Nos observations mettent en évidence un manque d'uniformité entre systèmes et un écart entre ce qui est attendu par les développeurs applicatifs lorsqu'ils utilisent ces indices et la réalité des implémentations. Cette contribution fournit ainsi une meilleure compréhension du fonctionnement réel des indices d'usage de données et propose des recommandations pratiques pour leur utilisation dans les applications HPC. Elle établit également les fondations d'une exploitation future de ces indices en conjonction avec des modèles prédictifs tels que GrIOt.

(3) Étude de l'impact du DVFS sur la performance et la consommation énergétique des E/S. La troisième contribution explore l'effet du DVFS sur les performances et la consommation énergétique des charges de travail HPC. Les résultats montrent que le DVFS peut être bénéfique pour certaines charges limitées par la mémoire ou les E/S, en réduisant la consommation énergétique sans dégrader les performances, tandis qu'il est moins efficace pour les charges de calcul intensives. L'analyse révèle également que l'impact du DVFS dépend du type d'E/S (bufferisées ou directes) et de leur taille, mettant en évidence la nécessité d'un pilotage du DVFS qui prenne en compte la nature des E/S. Cette étude constitue une première étape vers un modèle prédictif global capable d'adapter dynamiquement la fréquence du processeur en fonction du profil d'E/S, en vue d'optimiser conjointement performance et efficacité énergétique.

Dans l'ensemble, ces trois contributions apportent une compréhension plus fine des interactions entre les applications, le stockage et l'énergie dans les environnements HPC. Elles démontrent la complémentarité des approches de modélisation, de caractérisation expérimentale et d'analyse énergétique, et posent les bases d'une intégration unifiée des leviers identifiés — indices d'usage de données et DVFS — au sein d'un cadre prédictif

commun pour l'optimisation automatique des E/S.

6.2 Perspective

6.2.1 Court terme

La première perspective de recherche concerne l'évolution du modèle **GrIOt**. Bien que celui-ci permette la réutilisation et la fédération de modèles, fédérer des modèles en fusionnant les tables et graphes un par un ne prend pas pleinement en compte l'hétérogénéité interne des applications parallèles, notamment entre les rangs MPI. Une extension du modèle pour représenter explicitement cette diversité permettrait d'améliorer la précision et la robustesse des prédictions. De plus, l'étude du compromis entre granularité de modélisation et coût de calcul qui a été présentée pourrait être poursuivie afin d'adapter dynamiquement la profondeur d'instrumentation de la pile d'appels.

La seconde perspective concerne l'**étude des indices d'usage de données**. Notre analyse a porté sur Lustre et GPFS, deux systèmes de fichiers parallèles représentatifs mais non exhaustifs. Il serait pertinent d'étendre cette méthodologie à un plus large ensemble de PFS, tels que BeeGFS, CephFS ou HDFS, afin de mieux cerner les différences d'interprétation et de comportement des indices dans des contextes variés. Une telle extension permettrait également d'évaluer la stabilité des résultats obtenus et de renforcer la portée de nos recommandations pratiques.

Enfin, la troisième contribution, dédiée à l'évaluation du **DVFS** pendant les E/S, pourrait être approfondie selon deux axes. D'une part, l'étude pourrait être reproduite sur un plus grand nombre de plateformes matérielles et de modèles de processeurs afin de valider la généricité des conclusions obtenues. D'autre part, le développement d'un modèle analytique décrivant la relation entre motif d'E/S, fréquence CPU, performance et consommation d'énergie constituerait une avancée importante. Un tel modèle permettrait de prédire le gain énergétique potentiel avant application du DVFS et d'en guider l'activation en fonction des caractéristiques d'exécution des applications.

De manière plus transversale, une limite commune aux trois contributions réside dans leur traitement séparé : si chacune explore un levier spécifique, leur combinaison n'a pas encore été mise en œuvre dans un outil unifié. L'une des priorités à court terme serait donc de concevoir une infrastructure expérimentale commune permettant d'évaluer les interactions entre modèles de prédiction, indices d'usage et DVFS, et d'en mesurer les

effets conjoints sur les performances et l'efficacité énergétique.

6.2.2 Long terme

À plus long terme, les perspectives de ce travail visent l'**intégration de ces leviers dans un cadre prédictif et adaptatif unique**. Une première direction consiste à faire de GrIOt le cœur d'un système capable d'interagir dynamiquement avec le système d'exploitation et le système de fichiers parallèle. Les informations issues de la modélisation prédictive pourraient être utilisées pour actionner automatiquement les fonctions d'indices d'usage de données ou ajuster la fréquence CPU via le DVFS, en fonction des comportements observés en temps réel.

Une telle approche impliquerait la mise en place d'une **boucle de rétroaction continue** entre l'observation, la prédiction et l'action : le modèle GrIOt prédirait les motifs d'E/S, les indices d'usage seraient appliqués pour optimiser le placement et le cache, et le DVFS serait ajusté pour minimiser la consommation énergétique sans perte de performance.

Enfin, à une échelle encore plus large, ces travaux pourraient s'inscrire dans la conception de **systèmes de stockage auto-adaptatifs**, capables d'ajuster leur comportement en fonction du profil d'application, de la charge du système et des objectifs énergétiques globaux du centre de calcul. Ces perspectives s'inscrivent dans la transition vers un calcul haute performance plus économe et responsable, où la performance et l'efficacité énergétique constituent des objectifs indissociables de la conception des infrastructures et des logiciels.

6.3 Réflexions finales

Les travaux présentés dans cette thèse ont exploré plusieurs leviers complémentaires pour améliorer la performance et l'efficacité énergétique des systèmes de calcul haute performance. En s'appuyant sur une démarche mêlant modélisation, analyse expérimentale et évaluation énergétique, ils ont permis de mieux comprendre les interactions complexes entre applications, systèmes de fichiers et matériel.

Ces recherches montrent qu'une approche informée et prédictive du comportement des E/S peut constituer un levier puissant pour améliorer l'efficacité globale des supercalculateurs, à la fois sur le plan des performances et de la consommation énergétique. GrIOt

a démontré la faisabilité d'une modélisation légère et adaptable du comportement d'E/S ; l'étude des indices d'usage de données a mis en évidence la variabilité de leur interprétation dans les systèmes de fichiers parallèles ; et l'analyse du DVFS a révélé des marges d'optimisation énergétiques encore peu exploitées. Ensemble, ces travaux soulignent la nécessité d'une co-conception entre modèles, logiciels et matériel pour atteindre une efficacité réellement globale.

Les perspectives à long terme de ces travaux s'inscrivent ainsi dans la conception de systèmes HPC capables de s'auto-ajuster en fonction du profil des applications et des contraintes énergétiques. Une telle approche s'aligne avec les orientations actuelles vers des architectures exascales à haute efficacité énergétique, où la performance et la sobriété doivent être prises en compte conjointement dès la conception des logiciels et des infrastructures.

En définitive, cette thèse vise à contribuer, à son échelle, à la transition vers un calcul haute performance plus intelligent, plus économe et plus durable, où la connaissance fine des comportements d'E/S devient un élément clé de l'efficacité énergétique globale des systèmes.

BIBLIOGRAPHIE

[1] Lydia AIT-OUCHEGGOU et al., « QM-ARC : QoS-aware Multi-tier Adaptive Cache Replacement Strategy », in : *Future Gener. Comput. Syst.* 163 (fév. 2025), p. 107548, ISSN : 0167-739X, DOI : 10.1016/j.future.2024.107548.

[2] Alex AIZMAN, Gavin MALTBY et Thomas BREUEL, « High Performance I/O For Large Scale Deep Learning », in : *2019 IEEE International Conference on Big Data (Big Data)*, IEEE, p. 09-12, DOI : 10.1109/BigData47090.2019.9005703.

[3] AXBOE, *fio*, [Online ; accessed 23. Jan. 2025], jan. 2025, URL : https://github.com/axboe/fio.

[4] Alain AZAGURY et al., « Towards an object store », in : *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings.* IEEE, 2003, p. 165-176.

[5] Théophile BASTIAN, Stephen KELL et Francesco ZAPPA NARDELLI, « Reliable and fast DWARF-based stack unwinding », in : *Proc. ACM Program. Lang.* 3.*OOPSLA* (oct. 2019), p. 1-24, DOI : 10.1145/3360572.

[6] Anton BELOGLAZOV et al., « A Taxonomy and Survey of Energy-Efficient Data Centers and Cloud Computing Systems », in : *Advances in Computers*, t. 82, Walthm, MA, USA : Elsevier, jan. 2011, p. 47-111, DOI : 10.1016/B978-0-12-385512-1.00003-7.

[7] Jean Luca BEZ, Suren BYNA et Shadi IBRAHIM, « I/O Access Patterns in HPC Applications : A 360-Degree Survey », in : *ACM Comput. Surv.* 56.*2* (sept. 2023), p. 1-41, ISSN : 0360-0300, DOI : 10.1145/3611007.

[8] Francieli Zanon BOITO et al., « A Deep Look Into the Temporal I/O Behavior of HPC Applications », in : (2025).

[9] Francieli Zanon BOITO et al., « On server-side file access pattern matching », in : *2019 International Conference on High Performance Computing & Simulation (HPCS)*, IEEE, juill. 2019, p. 217-224, DOI : 10.1109/HPCS48598.2019.9188092.

[10] Jalil Boukhobza et Pierre Olivier, *Flash Memory Integration : Performance and Energy Issues*, Elsevier, 2017.

[11] Jalil Boukhobza, Pierre Olivier et Stéphane Rubini, « A Cache Management Strategy to Replace Wear Leveling Techniques for Embedded Flash Memory », in : *arXiv* (sept. 2012), doi : 10.48550/arXiv.1209.3099, eprint : 1209.3099.

[12] Jalil Boukhobza et al., « A Survey on Flash-Memory Storage Systems : A Host-Side Perspective », in : *ACM Trans. Storage* (mars 2025), Just Accepted, issn : 1553-3077, doi : 10.1145/3723167, url : https://doi.org/10.1145/3723167.

[13] Jalil Boukhobza et al., « Emerging NVM : A Survey on Architectural Integration and Research Challenges », in : *ACM Trans. Des. Autom. Electron. Syst.* 23.*2* (nov. 2017), p. 1-32, issn : 1084-4309, doi : 10.1145/3131848.

[14] M Scot Breitenfeld et al., « DAOS for extreme-scale systems in scientific applications », in : *arXiv preprint arXiv :1712.00423* (2017).

[15] Enrico Calore et al., « Evaluation of DVFS techniques on modern HPC processors and accelerators for energy-aware applications », in : *Concurrency and Computation Practice and Experience* 29.*2* (mars 2017), issn : 1532-0634, doi : 10.1002/cpe.4143.

[16] Philip Carns et al., « 24/7 Characterization of petascale I/O workloads », in : *Proceedings - IEEE International Conference on Cluster Computing, ICCC* (oct. 2009), p. 1-10, issn : 1552-5244, doi : 10.1109/CLUSTR.2009.5289150.

[17] Philip Carns et al., « Understanding and Improving Computational Science Storage Access through Continuous Characterization », in : *ACM Trans. Storage* 7.*3* (oct. 2011), p. 1-26, issn : 1553-3077, doi : 10.1145/2027066.2027068.

[18] Yong Chen et al., « Hiding I/O latency with pre-execution prefetching for parallel applications », in : *SC '08 : Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, IEEE, nov. 2008, p. 1-10, doi : 10.1109/SC.2008.5213209.

[19] Ryan Child et Philip A. Wilsey, « Using DVFS to optimize time Warp simulations », in : *Proceedings of the 2012 Winter Simulation Conference (WSC)*, IEEE, p. 09-12, doi : 10.1109/WSC.2012.6465146.

[20] M. Chrobak et J. Noga, « LRU Is Better than FIFO », in : *Algorithmica* 23.*2* (fév. 1999), p. 180-185, issn : 1432-0541, doi : 10.1007/PL00009255.

[21] Ryan Cochran et al., « Pack & Cap : adaptive DVFS and thread packing under power caps », in : *ACM Conferences*, New York, NY, USA : Association for Computing Machinery, déc. 2011, p. 175-185, doi : 10.1145/2155620.2155641.

[22] Alexis Colin, François Trahay et Denis Conan, « PYTHIA : an oracle to guide runtime system decisions », in : *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, sept. 2022, p. 106-116, doi : 10.1109/CLUSTER51413.2022.00025.

[23] Miyuru Dayarathna, Yonggang Wen et Rui Fan, « Data Center Energy Consumption Modeling : A Survey », in : *IEEE Commun. Surv. Tutorials* 18.*1* (sept. 2015), p. 732-794, doi : 10.1109/COMST.2015.2481183.

[24] Peter Desnoyers, « Analytic Models of SSD Write Performance », in : *ACM Trans. Storage* 10.*2* (mars 2014), p. 1-25, issn : 1553-3077, doi : 10.1145/2577384.

[25] Hariharan Devarajan et al., « DLIO : A Data-Centric Benchmark for Scientific Deep Learning Applications », in : *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, IEEE, p. 10-13, doi : 10.1109/CCGrid51090.2021.00018.

[26] Hariharan Devarajan et al., « Vidya : Performing Code-Block I/O Characterization for Data Access Optimization », in : *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*, IEEE, déc. 2018, p. 255-264, doi : 10.1109/HiPC.2018.00036.

[27] Matthieu Dorier et al., « Omnisc'IO : A Grammar-Based Approach to Spatial and Temporal I/O Patterns Prediction », in : *SC '14 : Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, nov. 2014, p. 623-634, doi : 10.1109/SC.2014.56.

[28] Matthieu Dorier et al., « Using Formal Grammars to Predict I/O Behaviors in HPC : The Omnisc'IO Approach », in : *IEEE Trans. Parallel Distrib. Syst.* 27.*8* (oct. 2015), p. 2435-2449, doi : 10.1109/TPDS.2015.2485980.

[29] Nikoli Dryden et al., « Clairvoyant prefetching for distributed machine learning I/O », in : *SC '21 : Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA : Association for Computing Machinery, nov. 2021, p. 1-15, isbn : 978-1-45038442-1, doi : 10.1145/3458817.3476181.

[30] James Elliott et al., « Combining Partial Redundancy and Checkpointing for HPC », in : *2012 IEEE 32nd International Conference on Distributed Computing Systems*, IEEE, p. 18-21, DOI : 10.1109/ICDCS.2012.56.

[31] M. Etinski et al., « Understanding the future of energy-performance trade-off via DVFS in HPC environments », in : *J. Parallel Distrib. Comput.* 72.*4* (avr. 2012), p. 579-590, ISSN : 0743-7315, DOI : 10.1016/j.jpdc.2012.01.006.

[32] Ohad Eytan et al., *It's Time to Revisit {LRU} vs. {FIFO}*, [Online ; accessed 18. Sep. 2025], 2020, URL : https://www.usenix.org/conference/hotstorage20/presentation/eytan.

[33] Facebook, *RocksDB | A persistent key-value store*, URL : https://rocksdb.org.

[34] Michael Factor et al., « Object storage : The future building block for storage systems », in : *2005 IEEE International Symposium on Mass Storage Systems and Technology*, IEEE, 2005, p. 119-123.

[35] Shaleen Garg et al., « CrossPrefetch : Accelerating I/O Prefetching for Modern Storage », in : ASPLOS '24, New York, NY, USA : Association for Computing Machinery, 2024, ISBN : 9798400703720, DOI : 10.1145/3617232.3624872, URL : https://doi.org/10.1145/3617232.3624872.

[36] Vladimir Gligorov, « Real-time data analysis at the LHC : present and future », in : *NIPS 2014 Workshop on High-energy Physics and Machine Learning*, PMLR, août 2015, p. 1-18, URL : https://proceedings.mlr.press/v42/glig14.

[37] Zvika Guz et al., « NVMe-over-fabrics performance characterization and the path to low-overhead flash disaggregation », in : *ACM Conferences*, New York, NY, USA : Association for Computing Machinery, mai 2017, p. 1-9, DOI : 10.1145/3078468.3078483.

[38] Frank T Hady et al., « Platform storage performance with 3D XPoint technology », in : *Proceedings of the IEEE* 105.*9* (2017), p. 1822-1833.

[39] Jun He et al., « I/O acceleration with pattern detection », in : *HPDC '13 : Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, New York, NY, USA : Association for Computing Machinery, juin 2013, p. 25-36, ISBN : 978-1-45031910-2, DOI : 10.1145/2462902.2462909.

[40] *Home - BeeGFS - The Leading Parallel Cluster File System*, [Online ; accessed 6. Nov. 2025], oct. 2025, URL : https://www.beegfs.io/c.

[41] *IBM Storage Scale*, [Online ; accessed 12. Nov. 2024], jan. 2024, URL : https://www.ibm.com/docs/en/storage-scale/5.1.9?topic=configuring-parameters-performance-tuning-optimization.

[42] *IOR*, [Online ; accessed 11. Jul. 2023], juill. 2023, URL : https://github.com/hpc/ior.

[43] Joseph IZRAELEVITZ et al., « Basic performance measurements of the intel optane DC persistent memory module », in : *arXiv preprint arXiv :1903.05714* (2019).

[44] Song JIANG et Xiaodong ZHANG, « LIRS : an efficient low inter-reference recency set replacement policy to improve buffer cache performance », in : *ACM SIGMETRICS Performance Evaluation Review*, t. 30, 1, New York, NY, USA : Association for Computing Machinery, juin 2002, p. 31-42, DOI : 10.1145/511334.511340.

[45] Hui JIN et al., « Checkpointing Orchestration : Toward a Scalable HPC Fault-Tolerant Environment », in : *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, IEEE, p. 13-16, DOI : 10.1109/CCGrid.2012.61.

[46] Adrian KHELILI, Sophie ROBERT et Soraya ZERTAL, « FiLiP : A File Lifecycle-based Profiler for hierarchical storage. », in : *Infocommunications Journal* 14.*4* (2022).

[47] Toshiya KOMODA et al., « Power capping of CPU-GPU heterogeneous systems through coordinating DVFS and task mapping », in : *2013 IEEE 31st International Conference on Computer Design (ICCD)*, IEEE, p. 06-09, ISBN : 978-1-4799-2987-0, DOI : 10.1109/ICCD.2013.6657064.

[48] Abhishek Vijaya KUMAR et Muthian SIVATHANU, *Quiver : An Informed Storage Cache for Deep Learning*, [Online ; accessed 7. Oct. 2025], 2020, URL : https://www.usenix.org/conference/fast20/presentation/kumar.

[49] Arezki LAGA et al., « Lynx : a learning linux prefetching mechanism for SSD performance model », in : *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, IEEE, août 2016, p. 1-6, DOI : 10.1109/NVMSA.2016.7547186.

[50] Sylvain LAIZET et Ning LI, « Incompact3d : A powerful tool to tackle turbulence problems with up to O(105) computational cores », in : *Int. J. Numer. Methods Fluids* 67.*11* (déc. 2011), p. 1735-1757, ISSN : 0271-2091, DOI : 10.1002/fld.2480.

[51] Vincent Lannurien et al., « HeROsim : An Allocation and Scheduling Simulator for Evaluating Serverless Orchestration Policies », in : *IEEE Internet Comput.* 28.*6* (déc. 2024), p. 8-16, DOI : 10.1109/MIC.2024.3511332.

[52] D. Lee et al., « LRFU : A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies », in : *IEEE Trans. Comput.* 50.*12* (déc. 2001), p. 1352-1361, ISSN : 0018-9340, DOI : 10.1109/TC.2001.970573.

[53] Dusol Lee et al., « P2Cache : An Application-Directed Page Cache for Improving Performance of Data-Intensive Applications », in : HotStorage '23, New York, NY, USA : Association for Computing Machinery, 2023, ISBN : 9798400702242, DOI : 10.1145/3599691.3603408, URL : https://doi.org/10.1145/3599691.3603408.

[54] Eunji Lee et al., « Reducing Journaling Harm on Virtualized I/O Systems », in : SYSTOR '16, New York, NY, USA : Association for Computing Machinery, 2016, ISBN : 9781450343817, DOI : 10.1145/2928275.2928289, URL : https://doi.org/10.1145/2928275.2928289.

[55] Jun Li et al., « Pattern-Based Prefetching with Adaptive Cache Management Inside of Solid-State Drives », in : *ACM Trans. Storage* 18.*1* (jan. 2022), p. 1-25, ISSN : 1553-3077, DOI : 10.1145/3474393.

[56] Zhen Liang et al., « DAOS : A Scale-Out High Performance Storage Stack for Storage Class Memory », in : *Supercomputing Frontiers*, Cham, Switzerland : Springer, juin 2020, p. 40-54, ISBN : 978-3-030-48842-0, DOI : 10.1007/978-3-030-48842-0_3.

[57] *Lustre∗ Software Release 2.x*, [Online ; accessed 12. Nov. 2024], nov. 2024, URL : https://doc.lustre.org/lustre_manual.xhtml#idm140494033959744.

[58] Jakob Luttgau et al., « Survey of Storage Systems for High-Performance Computing », in : *Supercomput. Front. Innov. : Int. J.* 5.*1* (mars 2018), p. 31-58, ISSN : 2409-6008, DOI : 10.14529/jsfi180103.

[59] Jakob Lüttgau et al., « Survey of storage systems for high-performance computing », in : *Supercomputing Frontiers and Innovations* 5.*1* (2018).

[60] Hocine Mahni et al., « Multicriteria File-Level Placement Policy for HPC Storage », in : *ACM Conferences*, New York, NY, USA : Association for Computing Machinery, mars 2025, p. 1399-1406, DOI : 10.1145/3672608.3707969.

[61] Hector Marco-Gisbert et Ismael Ripoll Ripoll, « Address Space Layout Randomization Next Generation », in : *Appl. Sci.* 9.*14* (juill. 2019), p. 2928, issn : 2076-3417, doi : 10.3390/app9142928.

[62] John D. McCalpin, « Memory Bandwidth and Machine Balance in Current High Performance Computers », in : *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (déc. 1995), p. 19-25.

[63] Nimrod Megiddo et Dharmendra S. Modha, « ARC : a self-tuning, low overhead replacement cache », in : *Guide Proceedings*, USENIX Association, mars 2003, p. 9, doi : 10.5555/1973355.1973364.

[64] Mike Mesnier, Gregory R Ganger et Erik Riedel, « Object-based storage », in : *IEEE Communications Magazine* 41.*8* (2003), p. 84-90.

[65] Timo Minartz et al., « Managing hardware power saving modes for high performance computing », in : *2011 International Green Computing Conference and Workshops*, IEEE, p. 25-28, doi : 10.1109/IGCC.2011.6008581.

[66] *MurmurHash2*, [Online ; accessed 11. Jul. 2023], juill. 2023, url : https://github.com/aappleby/smhasher.

[67] Djob Mvondo et al., « OFC : an opportunistic caching system for FaaS platforms », in : *ACM Conferences*, New York, NY, USA : Association for Computing Machinery, avr. 2021, p. 228-244, doi : 10.1145/3447786.3456239.

[68] C. G. Nevill-Manning et I. H. Witten, « Linear-time, incremental hierarchy inference for compression », in : *Proceedings DCC '97. Data Compression Conference*, IEEE, mars 1997, p. 3-11, isbn : 978-0-8186-7761, doi : 10.1109/DCC.1997.581951.

[69] Louis-Marie Nicolas et al., « SLRL : A Simple Least Remaining Lifetime File Eviction policy for HPC multi-tier storage systems », in : *SIGOPS Oper. Syst. Rev.* 56.*1* (juin 2022), p. 70-76, issn : 0163-5980, doi : 10.1145/3544497.3544509.

[70] *November 2014 | TOP500*, [Online ; accessed 30. Dec. 2024], url : https://top500.org/lists/top500/2014/11.

[71] *November 2024 | TOP500*, [Online ; accessed 30. Dec. 2024], url : https://top500.org/lists/top500/2024/11.

[72] Pierre Olivier, Jalil Boukhobza et Eric Senn, « Toward a Unified Performance and Power Consumption NAND Flash Memory Model of Embedded and Solid State Secondary Storage Systems », in : *arXiv* (juill. 2013), doi : 10.48550/arXiv.1307.1217, eprint : 1307.1217.

[73] *Omniscio*, [Online ; accessed 11. Jul. 2023], juill. 2023, url : https://github.com/mdorier/omniscio.

[74] Mayur R. Palankar et al., « Amazon S3 for science grids : a viable solution ? », in : *ACM Conferences*, New York, NY, USA : Association for Computing Machinery, juin 2008, p. 55-64, doi : 10.1145/1383519.1383526.

[75] Christina M. Patrick et al., « Cashing in on hints for better prefetching and caching in PVFS and MPI-IO », in : *HPDC '10 : Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, New York, NY, USA : Association for Computing Machinery, juin 2010, p. 191-202, isbn : 978-1-60558942-8, doi : 10.1145/1851476.1851499.

[76] Arnab K Paul, Ahmad Maroof Karimi et Feiyi Wang, « Characterizing machine learning I/O workloads on leadership scale HPC systems », in : *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, IEEE, 2021, p. 1-8.

[77] James C. Phillips et al., « Scalable molecular dynamics on CPU and GPU architectures with NAMD », in : *J. Chem. Phys.* 153.*4* (juill. 2020), p. 044130. issn : 1089-7690, doi : 10.1063/5.0014475, eprint : 32752662.

[78] James C. Phillips et al., « Scalable molecular dynamics with NAMD », in : *J. Comput. Chem.* 26.*16* (déc. 2005), p. 1781-1802, issn : 0192-8651, doi : 10.1002/jcc.20289.

[79] *posix_fadvise*, [Online ; accessed 12. Nov. 2024], nov. 2024, url : https://pubs.opengroup.org/onlinepubs/9699919799/functions/posix_fadvise.html.

[80] Sophie Robert, Soraya Zertal et Philippe Couvee, « Shaman : A flexible framework for auto-tuning hpc systems », in : *Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, Springer, 2020, p. 147-158.

[81] Liana V Rodriguez et al., « Learning cache replacement with {CACHEUS} », in : *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021, p. 341-354.

[82] Nosayba El-Sayed et Bianca Schroeder, « To checkpoint or not to checkpoint : Understanding energy-performance-I/O tradeoffs in HPC checkpointing », in : *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, p. 22-26, isbn : 978-1-4799-5548-0, doi : 10.1109/CLUSTER.2014.6968778.

[83] Frank Schmuck et Roger Haskin, « {GPFS} : A {Shared-Disk} file system for large computing clusters », in : *Conference on file and storage technologies (FAST 02)*, 2002.

[84] Philip Schwan et al., « Lustre : Building a file system for 1000-node clusters », in : *Proceedings of the 2003 Linux symposium*, t. 2003, 2003, p. 380-386.

[85] William C. Skamarock et al., *A Description of the Advanced Research WRF Version 2*, [Online ; accessed 18. Sep. 2025], juin 2005, url : https://apps.dtic.mil/sti/html/tr/ADA487419.

[86] *SQLite Home Page*, [Online ; accessed 3. Oct. 2024], sept. 2024, url : https://sqlite.org/index.html.

[87] Pradeep Subedi, Philip E. Davis et Manish Parashar, « RISE : Reducing I/O Contention in Staging-based Extreme-Scale In-situ Workflows », in : *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, sept. 2021, p. 146-156, doi : 10.1109/Cluster48925.2021.00021.

[88] Kun Tang et al., « Toward Managing HPC Burst Buffers Effectively : Draining Strategy to Regulate Bursty I/O Behavior », in : *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, IEEE, 2017, p. 20-22, doi : 10.1109/MASCOTS.2017.35.

[89] Luis Thomas et al., « Predicting file lifetimes for data placement in multi-tiered storage systems for HPC », in : *ACM Conferences*, New York, NY, USA : Association for Computing Machinery, avr. 2021, p. 1-9, doi : 10.1145/3439839.3458733.

[90] Aidan P. Thompson et al., « LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales », in : *Comput. Phys. Commun.* 271 (fév. 2022), p. 108171, ISSN : 0010-4655, DOI : 10.1016/j.cpc.2021.108171.

[91] Avishay Traeger et al., « A nine year study of file system and storage benchmarking », in : *ACM Trans. Storage* 4.*2* (mai 2008), p. 1-56, ISSN : 1553-3077, DOI : 10.1145/1367829.1367831.

[92] N. Tran et D. A. Reed, « Automatic ARIMA time series modeling for adaptive I/O prefetching », in : *IEEE Trans. Parallel Distrib. Syst.* 15.*4* (mars 2004), p. 362-377, DOI : 10.1109/TPDS.2004.1271185.

[93] Marc-André Vef et al., « GekkoFS - A Temporary Distributed File System for HPC Applications », in : *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, p. 10-13, DOI : 10.1109/CLUSTER.2018.00049.

[94] Vivekanand Vellanki et Ann L. Chervenak, « A cost-benefit scheme for high performance predictive prefetching », in : *SC '99 : Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, New York, NY, USA : Association for Computing Machinery, jan. 1999, 50-es, ISBN : 978-158113091, DOI : 10.1145/331532.331582.

[95] Giuseppe Vietri et al., *Driving Cache Replacement with {ML-based} {LeCaR}*, [Online ; accessed 18. Sep. 2025], 2018, URL : https://www.usenix.org/conference/hotstorage18/presentation/vietri.

[96] *vmstat(8) - Linux manual page*, [Online ; accessed 24. Jan. 2025], jan. 2025, URL : https://man7.org/linux/man-pages/man8/vmstat.8.html.

[97] Chen Wang et al., « Recorder 2.0 : Efficient parallel I/O tracing and analysis », in : *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, 2020, p. 1-8.

[98] Sage A Weil, « Ceph : reliable, scalable, and high-performance distributed storage », in : (2007).

[99] *xcompact3d/Incompact3d :* [Online ; accessed 5. Jan. 2024], jan. 2024, DOI : 10.5281/zenodo.5870206.

[100] Jie Yu et al., « Rethinking Node Allocation Strategy for Data-intensive Applications in Consideration of Spatially Bursty I/O », in : *ACM Conferences*, New York, NY, USA : Association for Computing Machinery, juin 2018, p. 12-21, DOI : 10.1145/3205289.3205305.

[101] Andrew Zaliwski, Stefan Lankes et Oliver Sinnen, « Evaluating DVFS Scheduling Algorithms on Real Hardware », in : *2016 45th International Conference on Parallel Processing Workshops (ICPPW)*, IEEE, p. 16-19, DOI : 10.1109/ICPPW.2016.48.

[102] Xiaowei Zhu, Wentao Han et Wenguang Chen, {*GridGraph*} *:* {*Large-Scale*} *Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning*, [Online ; accessed 3. Oct. 2024], 2015, URL : https://www.usenix.org/conference/atc15/technical-session/presentation/zhu.

COLLEGES SCIENCES
BRETAGNE POUR L'INGENIEUR
LOIRE ET LE NUMERIQUE

ENSTA
IP PARIS

Titre : Optimisation de l'Énergie des E/S par apprentissage frugal et fé- déré pour des applications HPC

Mot clés : Calcul Haute Performance, Énergie, Stockage, Modélisation des E/S

Résumé : La croissance exponentielle de la production, du traitement et du stockage de données numériques touche un large spectre d'acteurs, allant des géants du web (Google, Facebook, Netflix) aux institutions financières et aux programmes scientifiques de grande envergure. Parallèlement, les performances des processeurs ont progressé grâce à la miniaturisation des composants et à l'optimisation des architectures. Toutefois, malgré une amélioration notable des performances par Watt, la consommation énergétique des supercalculateurs reste très élevée, en raison de leur taille croissante : la puissance cumulée des dix plus grands supercalculateurs est passée de 63 MW en 2014 à 156 MW en 2024. Cette consommation, comparable à celle d'une petite ville, engendre des coûts significatifs et contribue aux émissions de gaz à effet de serre. Dans ce contexte, trois contributions sont proposées. La première est GrIOt, un modèle frugal, réutilisable et fédérable de prédiction des E/S. Ce modèle peut être utilisé pour actionner des indices d'usage de données et le DVFS, et ainsi optimiser la consommation énergétique dans un contexte HPC La deuxième contribution consiste en une méthodologie d'évaluation de l'effet des indices d'usage de données sur les systèmes de fichiers parallèles. Enfin, la troisième contribution porte sur une étude de l'effet du DVFS sur l'efficacité énergétique des phases d'E/S.

Title: I/O Energy Optimization through Frugal and Federated Learning for HPC Applications

Keywords: High Performance Computing, Energy, Storage, I/O Modeling

Abstract: The exponential growth in the production, processing, and storage of digital data affects a wide range of stakeholders, from web giants (Google, Facebook, Netflix) to financial institutions and large-scale scientific programs. In parallel, processor performance has improved through component miniaturization and architecture optimization. However, despite a significant increase in performance per Watt, the total energy consumption of supercomputers remains very high due to their growing size: the combined power of the world's ten largest supercomputers increased from 63 MW in 2014 to 156 MW in 2024. This consumption, comparable to that of a small city, leads to significant costs and contributes to greenhouse gas emissions. In this context, three contributions are proposed. The first is GrIOt, a frugal, reusable, and federable I/O prediction model. This model can be used to trigger file access pattern hints and DVFS to optimize energy consumption in HPC environments. The second contribution is a methodology for evaluating the impact of file access pattern hints on parallel file systems. Finally, the third contribution focuses on energy optimization via DVFS during I/O phases.