Optimisation de l'allocation de la mémoire cache CPU pour les fonctions cloud et les applications haute performance

Armel Jeatsa Toulepi

RÉPUBLIQUE FRANÇAISE
Liberté Égalité Fraternité

Université de Toulouse

Doctorat de l'Université de Toulouse

préparé à Toulouse INP

Optimisation de l'allocation de la mémoire cache CPU pour les fonctions cloud et les applications haute performance

Thèse présentée et soutenue, le 11 septembre 2024 par

Armel JEATSA TOULEPI

École doctorale
EDMITT - Ecole Doctorale Mathématiques, Informatique et Télécommunications de Toulouse

Spécialité
Informatique et Télécommunications

Unité de recherche
IRIT : Institut de Recherche en Informatique de Toulouse

Thèse dirigée par
Daniel HAGIMONT

Composition du jury
M. André-Luc BEYLOT, Président, Toulouse INP
Mme Fabienne BOYER, Rapporteuse, Université Grenoble Alpes
M. Nikos PARLAVANTZAS, Rapporteur, INSA Rennes
M. Djob MVONDO, Examinateur, Université de Rennes
M. Noël DE PALMA, Examinateur, Université Grenoble Alpes
M. Daniel HAGIMONT, Directeur de thèse, Toulouse INP

Membres invités
M. Boris Teabe, Toulouse INP

Résumé

Les services informatiques contemporains reposent principalement sur deux paradigmes majeurs : le cluster computing et le cloud computing. Le premier implique la répartition des tâches de calcul entre différents nœuds qui fonctionnent ensemble comme un seul système, tandis que le second se fonde sur la virtualisation de l'infrastructure informatique qui permet sa fourniture à la demande. Dans le cadre de cette thèse, notre attention se porte sur l'allocation du cache de dernier niveau (LLC) dans le contexte de ces deux paradigmes, en se concentrant spécifiquement sur les applications distribuées et les fonctions FaaS. Le LLC est un espace mémoire partagé et utilisé par tous les cœurs d'un socket NUMA. Étant une ressource partagée, il est sujet à de la contention qui peut avoir un impact significatif sur les performances. Pour pallier ce problème, Intel a mis en œuvre une technologie dans ses processeurs qui permet le partitionnement et l'allocation de la mémoire cache : *Cache Allocation Technology* (CAT).

Dans ce travail, à l'aide de la technologie CAT, nous examinons d'abord l'impact de la contention du LLC sur les performances des fonctions FaaS (*Function as a Service*). Ensuite, nous étudions comment cette contention dans un sous-ensemble de nœuds d'un cluster affecte les performances globales d'une application distribuée en cours d'exécution. De ces études, nous proposons *CASY* et *CADiA*, des systèmes d'allocation intelligents du LLC respectivement pour les fonctions FaaS et pour les applications distribuées. *CASY* utilise l'apprentissage automatique supervisé pour prédire les besoins en cache d'une fonction FaaS en se basant sur la taille des données d'entrée, tandis que *CADiA* construit dynamiquement le profil d'une application distribuée et effectue une allocation harmonisée sur tous les nœuds en fonction de ce profil. Ces deux solutions nous ont permis d'obtenir des gains de performance allant jusqu'à environ 11% pour *CASY*, et 13% pour *CADiA*.

Mots clés : *Cache CPU, Allocation, FaaS, Application distribuée parallèle, Sans Serveur, Calcul Haute Performance.*

Abstract

Contemporary IT services are mainly based on two major paradigms : cluster computing and cloud computing. The former involves the distribution of computing tasks between different nodes that work together as a single system, while the latter is based on the virtualization of computing infrastructure, enabling it to be provided on demand. In this thesis, our focus is on last-level cache (LLC) allocation in the context of these two paradigms, concentrating specifically on distributed parallel applications and FaaS functions. The LLC is a shared memory space used by all processor cores on a NUMA socket. As a shared resource, it is subject to contention, which can have a significant impact on performance. To alleviate this problem, Intel has implemented a technology in its processors that enables partitioning and allocation of cache memory : *Cache Allocation Technology* (CAT).

In this work, using CAT, we first examine the impact of LLC contention on the performance of FaaS (Function as a Service) functions. Then, we study how this contention in a subset of nodes in a cluster affects the overall performance of a running distributed application. From these studies, we propose *CASY* and *CADiA*, intelligent LLC allocation systems for FaaS functions and distributed applications respectively. *CASY* uses supervised machine learning to predict the cache requirements of a FaaS function based on the size of the input data, while *CADiA* dynamically constructs the cache usage profile of a distributed application and performs harmonized allocation across all nodes according to this profile. These two solutions enabled us to achieve performance gains of up to around 11% for *CASY*, and 13% for *CADiA*.

Keywords : *CPU Cache, Allocation, FaaS, Distributed Parallel Application, Serverless, High Performance Computing.*

Table des matières

Table des figures

Introduction générale

Les services de stockage et de traitement des données informatiques reposent aujourd'hui sur deux principaux paradigmes qui ont révolutionné le domaine. Le premier est le **cluster computing** [1] et le second le **cloud computing** [2]. La différence entre les deux résidant au niveau de leur architecture, leur déploiement et leur gestion.

Le **cloud computing** ou « informatique en nuages » désigne la fourniture à la demande de ressources informatiques par l'intermédiaire d'internet. Ces ressources incluent des serveurs physiques et virtuels, des logiciels et applications, des espaces de stockage. La tarification se fait à l'utilisation. Cette technologie repose principalement sur la virtualisation de l'infrastructure informatique qui est rendue abstraite. Cela permet sa mise en commun et sa division sans tenir compte des limites physiques du matériel. Ce qui permet aux fournisseurs de services cloud d'utiliser au maximum les ressources de leur centre de données. Un fournisseur de Cloud se caractérise par le modèle de service qu'il offre. Les principaux modèles de service sont les suivants : le **IaaS (Infrastructure As A Service)**, caractérisé par la fourniture des ressources d'infrastructure à la demande ; le **PaaS (Platform As A Service)** caractérisé par la fourniture d'une solution intégrée, d'une pile de solutions, ou d'un service via internet ; et le **SaaS (Software As A Service)** caractérisé par la fourniture d'une application complète gérée par le fournisseur par l'intermédiaire d'un navigateur web.

Ces modèles de service ont donné naissances au modèle **sans serveur** (*serverless*). Ce modèle transfère toutes les tâches de gestion de l'infrastructure back-end (provisionnement, mise à l'échelle, ordonnancement, application de correctifs) au fournisseur de services cloud. Il permet aux développeurs de se consacrer pleinement au code et à la logique métier de leurs applications. Il existe plusieurs modèles sans serveur, mais celui sur lequel se focalise notre travail est le modèle **FaaS (Function as a Service)** [3]. Le modèle FaaS repose sur le développement d'applications sous forme d'ensemble de **fonctions sans état** (*stateless*). Il

permet aux développeurs de créer, d'exécuter et de gérer des applications en tant qu'ensembles de fonctions, sans avoir à assurer la maintenance de leur propre infrastructure. Elles sont exécutées dans des conteneurs Linux entièrement gérés par le fournisseur. Dans cette configuration, la facturation ne se fait plus sur la base du temps de fonctionnement de l'infrastructure comme dans le IaaS ou de l'environnement d'exécution comme dans le PaaS, elle dépend plutôt du temps d'exécution des fonctions. La tarification se fait donc à l'usage et de façon plus granulaire.

La fonction constitue ainsi l'élément central du modèle FaaS. Par conséquent, l'amélioration de la qualité des services de ce modèle se concentre principalement sur la réduction du temps de démarrage et d'exécution des fonctions. Les travaux de recherche dans le domaine du FaaS ont donc principalement été faits suivant cet objectif. Ils peuvent être classés en trois niveaux : l'**intergiciel de cloud**, l'**environnement d'exécution**, et l'**infrastructure matérielle**. Au niveau de l'**intergiciel de cloud**, les travaux portent principalement sur les algorithmes d'ordonnancement, les algorithmes d'équilibrage de charge, et les techniques de minimisation du nombre démarrage à froid des conteneurs et machines virtuelles. Au niveau de l'**environnement d'exécution**, les travaux de recherche portent essentiellement sur l'optimisation du démarrage des conteneurs et machines virtuelles, les techniques d'optimisation de l'efficience énergétique. Et enfin, au niveau l'**infrastructure matérielle**, les travaux portent principalement sur les techniques de gestion et d'allocation des ressources matérielles telles que le processeur, la mémoire centrale, la bande passante, la TLB (*Translation Look aside Buffer*), et la mémoire cache du processeur. Notre travail se situe au niveau de l'infrastructure matérielle, plus précisément la gestion et l'allocation de la mémoire cache CPU.

La **mémoire cache** est une mémoire temporaire hautes performances, de petite taille, à proximité immédiate du processeur. Elle fournit un accès rapide aux données et instructions fréquemment utilisées. La vaste majorité des processeurs (de serveurs ou d'ordinateurs) modernes, à quelques exceptions près, ont trois niveaux de caches CPU. Les deux premiers niveaux sont dits « privés », c'est-à-dire accessibles uniquement par un seul cœur. Le dernier niveau quant à lui est dit « partagé », c'est-à-dire accessible par plusieurs ou tous les cœurs de la machine. Ce cache de dernier niveau ayant une taille limitée, il peut rapidement devenir un goulot d'étranglement lorsque plusieurs entités en réclament une quantité substantielle. En effet, lorsque le processeur doit exécuter une instruction d'un processus, il rapatrie les données nécessaires à l'exécution de cette instruction dans la mémoire cache. Si cette dernière

est pleine, une donnée existante sera évincée pour insérer la nouvelle. Cela aura un impact négatif sur le processus auquel appartient la donnée évincée. Par conséquent, plus il y a des processus qui réclament des quantités considérables de mémoire cache, plus ces cycles d'insertion-éviction sont fréquents. Le processeur est alors contraint d'aller plus fréquemment en mémoire centrale pour récupérer les informations, ce qui ralenti l'exécution des instructions, et donc des processus. Ce phénomène est appelé la **contention de cache**.

Les conteneurs Linux sur lesquels s'exécutent les fonctions FaaS ne sont rien d'autre que des groupes de processus. On rencontre donc également ce problème de contention de cache dans les environnements de FaaS. Il est donc pertinent de mettre en place une politique de gestion et d'allocation de la mémoire cache CPU pour limiter les effets de cette contention. Ce qui aura pour conséquence de minimiser le temps d'exécution des processus et par conséquent des fonctions FaaS. Pour répondre à ce problème, nous avons donc mis en œuvre *CASY (**C**ache **A**llocation **SY**stem)*, un système intelligent d'allocation du cache de dernier niveau (LLC[1]) pour les fonctions FaaS. *CASY* utilise l'apprentissage automatique supervisé pour prédire les besoins en cache d'une fonction FaaS en se basant sur la taille des données d'entrée. Lorsqu'une fonction s'exécute, *CASY* anticipe la quantité de cache qui lui est nécessaire et effectue l'allocation en tenant compte de la charge actuelle du cache par les autres fonctions en cours d'exécution. *CASY* nous a parmi obtenir des gains de performance allant jusqu'à environ 11%.

Le **cluster computing** ou « informatique en grappe » désigne le processus de partage des tâches de calcul entre plusieurs ordinateurs interconnectés qui fonctionnent ensemble comme un système unique appelé *cluster*. Les opérations s'effectuent sur la base du principe des systèmes distribués. Cette technologie est le plus souvent utilisée dans l'informatique à haute performance pour résoudre des opérations plus complexes de manière plus efficace. Elle permet une vitesse de traitement plus rapide et une meilleure intégrité des données. Dans ce contexte, les applications s'exécutent sous forme d'un ensemble de processus distribués sur les machines du cluster. On parle alors d'applications distribuées. Nous avons vu plus haut que les processus peuvent être sujets au problème de contention de cache. Par conséquent, les applications distribuées, qui ne sont qu'un ensemble de processus, ne sont pas exemptes de ce problème. Cette contention de cache peut donc ralentir l'exécution de l'application distribuée sur certaines machines du

1. *Last Level Cache*

cluster. Or, le temps d'exécution global d'une application distribuée dépend de ses performances sur chacune des machines sur lesquelles elle s'exécute. Il en résulte que la contention de cache sur certaines machines impacte le temps d'exécution global de l'application distribuée. De ce fait, il est intéressant de mettre en place une politique de gestion et d'allocation de la mémoire cache pour les applications distribuées, afin de minimiser l'impacte de la contention de cache. Toutefois, une vision à l'échelle d'une machine du problème de contention de cache entrainerait nécessairement une allocation différente du cache CPU à l'application distribuée d'une machine à l'autre. Ce qui conduirait à un gaspillage ou une allocation insuffisante de la mémoire cache sur certaines machines. Il est par conséquent nécessaire de prendre de la hauteur et d'adopter une vision à un échelon supérieur, celui du cluster. Cela permettra d'effectuer une allocation harmonisée sur toutes les machines, tout en tenant compte du voisinage de l'application distribuée sur chacune des machines. C'est dans cette optique que nous avons développé *CADiA (**C**ache **A**llocation system for **Di**stributed **A**pplications)*, un système d'allocation de cache pour applications distribuées parallèles. *CADiA* construit dynamiquement le profil d'une application distribuée en récupérant ses métriques d'utilisation du cache sur chacune des machines. Ensuite, elle utilise une politique d'allocation qui prend en compte ce profil ainsi que la priorité accordée à l'application distribuée. *CADiA* nous a parmi obtenir des gains de performance allant jusqu'à 13%.

La suite de cette thèse est structurée en plusieurs chapitres. Dans le premier chapitre, nous allons présenter le cadre théorique, autrement dit tous les concepts nécessaires à la compréhension de notre travail. Dans le deuxième chapitre, nous présenterons notre première contribution qui est *CASY*, un système d'allocation intelligent de mémoire cache CPU pour les environnements de FaaS. Dans le troisième chapitre, nous détaillerons *CADiA*, notre deuxième contribution, qui est un système d'allocation intelligent et dynamique de la mémoire cache CPU pour les applications distribuées parallèles. Dans le quatrième chapitre, nous ferons un état de l'art des travaux de recherches adressant des problématiques connexes à la nôtre. Dans l'avant-dernier chapitre, nous ferons une conclusion générale, faisant un bilan et discutant des limites des travaux effectués durant cette thèse. Et enfin, le dernier chapitre consistera à la présentation des perspectives à court terme et à long terme de notre travail.

Chapitre 1

Cadre théorique

Dans ce chapitre, nous allons fournir une explication approfondie des concepts et des outils essentiels pour comprendre notre travail.

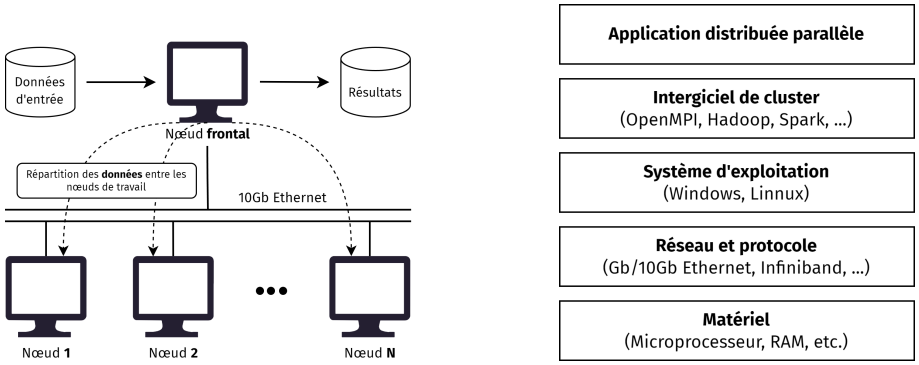
1.1 Cluster computing et applications distribuées

Comme nous l'avons vu dans l'introduction, le cluster computing fait référence au processus de partage des tâches de calcul entre plusieurs ordinateurs interconnectés qui fonctionnent ensemble comme un système unique. Il est couramment employé pour des opérations complexes qui demandent beaucoup de ressources (calcul haute performance, big data, serveur web, etc.), nécessitant ainsi la combinaison de la puissance de calcul de plusieurs machines simultanément. Pour fonctionner convenablement, les machines d'un cluster, également appelés **nœuds**, doivent être connectées par le biais d'interconnexions à haut débit, et généralement dans un même réseau local (LAN). Sur la figure 1.1a nous pouvons voir l'architecture simple d'un cluster. Le nœud frontal, point d'entrée et de sortie du cluster, reçoit les données d'entrée et les repartis entre les nœuds de travail pour leur traitement. Cette répartition est cruciale pour équilibrer la charge de travail et optimiser les performances du cluster. Le nœud frontal est également responsable de la collecte des résultats après traitement. Et enfin, les nœuds sont interconnectés via un réseau Ethernet, par exemple 10 Gigabit. On distingue principalement trois types de cluster :

1. **Les clusters à haute performance** [4] : majoritairement constitués de superordinateurs pour résoudre des problèmes de calcul complexe,

ils sont conçus pour tirer parti de la puissance de traitement parallèle de plusieurs nœuds.

2. **Les clusters à haute disponibilité** [5] : également appelés clusters de basculement et constitués de nœuds redondants, ils sont configurés de sorte qu'en cas de défaillance de l'un des nœuds, un autre peut être utilisé de manière transparente pour maintenir la disponibilité du service ou de l'application fourni.
3. **Les clusters à équilibrage de charge** [6] : généralement utilisés dans les environnements d'hébergement web, ils sont conçus de façon à ce que les requêtes entrantes soient réparties entre les nœuds, permettant ainsi d'éviter qu'un seul nœud ne reçoive une quantité disproportionnée de tâches.



(a) (b)

FIGURE 1.1 – À gauche Architecture simple d'un cluster et à droite les composants principaux d'une architecture d'un cluster

Les clusters sur lesquels portent notre etude sont les clusters à haute performance. Les applications capables de s'exécuter dans ces environnements de cluster, c'est-à-dire simultanément sur plusieurs machines, sont appelées **applications distribuées**. Et celles auxquelles nous allons nous intéresser sont celles suivant le modèle de programmation **SPMD**, qui signifie **Single Program, Multiple Data** (en français « programme unique, données multiples »), qui est le modèle de programmation le plus commun. Dans ce modèle de programmation, les données d'entrée sont partitionnées et envoyées sur chaque nœud de calcul qui effectuent le même ensemble d'opérations sur celles-ci. On a donc *un seul* programme, mais de *multiples* jeux de données.

Les principaux composants d'un cluster tels que représentés sur la figure 1.1b sont les suivants :

— **l'application distribuée** qui s'exécute sur les nœuds de travail ;
— **l'intergiciel de cluster** qui permet la communication et la gestion des données nécessaires à l'exécution des applications distribuées ;
— le **système d'exploitation** sous lequel fonctionnent les nœuds de travail ;
— le **matériel et protocole réseau** responsable de l'interconnexion et de la communication entre les nœuds de travail ;
— et enfin le **matériel** qui constitue les nœuds de travail.

Le cluster computing offre de nombreux avantages tels que la performance, la scalabilité, la tolérance aux pannes et la disponibilité. Néanmoins, il a deux inconvénients majeurs qui sont : un rapport coût-efficacité faible en raison du coût du matériel et un besoin en espace et matériel physique élevé à mesure que le besoin en ressource augmente. C'est pour pallier ces inconvénients du cluster computing qu'est née le cloud computing dont nous allons parler dans la section suivante.

1.2 Cloud Computing et fonctions FaaS

Le cloud computing désigne la mise à disposition à la demande de ressources informatiques, en particulier de puissance de calcul, de stockage de données, d'applications et d'autres ressources informatiques. Ceci par l'intermédiaire d'internet et avec une tarification à l'utilisation. Il offre donc à un particulier ou une entreprise la possibilité d'accéder à n'importe quelle ressource à n'importe quel moment. La gestion et de la maintenance de l'infrastructure matérielle est à la charge du fournisseur de service cloud. La figure 1.2 illustre l'architecture d'une plateforme de cloud computing. L'infrastructure client, qui constitue la partie *front end*, représente les systèmes et appareils (ordinateurs, smartphones, tablettes, etc.) du client qui interagissent avec les services cloud via internet. La partie *back end* est structurée en plusieurs couches. La couche **Application** contient les applications spécifiques que les utilisateurs exécutent sur la plateforme cloud. Il s'agit des logiciels ou services accessibles par les utilisateurs. La couche **Service** comprend les services web et API que les applications utilisent pour fonctionner. l'**Environnement d'exécution** est l'environnement dans lequel les applications s'exécutent. Cela peut inclure des machines virtuelles, des conteneurs ou d'autres environnements d'exécution. La couche **Stockage** fournit les capacités de stockage de données nécessaires pour les applications et services. Elle peut inclure des bases de données, des systèmes de fichiers distribués,

des objets de stockage, etc. La couche **Infrastructure** est la base matérielle et virtuelle qui soutient tout le reste de l'architecture. Elle comprend les serveurs, le matériel réseau, le matériel de stockage et les autres composants physiques. La couche **Gestion** traite de la gestion et de l'orchestration des différentes couches et services de la plateforme cloud. Elle inclut la gestion des ressources, la mise à l'échelle, le déploiement, la surveillance et l'entretien. Et enfin, la couche **Sécurité** englobe les mécanismes et politiques de sécurité appliqués à toutes les couches de l'architecture cloud. Elle couvre des aspects tels que l'authentification, l'autorisation, le chiffrement des données, la conformité et la protection contre les menaces.

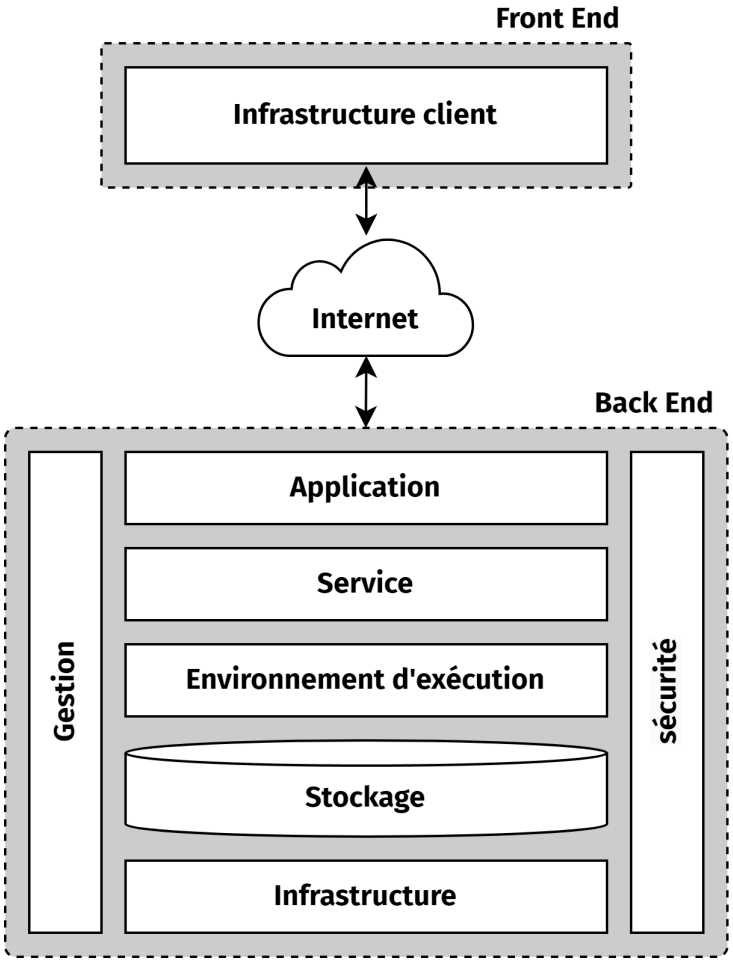


FIGURE 1.2 – Architecture du cloud computing

Il existe plusieurs modèles de cloud computing et celui qui nous

intéresse est le FaaS [3]. Il apporte aux développeurs l'abstraction nécessaire pour concevoir une application sous forme d'un ensemble de fonctions dont les exécutions sont déclenchées par des événements. Ces événements peuvent être des requêtes ou des commandes directes à partir d'un terminal d'administration. Une fonction ne contient que du code métier sauvegardé dans une base de données, et est exécutée dans un environnement isolé (*sandbox*), qui peut être un conteneur tel que Docker [7], ou une machine virtuelle légère telle que Firecracker [8]. Le fournisseur est responsable de l'allocation et de la gestion des ressources sans l'intervention de l'utilisateur. L'allocation des ressources se fait automatiquement en fonction de la charge de travail sollicitée par l'application de l'utilisateur, ce qui permet une facturation granulaire.

Les plateformes de FaaS, à quelques différences près, ont une architecture semblable à celle représentée sur la figure 1.3. Elles fonctionnent généralement de la façon suivante : les *événements* sont générés à partir des requêtes utilisateurs, des passerelles API, ou des plateformes de cloud. Ces événements passent par une *file d'attente* pour arriver au *contrôleur de FaaS*, qui, après analyse du contenu, récupère dans la *base de données* le code source de la *fonction* à exécuter, puis le passe à *l'orchestrateur*. L'orchestrateur, en fonction d'un algorithme d'ordonnancement, démarre un *conteneur* dans un des nœuds de travail, y injecte le code source de la fonction et démarre son exécution. Lorsque la fonction termine son exécution, elle enregistre le résultat de son exécution dans la base de données. Le contrôleur de FaaS récupère le résultat de la fonction dans la base de données et envoie finalement le résultat de la requête au client.

Le cloud computing a permis de corriger les inconvénients majeurs du cluster computing. Le premier étant la flexibilité, les ressources peuvent être facilement augmentées ou réduites à la demande, permettant ainsi de gérer efficacement les charges de travail et d'avoir une facturation granulaire et donc un rapport coût-efficacité optimal. Le deuxième, la centralisation et la mutualisation des ressources, grâce à la virtualisation les ressources matérielles sont mises en commun et partagées. Le fournisseur de services alloue ensuite les ressources de manière dynamique selon les besoins. Néanmoins, le cloud computing possède des inconvénients tels qu'une gestion de la sécurité plus complexe, une augmentation du risque lié à la confidentialité des données et une migration d'un fournisseur de cloud vers un autre coûteuse.

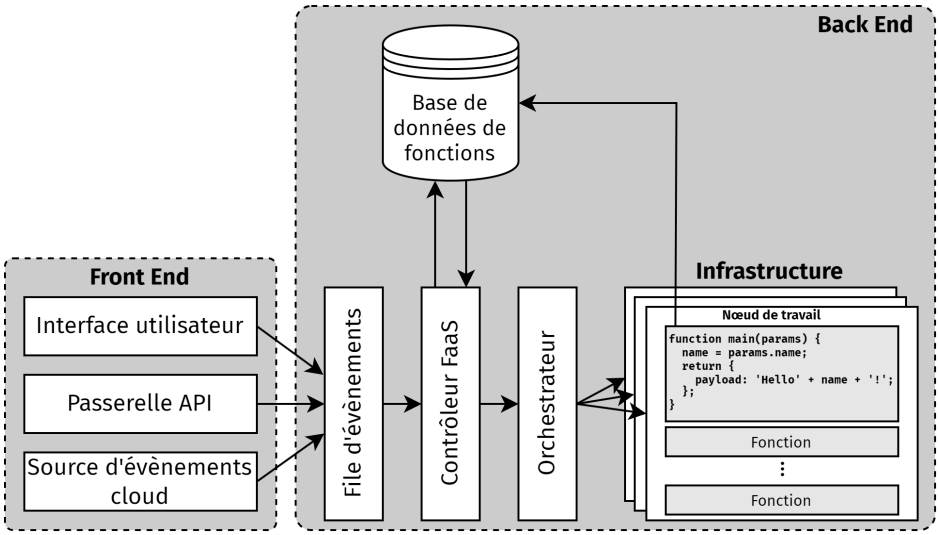


FIGURE 1.3 – Architecture simple d'une plateforme de FaaS

1.3 Microprocesseur et mémoire cache

Le microprocesseur est le circuit intégré qui effectue les opérations arithmétiques et logiques, contrôle les périphériques d'entrée et de sortie et exécute les instructions de la mémoire. C'est le cerveau d'un ordinateur ou d'un appareil électronique. Pour fonctionner efficacement, il est équipé d'une mémoire de travail située à proximité, appelée mémoire cache.

La mémoire cache est un composant mémoire située à proximité du processeur et contenant des entrées mémoire fréquemment utilisées [9] par celui-ci. En effet, lorsqu'il a besoin d'une donnée, le processeur tente d'abord de la récupérer depuis sa mémoire cache. Si la donnée n'est pas présente dans celle-ci, alors elle est récupérée depuis la mémoire centrale, puis insérée dans la mémoire cache avant d'être utilisée par le processeur. Ainsi, le processeur n'utilise jamais une donnée qui ne se trouve pas dans sa mémoire cache, elle est d'abord rapatriée dans cette dernière avant d'être utilisée.

La mémoire cache est en général répartie sur plusieurs niveaux de cache, et plus un niveau est éloigné du processeur, plus sa capacité est grande, mais plus son temps d'accès est élevé. On distingue en général trois niveaux de cache.

1. **Le cache de niveau 1 (L1)** : c'est la mémoire cache la plus rapide, dont la taille est généralement comprise entre 2 KB et 64 KB. Elle est généralement divisée en deux parties, l'une pour stocker les

instructions (L1i) et l'autre pour stocker les données (L1d).

2. **Le cache de niveau 2 (L2)** : il a une plus grande capacité et est légèrement plus lent en que le cache L1. Il peut être partagé entre plusieurs cœurs ou exclusif à un seul cœur, selon l'architecture du processeur. Sa taille est généralement comprise entre 256 Ko et 1024 Ko.
3. **Le cache de niveau 3 (L3 ou LLC)** : encore appelé cache de dernier niveau (en abrégé *LLC* pour *Last Level Cache*), c'est celui qui a la plus grande capacité, mais qui est également le plus lent. Il est partagé par tous les cœurs et sa taille varie entre 1 MB et 128 MB.

La figure 1.4 illustre une hiérarchie de cache à trois niveaux. Les caches de premier niveau *L1i* et *L1d* sont accessibles uniquement par les cœurs auxquels ils sont associés. Il en va de même pour les caches de deuxième niveau. En ce qui concerne le cache de dernier niveau (L3 ou LLC), il est accessible par tous les cœurs par l'intermédiaire d'interconnexions avec les niveaux supérieurs de mémoire cache.

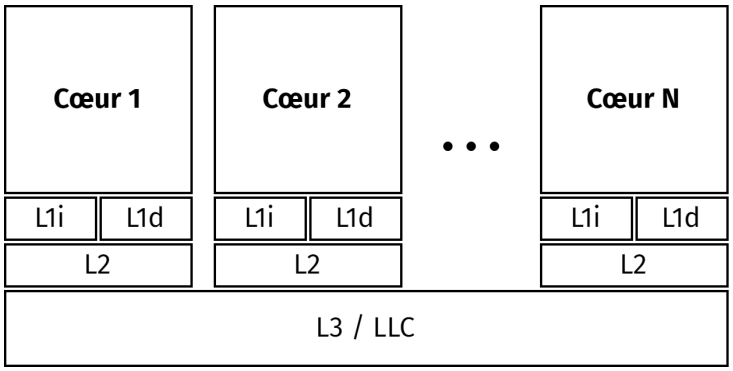


FIGURE 1.4 – Hiérarchie de cache d'un microprocesseur

Avant d'aller plus loin, il est important de parler des systèmes à architecture *NUMA*, car toutes les plateformes serveur que nous avons utilisé tout au long de notre travail reposent sur cette architecture. Un système *NUMA* (pour *Non Uniform Memory Access* signifiant accès mémoire non uniforme) est un système comportant plusieurs nœuds (ou *sockets*) de processeurs. La figure 1.5 présente une architecture *NUMA* simple à quatre nœuds. Chaque nœud a son propre espace mémoire local (et donc son propre LLC), et est relié aux autres nœuds par une connexion performante. Il a été conçu pour pallier les limites de l'architecture classique *SMP* (pour *symmetric multiprocessing*) dans laquelle tout l'espace mémoire est certes accessible par un unique bus,

mais rend de ce fait inefficace, par encombrement, les accès concurrents par les différents processeurs. L'architecture *NUMA* permet donc d'utiliser un plus grand nombre de processeurs simultanément. Chaque nœud contient des processeurs et de la mémoire, tout comme un petit système de SMP. La latence d'accès à une des mémoires peut varier en fonction de la distance physique entre le processeur et la mémoire où se trouvent les données requises. Ainsi, Quand un processeur accède à une mémoire qui ne se trouve pas dans son propre nœud (mémoire distante), les données doivent être transférées suivant la connexion *NUMA*, ce qui est plus lent que d'accéder à la mémoire locale. Les temps d'accès mémoire ne sont pas uniformes et dépendent de l'emplacement de la mémoire et du nœud depuis lesquels ils sont consultés, comme le nom de la technologie le suggère. Par exemple, si le processeur 1 de la figure 1.5 souhaite accéder à une information, il mettra plus de temps si celle-ci se trouve dans une mémoire distante (nœud 0, 2 ou 3) que si elle se trouve dans la mémoire locale (nœud 1). La situation est la même lorsque le processeur veut accéder à une information qui est située dans une mémoire cache (LLC) distante.

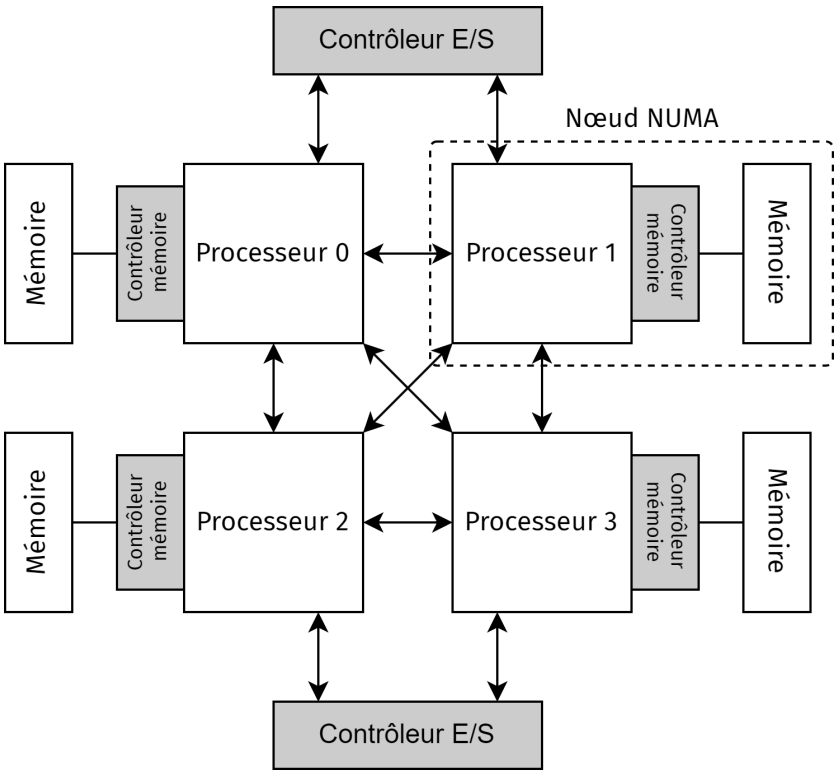


FIGURE 1.5 – Architecture *NUMA* à 4 nœuds

Nous verrons dans la section suivante les métriques qui sont les plus utilisées pour mesurer les performances concernant l'utilisation de la mémoire cache CPU.

1.4 Défauts et taux de défaut de cache

Lorsqu'une donnée requise par le microprocesseur ne se trouve pas dans la mémoire cache, il se produit ce qu'on appelle un **défaut de cache** (*cache miss*). Le processeur est alors obligé d'aller chercher la donnée en mémoire centrale, ce qui ralentit l'opération. On parle de **succès de cache** (*cache hit*) lorsque la donnée se trouve effectivement dans la mémoire cache et que le processeur n'a pas besoin d'aller en mémoire centrale. On distingue 3 types de défaut de cache.

1. **Les défauts obligatoires**, également appelée **défaut à froid**, se produisent lorsqu'une donnée est consultée pour la première fois. Puisque que la donnée n'a pas été demandée auparavant, elle n'est pas présente dans la mémoire cache, ce qui entraîne un défaut.
2. **Les défauts de capacité**, se produit lorsque l'*ensemble de travail* (l'ensemble des données auxquelles un programme accède fréquemment) est plus grand que la taille de la mémoire cache. Lorsque celle-ci est pleine et qu'une nouvelle donnée est référencée, une donnée existante doit être expulsée pour accueillir la nouvelle donnée, ce qui entraîne un défaut.
3. **Les défauts de conflit**, également appelées **défauts de collision**, se produisent lorsqu'une donnée, mappée dans un ensemble déjà plein, doit y être insérée. Une autre donnée doit être expulsée, ce qui entraîne un défaut si l'on accède à nouveau à la donnée expulsée.
4. **Les défauts de cohérence**, qui se produit lorsqu'un processeur met à jour une donnée dans sa mémoire cache privée, ce qui rend caduque la donnée correspondante dans la mémoire cache d'un autre processeur. Lorsque le second processeur accède à la donnée obsolète, un défaut de cohérence se produit.

Les défauts de conflit sont les plus fréquents et sont ceux qui nous intéressent. Dans la suite de notre travail, c'est à eux que nous ferons référence lorsque nous parlerons de défaut de cache.

Plus la fréquence de défaut de cache est élevée, plus les performances d'exécutions s'en trouvent détériorées. Cette fréquence est définie par le *taux de défaut* (ou *taux d'échec*), qui est calculé sur un intervalle de temps donné en faisant le ratio entre le nombre références qui se soldent par des

défauts de cache et le nombre total de références. Elle est généralement exprimée en pourcentage.

$$\text{Taux de défauts } (\%) = \frac{\text{Nombre de défaut}}{\text{Nombre total de références}} \times 100 \tag{1.1}$$

Le taux d'échec (*miss rate*) est donc un indicateur de performance très utile lorsqu'on veut optimiser l'allocation du cache CPU. Une approche ayant fait ses preuves consiste à utiliser les courbes d'utilisation de la mémoire cache, qui représentent le taux d'échec en fonction de la taille de la mémoire cache occupée. Elles portent le nom de **courbe du taux d'échec** [10], qu'on appelle communément **MRC** (pour *Miss Rate Curve*). Ces courbes constituent des outils efficaces pour surveiller et améliorer l'utilisation de la mémoire cache. La figure 1.6 illustre un exemple de courbe de taux d'échec. Elle montre, pour une charge de travail donnée, le taux de défaut de cache (axe des y) en fonction de l'occupation de la mémoire cache (axe des x). Cette courbe est théoriquement toujours

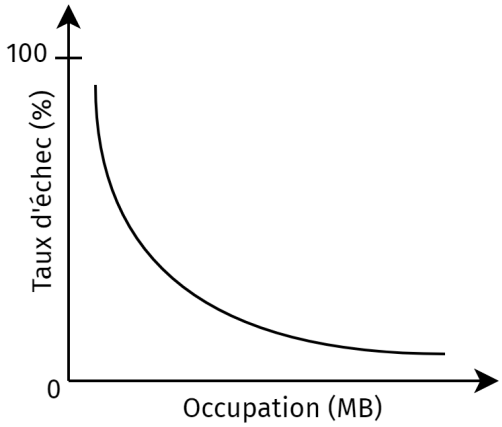


FIGURE 1.6 – Exemple d'une courbe de taux d'échec

décroissante, car une augmentation de la taille de la mémoire cache entraîne une diminution du taux d'échec. Les courbes de taux d'échec donnent un aperçu du comportement historique d'une application envers la mémoire cache.

1.5 Partitionnement de cache et Intel *Cache Allocation Technology* (CAT)

Pour fonctionner de manière efficace, la mémoire cache de processeur utilise un mécanisme de stockage et d'indexation particulier, différent de celui des autres types de mémoires. La figure 1.7 illustre sa structure organisationnelle. La mémoire cache est divisée en **lignes de cache**, la taille de chaque ligne est généralement de 64 Octets. Ces lignes sont organisées en **ensembles** (*sets*). Les lignes de cache d'un ensemble sont également appelées **voies** (*ways*). L'ensemble auquel appartient un bloc mémoire est déterminé à partir d'une partie de son adresse mémoire : l'**index**. plusieurs blocs peuvent donc être affectés à un même ensemble, ils seront donc répartis sur les différentes voies de cet ensemble. Le nombre de lignes ou voies par ensemble détermine l'associativité du cache. De ce fait, si une mémoire cache est organisée en 11 voies par ensemble comme sur la figure 1.7, alors son associativité est de 11.

La façon dont cette division voies/ensemble est faite est appelée technique de **mappage**. On distingue trois techniques de mappage pour les mémoires cache, chacune ayant ses avantages et inconvénients :

1. le **mappage direct** (*direct mapping*) : dans cette technique, chaque bloc de la mémoire principale est associé à une ligne spécifique de la mémoire cache. Il est simple et rentable, mais peut entraîner un taux élevé de défaut de cache lorsque plusieurs blocs de mémoire tentent d'accéder à la même ligne de cache.
2. le **mappage entièrement associatif** (*fully associative mapping*) : dans cette technique, un bloc de mémoire principale peut être chargé dans n'importe quelle ligne de la mémoire cache. Ce qui réduit le taux d'échec, mais augmente la complexité et le coût de la mémoire cache, car elle nécessite plus de ressource pour savoir quel bloc de mémoire se trouve dans quelle ligne de la mémoire cache.
3. le **mappage associatif par ensemble** (*set-associative mapping*) : qui est un compromis entre le mappage direct et le mappage entièrement associatif. Ici le cache est divisé en ensembles et chaque bloc de la mémoire principale peut être chargé dans n'importe quelle ligne d'un ensemble spécifique. Cette technique réduit le taux d'échec par rapport à la mise en correspondance directe, tout en étant moins complexe et moins coûteuse que le mappage entièrement associatif.

En raison de sa capacité limitée, le cache CPU est sujet à des

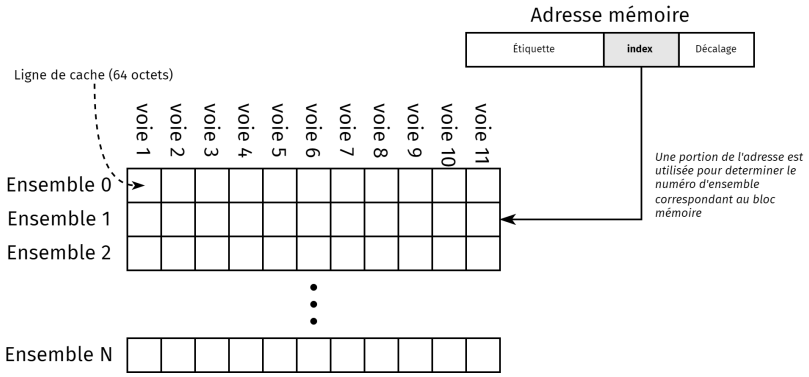


FIGURE 1.7 – Organisation des blocs dans la mémoire cache

problèmes de partage, en particulier le LLC. Ainsi, lorsque plusieurs applications requièrent simultanément et de façon substantielle de la mémoire cache, il se produit ce qu'on appelle la **contention de cache**. Les processus entrent en compétition pour l'occupation de la mémoire cache. Ce qui conduit à une augmentation du taux de défaut de cache et, par conséquent, de la latence d'accès aux données. Il est donc essentiel de mettre en œuvre des techniques de partitionnement et d'allocation du LLC adaptées aux applications. Le partitionnement du LLC est un sujet de recherche qui intéresse la communauté depuis plusieurs décennies. Avant l'avènement du partitionnement matériel, la coloration des pages était l'une des solutions les plus populaires [11, 12]. Cette technique attribue des couleurs aux zones de la mémoire principale et les associe à des voies de cache spécifiques. Ce qui permet de contrôler l'allocation du cache en stockant les données dans des emplacements spécifiques de la mémoire principale avec des couleurs définies.

De nouvelles fonctionnalités intégrées aux microprocesseurs ont permis d'intégrer partitionnement de la mémoire cache. C'est le cas de la technologie CAT (*Cache Allocation Tehcnology*) [13, 14] qui fait partie de la technologie RDT (*Resource Director Technology*) [15] d'Intel. Cette technologie fournit un contrôle programmable par logiciel de la quantité d'espace de mémoire cache qui peut être utilisée par un thread, une application, une machine virtuelle ou un conteneur donné. Cela permet de protéger et hiérarchiser les processus et machines virtuelles dans un environnement de data center bruyant en assurant une allocation efficace et équitable de la mémoire cache.

Cette technologie introduit une structure intermédiaire appelée *classe de service* (en abrégé CLOS pour *CLass Of Service*), qui agit comme une balise de contrôle des ressources. L'unité d'allocation est la voie de cache.

À Chaque CLOS correspond un *masque de capacité* (en abrégé CBM pour *Capacity BitMask*). Dans ce masque, le bit "1" signifie qu'un processus appartenant à ce CLOS peut utiliser la voie de cache spécifiée et le bit de "0" signifie le contraire. Ceci permet donc de contrôler la quantité de cache que les processus d'un CLOS peuvent consommer. Les valeurs à l'intérieur du CBM indiquent la quantité relative de cache disponible et le degré de chevauchement ou d'isolement. Par exemple, sur la figure 1.8 les processus appartenant à la classe de service CLOS[0] n'ont accès qu'aux trois premières voies de cache, il en résulte qu'ils ont accès à $\frac{3}{11^{ème}}$ (environ 27%) de la mémoire cache. Nous pouvons également voir que le CLOS[0] a moins de cache disponible que le CLOS[3] et pourrait être considéré comme moins prioritaire ou ayant des besoins moindres en cache.

	voie 1	voie 2	voie 3	voie 4	voie 5	voie 6	voie 7	voie 8	voie 9	voie 10	voie 11	
CLOS[0]	1	1	1	0	0	0	0	0	0	0	0	CBM = **0x700**
CLOS[1]	0	0	0	1	1	1	0	0	0	0	0	CBM = **0x0e0**
CLOS[2]	0	0	0	0	0	0	1	1	1	0	0	CBM = **0x01c**
CLOS[3]	0	0	0	0	0	0	1	1	1	1	1	CBM = **0x01f**

FIGURE 1.8 – Chevauchement et isolement de masques de capacité (CBM) entre plusieurs classes de service (CLOS)

La technologie CAT apporte de nouvelles possibilités d'amélioration dans le domaine du partitionnement et de l'allocation de cache. Cependant, il n'est possible de définir qu'un nombre limité de CLOS.

1.6 Apprentissage supervisé et classification

L'apprentissage supervisé [16] est une branche de l'intelligence artificielle. Il consiste à utiliser des ensembles de données étiquetées pour entraîner les algorithmes à classer les données ou à prédire les résultats avec précision. Un ensemble d'apprentissage est utilisé pour enseigner aux modèles à produire les résultats souhaités. Cet ensemble d'entraînement comprend des entrées et des sorties correctes, ce qui permet au modèle de s'améliorer progressivement. L'apprentissage supervisé permet de traiter plusieurs types de problèmes parmi lesquels la **classification**.

La classification utilise un algorithme pour attribuer avec précision des données de test à des catégories particulières. L'algorithme reconnaît des entités spécifiques dans le jeu de données et tente de tirer des conclusions sur la façon dont ces entités doivent être étiquetées ou définies. Les algorithmes de classification courants sont les classificateurs linéaires, les machines à vecteurs de support (SVM), les arbres de décision, les k plus proches voisins et les forêts d'arbres décisionnels. L'algorithme des k plus proches voisins est le plus simple et l'un des plus populaires, c'est sur lui que nous allons nous attarder.

L'algorithme des **k plus proches voisins**, également connu sous le nom d'algorithme **kNN** (*k - Nearest Neighbor*), est un algorithme non paramétrique qui classe les points de données en fonction de leur proximité et de leur association avec d'autres données disponibles. Cet algorithme suppose que des points de données similaires peuvent être trouvés à proximité les uns des autres. En conséquence, il cherche à calculer la distance entre les points de données, généralement au moyen de la distance euclidienne, puis attribue une catégorie en fonction de la catégorie la plus fréquente. Sa facilité d'utilisation et son faible temps de calcul en font un algorithme privilégié. Sur la figure 1.9 nous pouvons voir un exemple d'application de l'algorithme kNN dans le cadre d'une classification à deux catégories (bleue et rouge) et avec $k = 3$. Nous constatons que pour le point que nous voulons étiqueter, parmi ses 3 plus proches voisins, deux sont rouges et un seul est bleu. On en déduit donc que celui-ci appartient à la catégorie rouge.

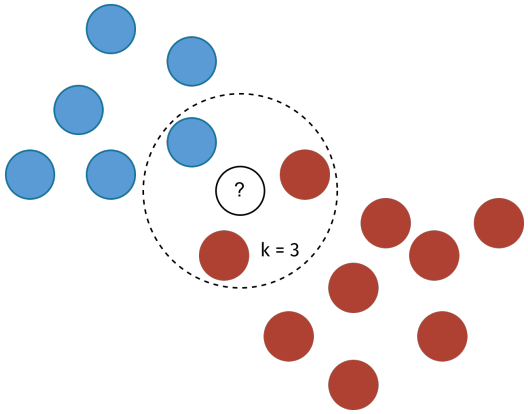


FIGURE 1.9 – Exemple d'application de l'algorithme kNN

1.7 Conclusion

Dans ce chapitre, nous avons préalablement présenté le cluster computing ainsi que les applications distribuées qui en découlent. Par la suite, une description générale du cloud computing a été effectuée, mettant en lumière le Function as a Service (FaaS) en tant que modèle spécifique au sein du cloud. Après, nous avons procédé à une exposition détaillée du fonctionnement de la mémoire cache, en abordant notamment le taux d'échec, la courbe du taux d'échec, ainsi que la technologie d'allocation de cache Intel CAT. Et enfin, nous avons terminé par une explication du mécanisme de classification par apprentissage supervisé, en mettant l'accent sur l'algorithme kNN. Ayant posé les bases théoriques de notre travail, nous allons dans le chapitre suivant présenter notre première contribution qui se situe dans le domaine du cloud computing.

Chapitre 2

CASY, un système d'allocation de cache CPU intelligent pour les fonction FaaS

Introduction

Aujourd'hui, avec l'avènement du Cloud Computing [17], nous assistons à l'émergence de nouveaux services sans serveur tel que le FaaS [3]. Le modèle FaaS permet aux développeurs de créer, d'exécuter et de gérer leurs applications comme un ensemble de fonctions sans état (*serverless*), sans avoir à maintenir leur propre infrastructure. Le fonctionnement du modèle FaaS est le suivant :

(1) Le développeur télécharge sa fonction écrite dans un langage de programmation pris en charge par la plateforme FaaS (par exemple, Java, Python, JavaScript, etc.) qui est ensuite enregistrée dans un registre (le registre est une sorte de base de données qui stocke les fonctions) ;

(2) Le développeur spécifie un ensemble d'événements (par exemple, de nouveaux enregistrements dans une base de données, la réception d'un message de la part d'un utilisateur, réception d'une requête spécifique, etc.) qui déclencheront l'exécution des fonctions ;

(3) à chaque fois qu'un événement se produit, la plateforme de FaaS exécute la fonction correspondante dans un environnement isolé (*sandbox*), qui peut être un conteneur tel que Docker [7] ou une machine virtuelle légère telle que Firecracker [8].

Une plateforme de FaaS peut être déployée et administrée de deux manières. La première consiste à faire appel aux fournisseurs de cloud

(par exemple Google Cloud Functions [18], Azure Functions [18], AWS Lambda [18], et IBM Cloud Functions [18]). La deuxième, dite locale, utilise un logiciel (par exemple OpenWhisk [19]) permettant d'administrer l'exécution des fonctions localement.

Dans le FaaS, la plupart des fonctions suivent le modèle d'exécution ETL (*Extract-Transform-Load*) [20], qui peut être décrit comme suit :

(E) la fonction lit les données d'entrée (par exemple une image, un enregistrement sonore) à partir d'un stockage persistant ;

(T) la fonction effectue des calculs sur les données (par exemple le traitement d'images, l'encodage audio) ;

(L) et enfin, la sortie de la fonction est stockée dans une mémoire persistante.

Le modèle ETL motive l'utilisation de *sandbox* pour les exécutions de fonctions afin d'avoir un environnement isolé qui contient toutes les dépendances nécessaires. Outre l'aspect "sans serveur" du FaaS, son attrait provient également de son système de facturation. Avec d'autres types de services cloud (IaaS, PaaS et SaaS), la facturation est basée sur le temps d'exécution de l'environnement dans lequel l'application est exécutée (par exemple, le temps d'exécution total de la machine virtuelle), alors qu'avec le FaaS, la facturation est basée sur le temps d'exécution de la fonction.

En effet, la facturation dans le FaaS est effectuée à chaque invocation de fonction et est basée sur le temps d'exécution total [21], fournissant ainsi un système de facturation à grain fin. Un temps d'exécution plus court pour une fonction implique un coût moindre. Le temps d'exécution est donc un élément clé du FaaS. Plusieurs travaux de recherche ont étudié des approches visant à réduire le temps d'exécution des fonctions. Beaucoup de ces travaux s'intéressent à l'environnement d'exécution, comme l'optimisation des temps de démarrage des conteneurs [22, 23], tandis que d'autres ont proposé des systèmes de cache élaborés pour les données afin de raccourcir l'étape de chargement (l'étape E dans ETL) avant l'exécution de la fonction [24, 20]. Dans ce travail, nous nous intéressons à l'optimisation du cache CPU dans le contexte de l'exécution FaaS.

La plupart des processeurs utilisent de la mémoire cache pour accélérer l'accès aux données et aux instructions. Les caches processeur sont des composants de mémoire rapide situés à proximité du cœur du processeur et contenant des entrées de mémoire fréquemment utilisées [9]. Lors de l'accès à un emplacement de mémoire, le processeur vérifie d'abord sa présence dans le cache et récupère les données directement si

elles sont présentes. Dans le cas contraire, le processeur doit accéder à ces données depuis la mémoire principale. Pour minimiser les défauts de cache, la plupart des processeurs modernes utilisent une hiérarchie de caches multiples. Les niveaux de cache les plus éloignés du cœur du processeur (L2 et L3) sont généralement plus lents que ceux qui sont plus proches du processeur, mais aussi beaucoup plus grands [9]. Les premiers niveaux de cache sont souvent locaux à chaque cœur de processeur ; le dernier niveau de cache, appelé à juste titre cache de dernier niveau (LLC), est souvent partagé par tous les cœurs sur le même nœud NUMA. Une utilisation abusive du LLC par une application s'exécutant sur un cœur peut conduire à la contention du cache. Dans le FaaS, à des fins de densité, plusieurs fonctions sont exécutées sur le même serveur, partageant ainsi le même LLC. Pour remédier à la contention du cache dans le LLC, Intel a introduit une nouvelle technologie connue sous le nom de *Cache Allocation Technology* [13, 14] (CAT en abrégé), qui fournit un contrôle programmable par logiciel sur la quantité de cache qui peut être utilisée par un processus, une machine virtuelle ou un conteneur. Cependant, aucune plateforme FaaS n'utilise cette technologie CAT pour allouer le cache du processeur aux fonctions. Dans ce travail, nous proposons un système appelé *CASY* (**C**PU cache **A**llocation **SY**stem), qui s'appuie sur la technologie CAT pour allouer le LLC aux fonctions dans une plateforme FaaS.

Notre solution *CASY* alloue le cache du processeur aux fonctions FaaS en fonction de leurs besoins. La construction d'un tel système soulève deux défis :

Défi n°1. Comment déterminer la quantité de cache CPU nécessaire à une fonction ? En effet, l'utilisation du cache d'une fonction dépend non seulement de son implémentation, mais aussi des données d'entrée. Par exemple, une fonction n'aura probablement pas les mêmes besoins en cache lorsqu'elle traite un fichier de 100 KBytes et un fichier de 200 MBytes. Cet élément rend l'allocation de cache plus complexe.

Défi n°2. Comment allouer le LLC aux fonctions en sachant qu'il s'agit d'une ressource limitée ? Il est très probable que la taille du LLC ne soit pas suffisante pour satisfaire les besoins en LLC de toutes les fonctions en cours d'exécution. Par conséquent, la question suivante se pose : comment allouer le LLC à ces fonctions afin d'obtenir l'exécution la plus rapide possible pour toutes les fonctions ?

Pour répondre à ces deux défis, *CASY* intègre deux composants : (1) *CASY ML*, qui est un module d'apprentissage automatique visant à construire les modèles capables de prédire les besoins en cache d'une fonction dépendamment des données d'entrée, et (2) *CASY Cache Allocator*, qui est le composant qui alloue le cache aux fonctions selon leurs besoins prédits en cache et du niveau de charge actuel du cache.

CASY Machine Learning (ML) s'appuie sur l'**apprentissage automatique supervisé** pour déterminer la quantité de cache nécessaire à une fonction selon les données d'entrée. Nous définissons deux types de fonctions selon leur utilisation de la mémoire cache : Les fonctions sensibles à la mémoire cache (CS) et les fonctions insensibles à la mémoire cache (CI). Les fonctions CS sont affectées par la contention de la mémoire cache du processeur, tandis que les fonctions CI ne subissent que peu ou pas d'impact sur les performances en raison de la contention de la mémoire cache. Il convient de noter qu'une fonction peut être à la fois CS et CI en fonction des données d'entrée. Ainsi, pour une fonction donnée avec un fichier de données d'entrée donné, le module ML permet de déterminer s'il s'agit d'une fonction CS ou CI et de calculer la quantité de LLC nécessaire pour la fonction. Pour ce faire, *CASY ML* construit des modèles ML pour chaque fonction. Nous avons remarqué dans notre étude et également sur la base de travaux antérieurs [20] que l'activité mémoire et cache d'une fonction dépend de la taille des données d'entrée. Ainsi, pour une fonction, ses modèles ont été construits en utilisant la taille des données comme variable d'entrée.

CASY Cache Allocator, ou encore l'allocateur de cache, est le module qui alloue le LLC à chaque fonction. Après que *CASY ML* ait calculé la quantité de LLC à allouer, le module d'allocation alloue le LLC aux fonctions selon leurs besoins et de l'occupation du LLC. Globalement, le cache est divisé en voies et ces voies peuvent être réparties entre les cœurs avec intel CAT. L'allocateur de cache *CASY* satisfait les fonctions en allouant le nombre requis de voies, tout en veillant à ce que la charge globale soit répartie de manière égale sur les voies du cache. Pour ce faire, l'allocateur de cache *CASY* attribue une **charge** à chaque voie de cache qui représente son occupation par les fonctions. Chaque fois qu'une nouvelle fonction est programmée, l'allocateur attribue à cette fonction les voies de cache les moins utilisées, ce qui permet de répartir la charge de manière égale sur toutes les voies de cache.

Nous avons mis en œuvre notre système et nous l'avons évalué de manière approfondie avec différentes fonctions, telles que les fonctions qui effectuent le traitement d'images et de vidéos. En résumé, les contributions de ce travail sont les suivantes :

— nous présentons *CASY*, un système d'allocation de cache CPU ciblant les plateformes FaaS qui utilise Intel CAT pour allouer le LLC aux fonctions dépendamment de la taille des données d'entrée ;

— *CASY* utilise des modèles ML pour prédire la quantité de cache CPU nécessaire à chaque fonction ;

— *CASY* met en œuvre une stratégie d'allocation qui distribue la charge sur les voies LLC et garantit de bonnes performances aux fonctions ;

— nous avons évalué *CASY* avec plusieurs fonctions, et nous avons observé une amélioration des performances allant jusqu'à 11% pour certaines fonctions sans dégrader les performances des autres fonctions.

La suite de ce chapitre est organisée de la façon suivante. La section 2.1 présente nos observations et nos motivations concernant la création de *CASY*. Les sections 2.2 et 2.3 présentent la conception et la mise en œuvre de *CASY*. La section 2.4 présente l'évaluation des résultats. La dernière section sera la conclusion de ce chapitre.

2.1 L'allocation de cache dans les environnements de FaaS

Pour motiver le besoin d'allocation du cache CPU dans les environnements FaaS, nous avons effectué deux évaluations : l'une sur l'impact de la contention de cache sur le temps d'exécution des fonctions et l'autre sur l'impact de la restriction du LLC à l'aide d'Intel CAT. L'environnement expérimental est le même que celui décrit dans la section 2.4, à savoir un environnement à deux 2 processeurs Intel(R) Xeon(R) Silver 4210R CPU, chacun ayant un LLC de 13.75 Mo avec 11 voies, et 64 Go de mémoire RAM ; les fonctions utilisées également décrites dans cette section sont : Effect, Blur, Speech Recognition et Resize.

Nous avons exécuté ces quatre fonctions dans deux scénarios, et à chaque fois, nous avons relevé leur temps d'exécution. Tout d'abord, chaque fonction est exécutée seule sur le serveur ; ensuite, les quatre fonctions sont exécutées simultanément sur le même nœud NUMA. Pour éviter toute surcharge du système, les environnements d'exécution des fonctions sont assignés à des cœurs individuels. Les tailles de fichier d'entrée sont de 1.4 Mo pour Blur, 100 Mo pour Effect, 127 Mo pour Resize et 79 Mo pour Speech Recognition. Les résultats obtenus sont

présentés dans la figure 2.1. Nous pouvons remarquer que le temps d'exécution de deux fonctions : Blur et Speech Recognition diminuent respectivement de 37% et de 16%, tandis que les autres fonctions sont moins affectées. Comme nous le verrons plus tard, cette variation de performance est causée par la contention du cache par les deux autres fonctions Effect et Resize. Cela motive le besoin d'un système qui permet d'isoler les fonctions les unes des autres.

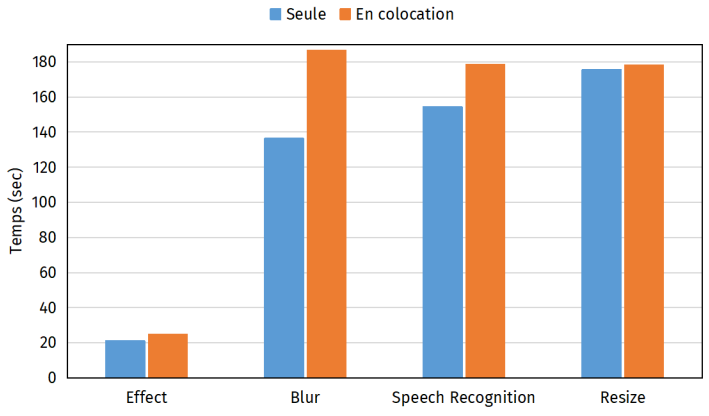


FIGURE 2.1 – Temps d'exécution de 4 fonctions FaaS fonctionnant seules et en colocation.

Nous avons également exécuté nos fonctions individuellement avec différentes tailles de données d'entrée tout en utilisant Intel CAT pour allouer différentes quantités de cache. Les résultats obtenus sont présentés dans la figure 2.2. Sur chaque graphe, chaque courbe représente le profil de la fonction pour une taille de fichier d'entrée, exprimée relativement à celle du fichier initial. En vert, le fichier d'entrée correspond à la totalité du fichier initial. En rouge, il représente les trois quarts du fichier initial, en orange la moitié, et en bleu le quart. On rappelle que les tailles de fichier initiales sont de 1.4 Mo pour Blur, 100 Mo pour Effect, 127 Mo pour Resize et 79 Mo pour Speech Recognition.

Nous pouvons observer que pour Blur, lorsque la taille d'entrée est maximale (taille = T), la fonction est affectée par la quantité de cache allouée ; mais lorsque la taille d'entrée est réduite (taille $\leq$ 3T/4), la fonction devient moins sensible à l'allocation de cache. En ce qui concerne Speech Recognition, cette fonction reste sensible à la mémoire cache quelle que soit la taille des données. À l'inverse, les fonctions Effect et Resize sont insensibles à l'allocation de la mémoire cache, quelle que soit la taille des données.

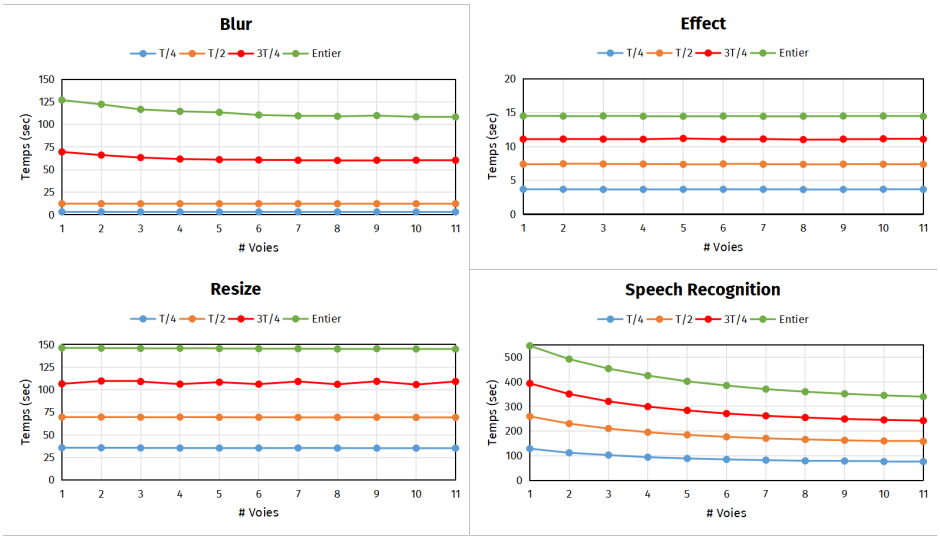


FIGURE 2.2 – Impact de l'allocation du cache avec CAT sur le temps d'exécution de la fonction pour différentes tailles de fichiers d'entrée.

Les observations que nous pouvons faire à partir de ces évaluations sont les suivantes :

(1) la contention du cache peut avoir un impact sur le temps d'exécution des fonctions ;

(2) l'utilisation du cache par les fonctions dépend des données d'entrée

(3) une fonction peut être sensible au cache avec certaines données d'entrée, tout en étant insensible avec d'autres. Ce comportement rend l'allocation de la mémoire cache CPU plus complexe.

Il semble donc intéressant de proposer un système qui permette d'allouer le cache du CPU aux fonctions, et ce système devrait être facilement intégrable aux plateformes FaaS.

Comme nous pouvons le constater, ces quatre applications présentent des utilisations de la mémoire cache qui couvrent toutes les catégories. En effet, nous avons les applications insensibles à la mémoire cache, mais qui en consomment peu, représentées par Effect. Ensuite, il y a les applications également insensibles à la mémoire cache, mais qui génèrent de la pollution, comme Resize. Enfin, nous avons les applications plus ou moins sensibles à la mémoire cache, représentées par Blur et Speech Recognition. Ainsi, ces quatre applications reflètent un large spectre des besoins en mémoire cache des applications FaaS.

2.2 Contribution

Pour répondre au problème de l'allocation du cache CPU dans les environnements de FaaS, nous proposons *CASY (**C**ache **A**llocation **SY**stem)*, un système qui s'appuie sur le *Machine Learning* pour profiler le comportement cache des fonctions et qui utilise ensuite le modèle d'apprentissage automatique supervisé pour prédire et allouer le cache processeur à l'aide de la technologie Intel CAT. La figure 2.3 illustre l'architecture globale de notre système et le flux d'interaction avec une plateforme FaaS.

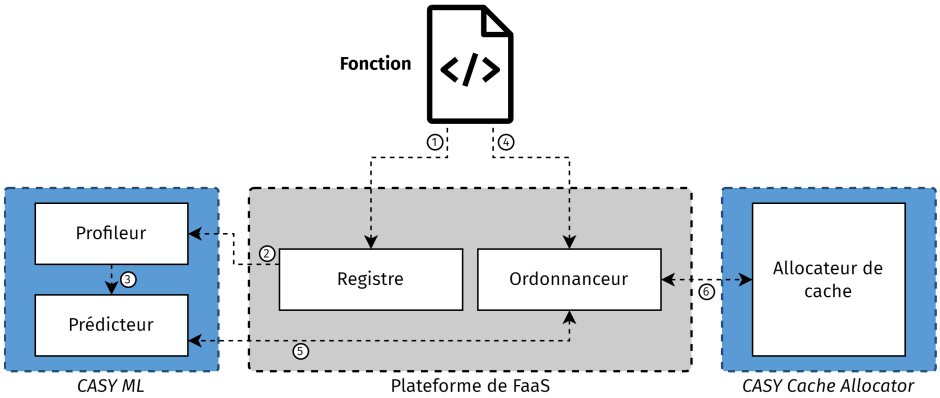


FIGURE 2.3 – Architecture générale de *CASY*, y compris ses composants et son interaction avec une plateforme FaaS.

2.2.1 Vue d'ensemble

La figure 2.3 montre l'architecture générale de *CASY*, qui se compose de deux composants principaux (encadrés bleus dans la figure 2.3) : *CASY ML* et *CASY Cache Allocator*. Le composant ML réalise la partie apprentissage automatique, c'est-à-dire l'apprentissage et la prédiction de la quantité de LLC nécessaire à une fonction pour des données d'entrée données. À cette fin, il se compose de deux éléments : le *profileur* et le *prédicteur*. L'objectif du profileur est de construire des modèles ML pour chaque fonction. Chaque fois qu'une nouvelle fonction est stockée dans le registre/la base de données (étape ❶), le profileur récupère les fonctions et les exécute avec différentes tailles de données (étape ❷) sur un serveur dédié. Ces différentes exécutions permettent de construire des modèles ML pour la fonction ; nos modèles utilisent la taille des données comme variable d'entrée. Cette opération est réalisée hors ligne (*offline*),

c'est-à-dire sans que les événements qui déclenchent la fonction ne se produisent. Une fois les modèles construits, ils sont transmis au prédicteur (étape ❸) pour une allocation future. Chaque fois qu'une fonction est invoquée (étape ❹), le planificateur de la plateforme FaaS s'adresse au prédicteur pour lui fournir la taille de cache requise (étape ❺). Une fois la prédiction effectuée, l'ordonnanceur s'adresse à l'allocateur de cache *CASY* pour configurer l'environnement d'exécution (*sandbox*) de la fonction (étape ❻). L'allocation du cache est effectuée en tenant compte des exigences de la fonction ainsi que de la charge actuelle du cache. Dans les sections suivantes, nous détaillons le fonctionnement de chaque composant de *CASY*.

2.2.2 *CASY ML*

Avant de présenter le module ML, il est nécessaire de rappeler notre classification des fonctions en ce qui concerne l'utilisation de la mémoire cache. Comme nous l'avons indiqué dans l'introduction de ce chapitre, nous avons défini deux classes de fonctions, *CI* et *CS* :

(a) *CS* pour les fonctions sensibles à la contention du cache. Il s'agit de fonctions pour lesquelles la contention de la mémoire cache a un impact substantiel sur leur temps d'exécution. Leurs données tiennent dans la mémoire cache LLC et elles causent très peu de défaut de cache.

(b) *CI* pour les fonctions insensibles à la contention de la mémoire cache. Dans cette classe, nous distinguons deux types de fonctions : celles dont les données ne tiennent pas dans le cache ou ne sont jamais réutilisées, et qui polluent donc les charges de travail voisines (fonctions *cache-thrashing*), ainsi que celles qui utilisent peu de cache et ne sont donc pas affectées par la contention du cache.

Il est important de noter qu'une fonction peut-être *CI* ou *CS* en fonction des données d'entrée. La classe d'une fonction est donc définie en fonction des données d'entrée. Ces deux classes seront utiles pour l'attribution des voies de cache. L'objectif du profileur est double :

1. construire un modèle capable de prédire le nombre de voies de cache requis par une fonction avec des données d'entrée fournies ;
2. et construire un modèle capable de spécifier si une fonction est *CI* ou *CS*.

En ce qui concerne le premier objectif, le profileur construit un modèle ML à partir des exécutions antérieures de la fonction avec divers

ensembles de données sur un serveur dédié. C'est ce que nous appelons *profilage de fonction*. Le profilage de la fonction est effectué une fois et doit être réalisé sur un serveur libre afin d'éviter la contention du cache. Nous supposons qu'une fonction est généralement invoquée fréquemment et sur une longue période. Par conséquent, le coût du profilage est minime par rapport au gain à long terme.

Dans *CASY*, nous supposons que chaque fois qu'une fonction est enregistrée, le client peut soit fournir un ensemble de données de test sur lequel nous pouvons former et construire le modèle, soit fournir le type de données d'entrée afin que *CASY* génère les données nécessaires. Nous avons observé dans nos résultats et dans les travaux précédents [20] que la taille du fichier d'entrée est une propriété suffisante pour déterminer le comportement d'une fonction vis-à-vis du cache CPU. Le profilage de la fonction est effectué en suivant l'algorithme 1. Il s'agit d'exécuter la fonction avec différentes tailles de fichier (lignes 12 et 13 de l'algorithme 1) tout en réduisant le nombre de voies de cache allouées (lignes 11 et 12). L'objectif est d'identifier, pour chaque taille de fichier, le nombre minimal de voies qui permet d'obtenir le même niveau de performance que lorsque la fonction a un accès complet au cache (ligne 5). Pour ce faire, nous définissons un seuil (5% dans ce travail) qui représente le taux de dégradation à partir duquel nous considérons qu'il y a un impact sur le temps d'exécution. Pour une fonction avec un fichier d'entrée donné, nous réduisons progressivement le nombre de voies de cache allouées et définissons le plus petit nombre de voies de cache possible tout en restant en dessous du seuil de performance. À la ligne 14, nous calculons la dégradation (nommée Δ dans l'algorithme) que nous comparons à notre seuil de la ligne 10.

Une fois cette étape d'exécution réalisée, nous disposons pour chaque fonction d'une paire de valeurs $(size, nr)$, où *size* est la taille du fichier d'entrée et *nr* le nombre minimal de voies de cache associé. Ensuite, nous utilisons ces paires de valeurs pour construire un modèle ML. Notre modèle ML utilise la taille du fichier comme variable d'entrée et renvoie le nombre de voies de cache.

Le deuxième objectif du profileur est de construire un modèle capable de prédire la classe d'une fonction en fonction de la taille des données d'entrée. Pour ce faire, le profileur utilise la sensibilité au cache d'une fonction. **Nous définissons la sensibilité S d'une fonction f avec une donnée d'entrée particulière d comme la dégradation des performances lorsque la fonction n'a accès qu'à une seule voie du cache par rapport à lorsqu'elle a un accès complet à l'ensemble du LLC.** L'équation 2.1 présente la formule de la sensibilité $S_{f(d)}$ d'une fonction f, pour une taille

Algorithm 1 Algorithme de profilage de fonction

1: $f \leftarrow$ RECUPERER_FONCTION( )
2: $L_{\text{fichiers}} \leftarrow$ RECUPERER_FICHIERS_D_ENTREE( )
3: $N_{\text{voies}} \leftarrow$ RECUPERER_NOMBRE_DE_VOIES_MAX( )
4: **for** fichier **in** L_{fichiers} **do**
5: ALLOUER(N_{voies})
6: $\Delta \leftarrow 0$
7: $T_{\text{Ref}} \leftarrow$ TEMPS_D_EXECUTION(f, fichier)
8: taille $\leftarrow$ TAILLE_DE(fichier)
9: $nb \leftarrow N_{\text{voies}} + 1$
10: **while** $\Delta <$ Seuil **and** $nb > 1$ **do**
11: $nb \leftarrow nb$ - 1
12: ALLOUER(nb)
13: $T_{\text{actuel}} \leftarrow$ TEMPS_D_EXECUTION(f, fichier)
14: $\Delta \leftarrow \frac{T_{\text{Ref}} - T_{\text{actuel}}}{T_{\text{Ref}}} \times 100$
15: **end while**
16: AFFECTER_NOMBRE_DE_VOIES(taille, $nb + 1$)
17: **end for**

de données d'entrée d, avec $T^{N_{max}}_{f(d)}$ et $T^{1}_{f(d)}$ représentant respectivement son temps d'exécution avec le LLC entier et une seule voie de cache.

$$S_{f(d)} = \frac{T^{N_{\text{max}}}_{f(d)} - T^{1}_{f(d)}}{T^{N_{\text{max}}}_{f(d)}} \times 100 \tag{2.1}$$

Une fonction est dite *CI* si cette sensibilité est supérieure à un seuil (nous considérons que 5% est une dégradation assez importante), et CS dans le cas contraire. Pour chaque fonction, les résultats de différentes exécutions avec diverses données sont utilisés pour construire un modèle ML qui prendra comme entrée la taille des données et les classera comme *CI* ou *CS*.

2.2.3 *CASY Cache allocator*

Comme nous l'avons mentionné dans la section 1.5, l'architecture du processeur a un nombre limité de CLOS (classes de services) qui peuvent être définies et un nombre de voies LLC qui peuvent être assignées. Par conséquent, il est important de configurer correctement ces CLOS. Sur tous les serveurs, nous définissons de manière statique la configuration

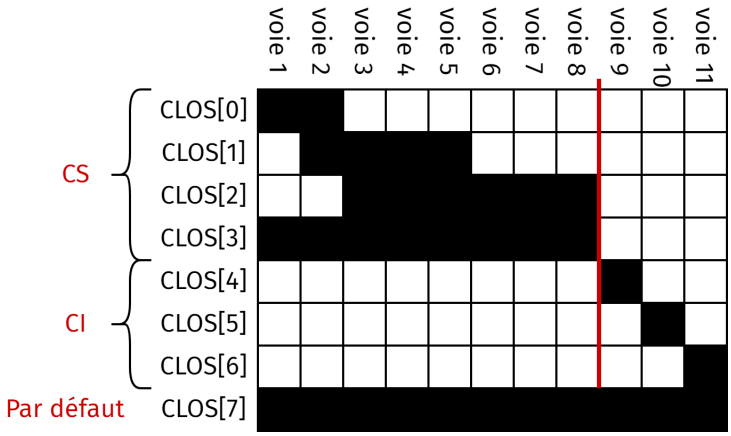


FIGURE 2.4 – Exemple de configuration pour un processeur ayant 8 CLOS et un LLC à 11 voies.

des CLOS et les modes de cache associés. Nous divisons nos CLOS en deux parties : l'une pour les fonctions *CI* et l'autre pour les fonctions *CS*. Appelons N_{voies} le nombre total de voies de cache disponibles pour un processeur et N_{CLOS} le nombre de CLOS pouvant être définis. Nous divisons N_{voies} en deux parties : N_{CI} et N_{CS} donc $N_{voies} = N_{CI} + N_{CS}$. Cette configuration vise à séparer les fonctions *CI* et les fonctions *CS* afin de réduire la contention du cache. Le reste de la configuration concerne l'attribution de voies aux CLOS.

— Pour les voies N_{CI}, ils sont configurés en CLOS à voie unique. Comme ces fonctions ne sont pas affectées par le nombre de voies de cache allouées, leurs environnements d'exécution seront configurés avec une seule voie.

— Les N_{CS} voies, nous configurons les CLOS associés en suivant deux règles :

1) les voies de cache seront allouées au plus petit nombre possible de CLOS ;
2) et les CLOS couvriront autant de combinaisons que possible avec les voies disponibles.

— Finalement, un *CLOS par défaut*, qui contient toutes les voies de cache, sera défini pour les fonctions qui n'ont pas de modèles ML.

Pour illustrer cela, considérons une LLC à 11 voies pour une architecture de processeur sur laquelle nous pouvons configurer 8 CLOS comme le montre la figure 2.4. Ainsi, si nous décidons de définir trois voies pour les fonctions *CI* et huit pour les fonctions *CS*, nous obtenons la

configuration illustrée dans la figure 2.4. Chaque CLOS pour les fonctions *CI* a une seule voie (voir figure 2.4). Comme pour le CLOS pour les fonctions *CS*, nous disposons de huit voies de cache restantes. Nous configurons le CLOS de deux à huit voies, une voie étant utilisée dans un maximum de trois CLOS. Enfin, le *CLOS 7* (le CLOS par défaut) qui comprend toutes les voies de cache sera utilisé pour les fonctions qui n'ont pas de modèles ML.

L'allocation des CLOS aux fonctions se fait sur la base de la charge sur le CLOS et du nombre prédit de voies requises pour la fonction. Avant de présenter l'heuristique d'allocation, il est important de définir trois mesures : le poids d'une fonction, la charge d'une voie de cache et la charge d'un CLOS. Nous définissons le poids W d'une fonction f avec des données d par rapport à un CLOS de N_{CLOS} voies dans l'équation 2.2. $P_{f(d)}$ est le nombre de voies fournies par le prédicteur.

$$W_{f(d)} = \frac{P_{f(d)}}{N_{\text{CLOS}}} \tag{2.2}$$

La charge L_{voie} sur une voie de cache est définie comme la somme de tous les poids de fonction qui utilisent cette voie. L'équation 2.3 présente la formule de calcul de la charge d'une voie.

$$L_{\text{voie}} = \sum_{i=1}^{nb_{\text{Fonc}}} W_{f_i(d)} \tag{2.3}$$

La charge L_{CLOS} d'un CLOS est définie dans l'équation 2.4 et correspond à la somme des charges des voies de cache affectées au CLOS.

$$L_{\text{CLOS}} = \sum_{i=1}^{nb_{\text{Voie}}} L_{\text{voie}_i} \tag{2.4}$$

L'algorithme 2 présente notre heuristique d'allocation. À chaque fois qu'une fonction est invoquée, l'ordonnanceur interroge le prédicteur en indiquant la fonction et la taille des données. Deux cas sont alors possibles :

(1) Les modèles associés à la fonction ne sont pas disponibles, de sorte que l'environnement d'exécution est simplement défini sur la valeur par défaut CLOS (ligne 5) ;

(2) Les modèles existent et le prédicteur fournit le nombre de voies nécessaires et la classe de la fonction (ligne 3). C'est le cas intéressant que nous détaillons dans le paragraphe suivant.

Si la fonction est une *CI*, alors son environnement d'exécution sera configuré avec un CLOS dédié à ces fonctions (ligne 9) ; sinon, nous utilisons les CLOS réservés aux *CS* (lignes 11). Nous calculons ensuite la nouvelle charge potentielle sur toutes les voies de cache avec les nouveaux poids des fonctions (ligne 14), et nous trions la liste des CLOS en fonction de cette nouvelle charge potentielle (ligne 16). Nous nous assurons ensuite que s'il y a plusieurs CLOS avec une charge inférieure à 1, alors parmi ces CLOS celui avec la charge la plus élevée est choisi, afin d'éviter de choisir un CLOS avec plus de voies que nécessaire (lignes 18 à 21). Dans le cas contraire, le CLOS sélectionné sera celui dont la charge est la plus faible (ligne 23). À la fin de l'exécution de la fonction, les charges des CLOS et des voies sont mises à jour.

Algorithm 2 Obtenir le CLOS approprié pour une fonction

1: $f \leftarrow$ RECUPERER_FONCTION( )
2: taille $\leftarrow$ RECUPERER_TAILLE_DU_FICHIER_D_ENTREE( )
3: $P_f, C_f \leftarrow$ PREDIRE_NOMBRE_DE_VOIES_ET_CLASSE(f , taille)
4: **if** P_f == null **then**
5: S_{CLOS} = RECUPERER_CLOS_PAR_DEFAUT( )
6: **return** S_{CLOS}
7: **end if**
8: **if** C_f == "CI" **then**
9: $\text{liste}_{\text{clos}} \leftarrow$ RECUPERER_LISTE_DE_CLOS(“CI”)
10: **else** ▷ C_f == "CS"
11: $\text{liste}_{\text{clos}} \leftarrow$ RECUPERER_LISTE_DE_CLOS(“CS”)
12: **end if**
13: $\text{liste}_{\text{voies}} \leftarrow$ RECUPERER_VOIES($\text{liste}_{\text{clos}}$)
14: $\text{liste}_{\text{voies}} \leftarrow$ AJOUTER_CHARGE_DE_VOIE($\text{liste}_{\text{voies}}$, f , P_f)
15: $\text{liste}_{\text{clos}} \leftarrow$ METTRE_A_JOUR_CHARGES_DE_CLOS($\text{liste}_{\text{clos}}$, $\text{liste}_{\text{voies}}$)
16: $\text{liste}_{\text{clos}} \leftarrow$ TRIER_PAR_CHARGE($\text{liste}_{\text{clos}}$)
17: **for** clos **in** $\text{liste}_{\text{clos}}$ **do** ▷ $\text{liste}_{\text{CLOS}}$ est trié
18: **if** CHARGE(clos) ≤ 1 **then**
19: $S_{\text{CLOS}} \leftarrow$ clos
20: **end if**
21: **end for**
22: **if** S_{CLOS}== "Null" **then**
23: $S_{\text{CLOS}} \leftarrow \text{liste}_{\text{clos}}[0]$
24: **end if**

2.3 Mise en œuvre de *CASY*

Le principal défi de la mise en œuvre d'*CASY* est d'offrir un système qui peut être facilement intégré aux plateformes FaaS. C'est pourquoi nous avons conçu *CASY* comme un système autonome qui peut s'interfacer avec les plateformes FaaS. *CASY* est entièrement implémenté en Python. Le reste de cette section se concentre sur la mise en œuvre des deux composants de *CASY* et à l'intégration dans Apache OpenWhisk.

2.3.1 *CASY ML*

Ce composant se compose de deux éléments : le **profileur** et le **prédicteur**. Le profileur extrait des fonctions du registre ou de la base de données (Apache CouchDB dans le cas d'Open Whisk) et les exécute avec diverses données d'entrée. Le profilage est effectué en exécutant les fonctions sur un serveur dédié, le tout hors ligne. Nous supposons qu'un serveur sera dédié au profilage des fonctions en cas de besoin, ou un nœud NUMA sur le serveur. Les données de profilage peuvent être fournies par l'utilisateur ou par *CASY*. *CASY* dispose actuellement de vidéos et d'images qui peuvent être utilisées pour entraîner les fonctions. En ce qui concerne l'apprentissage automatique, nous avons utilisé comme modèle de classification, le modèle des *k - plus proches voisins* (en abrégé *k-NN* pour *k-nearest neighbors*), avec la bibliothèque Scikit-learn. Ce choix est justifié par le fait que pour les modèles à construire par fonction, il s'agit de prédiction vers un univer discret : de 1 au nombre maximum de voies. Il convient de noter qu'avec ce modèle d'apprentissage automatique (comme avec presque tous les modèles d'apprentissage automatique), plus l'ensemble de données d'apprentissage est diversifié, plus le modèle sera précis. La section d'évaluation montre que dans notre cas, un ensemble de données d'apprentissage relativement restreint nous permet d'obtenir un bon niveau de précision avec k-NN.

2.3.2 *CASY Cache Allocator*

Ce composant est responsable de la configuration de l'environnement d'exécution. Actuellement, l'implémentation que nous avons utilisé l'outil Intel pqos [25] pour configurer les CLOS et rdtset [25] pour associer un cœur de processeur à un CLOS. Cette bibliothèque permet de définir les CLOS, les voies qui leur sont associées, et également de lier les cœurs aux CLOS. Ainsi, sur tous les serveurs, nous utilisons ces bibliothèques

pour configurer tous les CLOS et ensuite, lors de l'invocation d'une fonction, *CASY Cache Allocator* ajoute les cœurs sélectionnés pour exécuter une fonction au CLOS désigné par son algorithme. La version actuelle de *CASY* permet de configurer les CLOS pour les conteneurs.

2.3.3 Intégration dans Apache OpenWhisk

Apache OpenWhisk est un gestionnaire de FaaS open source très populaire. Nous décrirons l'intégration de *CASY* dans ce gestionnaire FaaS. Les principaux composants d'OpenWhisk tels que présentés sur la figure 2.5 sont les suivants :

— un serveur **Nginx** qui est le point d'entrée pour les demandes/événements ;

— un **contrôleur** qui traduit les demandes en invocations de fonctions, et inclut un équilibreur de charge qui choisit sur quel serveur la fonction sera exécutée. L'équilibreur de charge définit également les paramètres de configuration de l'environnement d'exécution, c'est-à-dire l'épinglage sur un cœur du processeur et la quantité de mémoire ;

— un serveur **Kafka** qui sert d'outil de communication entre le contrôleur et les invocateurs sur les serveurs ;

— un serveur de base de données **CoucheDB** (le registre) qui contient les codes sources et les résultats des fonctions ;

— **l'invoqueur** qui récupère le code source de la fonction dans la base de données, lance un conteneur, injecte le code source dans le conteneur, démarre l'exécution de la fonction dans ce conteneur, récupère le résultat une fois l'exécution terminée, et enregistre ces résultats dans la base de données avant de détruire le conteneur.

L'intégration de *CASY* dans un tel système est assez simple, car *CASY ML* interagit avec CoucheDB pour extraire les fonctions, les exécuter et enregistrer les données de formation. Au niveau du contrôleur, puisque c'est l'équilibreur de charge qui décide et configure l'environnement d'exécution, nous avons modifié la communication d'invocation pour savoir quelles fonctions doivent être exécutées et quelle est la taille des données d'entrée. Nous pouvons ainsi utiliser le prédicteur pour déterminer la classe et le nombre de voies nécessaires. Ensuite, sur la base des paramètres fournis par l'équilibreur de charge, c'est-à-dire le choix des cœurs sur lesquels le conteneur va être épinglé, *CASY Cache allocator* configure l'affinité de ces cœurs avec le CLOS désigné par son algorithme d'allocation.

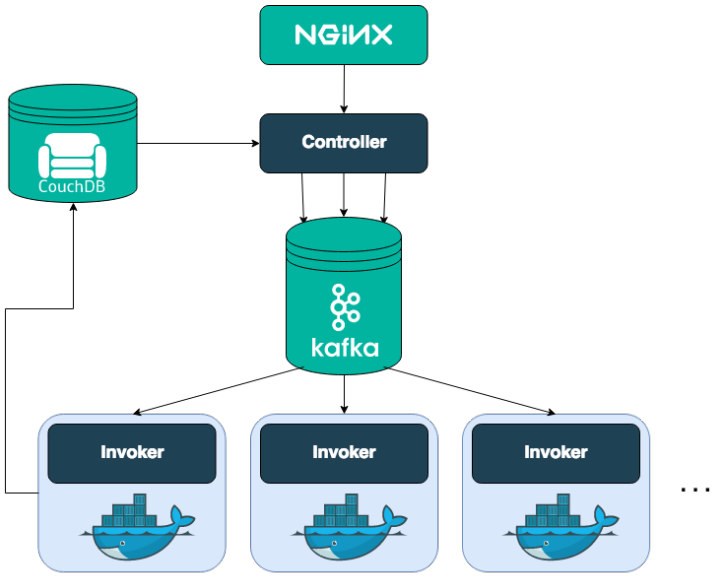


FIGURE 2.5 – Flux de traitement interne d'OpenWhisk

2.4 Évaluation

Dans cette section, nous présentons l'évaluation des performances de *CASY*. Ces évaluations nous permettront de répondre aux questions suivantes.

(a) Quelle est la précision du module *CASY ML* ?

(b) Quelle est l'amélioration du temps d'exécution apportée par *CASY* ?

(c) Quel est le surcoût induit par *CASY* ?

Par conséquent, pour évaluer *CASY*, nous avons procédé en deux étapes principales : La validation de notre modèle d'apprentissage et l'évaluation de notre système d'allocation.

2.4.1 Environnement expérimental

Matériel

Nous avons évalué *CASY* sur un serveur présentant les caractéristiques décrites dans le tableau 2.1. Le processeur du serveur est doté de la technologie CAT et permet la configuration de 8 CLOS. Nous avons configuré les CLOS comme dans la section 2.2.

Composants		Caractéristiques
CPU	Cœurs	Intel(R) Xeon(R) Silver 4210R CPU
	Cache niveau 1	20×32KB, 8-voies
	Cache niveau 2	20×1MB, 16-voies
	Cache niveau 3	Partagé, 2×13.75MB, 11-voies
Mémoire		64 GBytes
Ethernet		Intel X710 10GbE
Stockage		1.0 TB HDD
Système d'exploitation		Ubuntu 20.04.3, Linux kernel 5.4.0

TABLE 2.1 – configuration matérielle.

Applications

Nous avons évalué *CASY* en utilisant plusieurs fonctions. Le tableau 2.2 fournit la liste des fonctions utilisées dans nos expériences et leurs descriptions. Nous avons utilisé les mêmes fonctions déployées que dans les documents traitant de la performance dans FaaS [20].

Fonction	Description
Blur	Prend en entrée un fichier image PNG et applique un flou
Effect	Prend en entrée un fichier audio au format WAV et applique un effet sonore
Resize	Prend une vidéo en haute résolution et renvoie une vidéo en basse résolution
Speech Recognition	Prend un fichier audio et effectue des reconnaissances vocales

TABLE 2.2 – Descriptions des fonctions.

2.4.2 Validation du modèle d'apprentissage

Cette étape consiste à vérifier la précision de notre modèle de prédiction. Pour chaque fonction spécifiée ci-dessus (dans le tableau 2.2), nous avons effectué un entraînement avec un ensemble de fichiers, 4 fichiers dont les caractéristiques sont listées dans le tableau 2.3, puis nous avons évalué le modèle de prédiction avec 10 fichiers différents.

Le tableau 2.3 présente la taille des fichiers utilisés pour chaque fonction ainsi que le nombre de voies détectées par notre algorithme de

profilage pour ces différentes tailles. Nous pouvons observer que les fonctions Effect et Resize nécessitent toujours 1 voie de cache. Ce sont des fonctions *CI* alors que les fonctions Blur et Speech Recognition voient leur besoin en cache varier en fonction de la taille du fichier d'entrée. Plus particulièrement, la fonction Blur voit son besoin en nombre de voies augmenter de 1 à 7 alors que la fonction Speech Recognition passe à 4.

Effect		**Blur**	
Taille	# Voies	Taille	# Voies
26 MB	1	124 KB	1
51 MB	1	361 KB	1
76 MB	1	699 KB	4
100 MB	1	1.4 MB	7
Speech Recognition		**Resize**	
Taille	# Voies	Taille	# Voies
40 MB	10	31 MB	1
79 MB	11	61 MB	1
118 MB	11	92 MB	1
157 MB	11	127 MB	1

TABLE 2.3 – Quantité de cache nécessaire pour chaque taille de fichier d'entraînement pour nos benchmarks

Une fois les modèles construits, l'étape suivante consiste à les valider. Pour valider nos modèles, nous avons utilisé le modèle pour prédire le nombre de voies de cache nécessaires avec différentes tailles de fichiers et nous avons comparé les résultats à une exécution avec un accès complet au LLC. La figure 2.6 présente les résultats obtenus. Chaque barre représente une taille de fichier utilisée et le temps d'exécution obtenu lorsque la fonction a un accès total au cache et lorsqu'elle a accès à la quantité prédite comme nécessaire. Le nombre au-dessus de chaque barre représente le nombre de voies prédit par les modèles.

Nous remarquons que pour toutes les fonctions, les temps d'exécution avec *CASY* et avec tout le cache sont identiques. Cela signifie que le nombre de voies de cache prédit par *CASY* semble correspondre aux besoins réels des applications. En outre, pour toutes les fonctions, le modèle a toujours été en mesure de déterminer la classe d'une fonction, c'est-à-dire que Blur et Speech Recognition sont *CS* et Effect et Resize sont CI.

Nous observons également une non-proportionnalité entre certaines tailles de fichiers et le temps d'exécution pour les charges de Blur et de

Speech Recognition. Cela peut s'expliquer simplement par le fait que certains fichiers de grande taille ont une résolution plus faible et induisent donc un temps de traitement plus court.

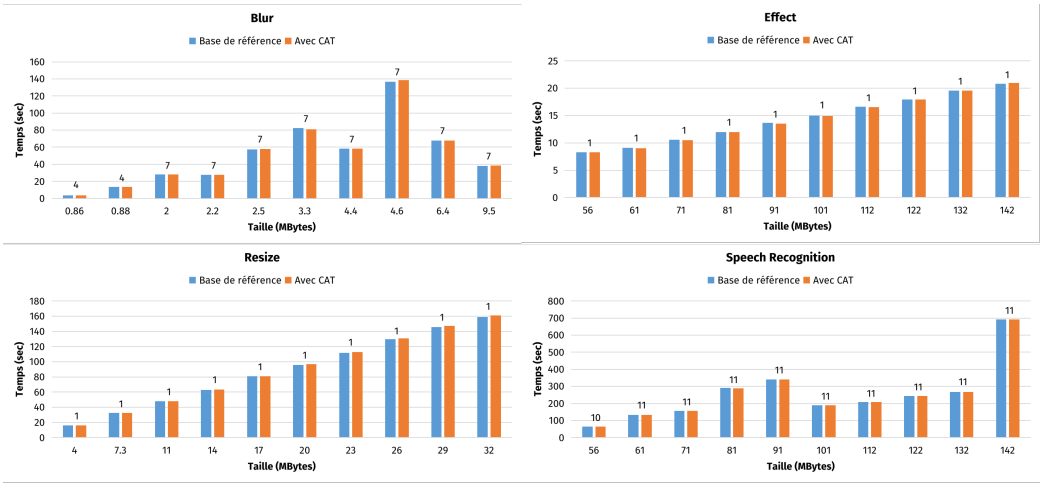


FIGURE 2.6 – Évaluations du modèle d'apprentissage automatique avec toutes les fonctions en utilisant différentes tailles de fichiers d'entrée.

2.4.3 Évaluation de l'allocation

Pour l'évaluation de notre algorithme d'allocation, il était nécessaire de rejouer les scénarios d'exécution des fonctions d'un centre de données. Chaque scenario étant caractérisé par les dates de démarrage et les fichiers d'entrée de différentes fonctions. Comme nous ne disposions pas de ces données, pour générer des scénarios d'exécution, nous avons considéré le déclenchement de l'exécution des fonctions comme un processus de poisson, et mis en œuvre un émulateur qui suit une distribution de probabilité, la **distribution exponentielle**. En d'autres termes, nous avons utilisé la distribution exponentielle pour générer la date de début des fonctions et choisir au hasard la fonction à exécuter et le fichier à utiliser. Ainsi, la formule suivante permet de calculer l'intervalle de temps (en secondes) qui sépare deux dates de lancements successifs de fonctions :

$$\Delta T = -\frac{\ln U}{\lambda} \tag{2.5}$$

Où U est un nombre aléatoire tiré de la distribution uniforme dans l'intervalle $[0,1]$, et λ l'intensité qui est en fait le nombre moyen de

fonctions déclenchées par seconde. Pour nos évaluations, nous avons choisi $\lambda = 2$.

Le tableau 2.4 montre les différentes tailles de données d'entrée utilisées avec chaque fonction. Nous présentons également dans le tableau l'acronyme que nous utilisons sur nos figures. Ainsi, nous avons utilisé notre générateur pour générer des scénarios impliquant 6, 10 et 20 fonctions s'exécutant simultanément avec et sans *CASY*. Chaque scénario est exécuté 10 fois et les résultats présentés sont les temps d'exécution moyen des fonctions. Les figures 2.7, 2.8 et 2.9 montrent les résultats de certains de ces scénarios.

Acronyme	**Configuration**
Blur1	Blur avec une taille de fichier de 4.6 MB
Blur2	Blur avec une taille de fichier de 6.4 MB
Blur3	Blur avec une taille de fichier de 9.5 MB
Resize1	Resize avec une taille de fichier de 37 MBs
Resize2	Resize avec une taille de fichier de 67 MB
Resize3	Resize avec une taille de fichier de 127 MB
Speech_Rec1	Speech Recognition avec une taille de fichier de 79 MB
Effect1	Effect avec une taille de fichier de 100 MB

TABLE 2.4 – Fonction avec les tailles de fichier et l'acronyme associés

Commençons notre analyse par l'exécution de 6 fonctions (figure 2.7). Nous pouvons remarquer que Blur est celle qui montre une amélioration avec notre solution. Cette amélioration est d'environ 11% (carrés rouges sur la figure). De plus, nous pouvons également observer que Resize, qui est exécuté avec Blur, n'est pas affecté par *CASY*. Cela montre que *CASY* a été en mesure d'identifier que Resize est une fonction *CI* et d'effectuer une allocation en conséquence.

Poursuivons notre analyse avec l'exécution de 10 fonctions (figure 2.8). Nous pouvons voir que la fonction qui a le plus d'impact est toujours Blur (7%), suivie de Speech Recognition (5%). Comme nous l'avons noté dans la section précédente, Speech Recognition nécessite 11 voies de cache, ce qui correspond au nombre total disponible sur notre serveur. Ainsi, l'impact de notre système d'allocation reste limité sur ce benchmark.

Enfin, le dernier scénario concerne l'exécution de 20 fonctions simultanément (figure 2.9). Nous remarquons qu'il y a peu d'améliorations sur la performance pour le Blur et Speech Recognition. Ces améliorations sont moindres que dans les scénarios avec moins de

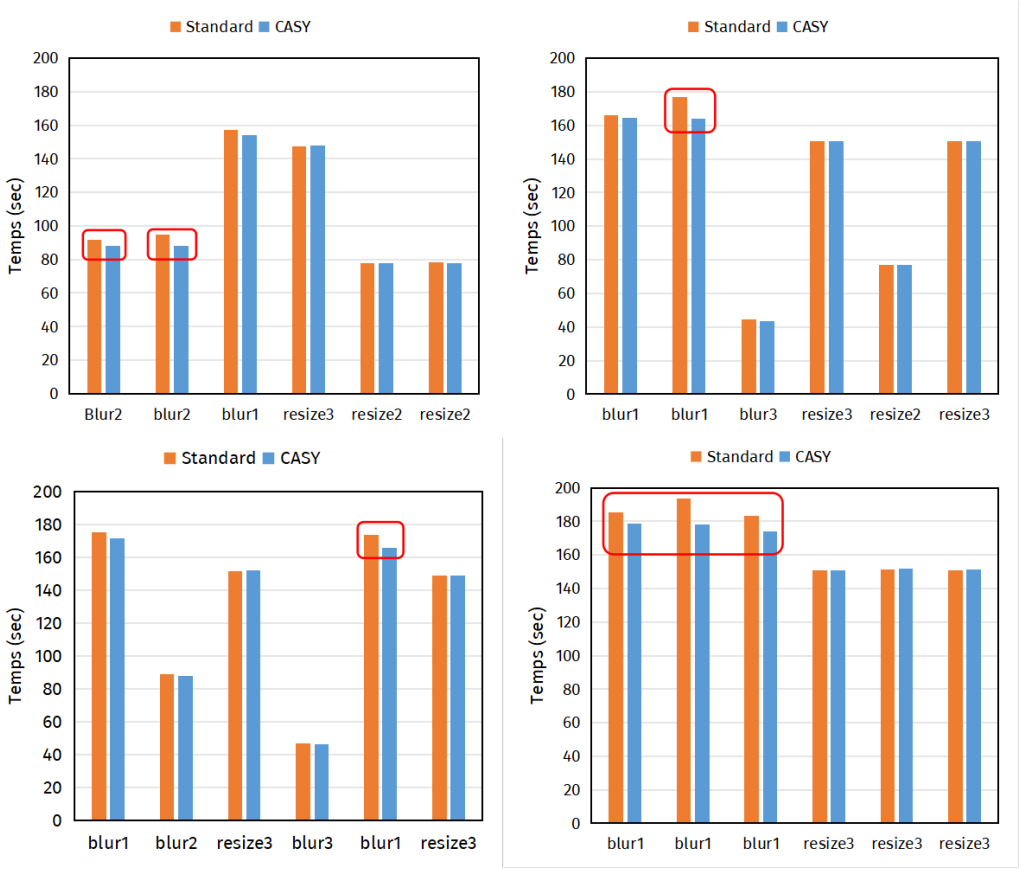


FIGURE 2.7 – Évaluation de l'allocation pour une exécution simultanée de 6 fonctions

fonctions (6 et 10). Cela s'explique par le fait que le cache est une ressource limitée et que plus il y a de fonctions qui utilisent le cache, moins l'impact sur l'allocation du cache est perceptible. Néanmoins, nous remarquons que *CASY* n'entraîne aucune dégradation des performances pour les autres fonctions.

2.4.4 Surcoût de *CASY*

Le surcoût induit par *CASY* provient principalement du profilage, car la prédiction et la configuration de CLOS ne consomment pratiquement pas de ressources (CPU et mémoire). Le processus de profilage de la fonction est le coût indirect le plus important de *CASY*, car il nécessite l'exécution préalable de la fonction sur un serveur dédié (ou sur un nœud NUMA dédié sur un serveur) avec différentes tailles de fichiers. Comme

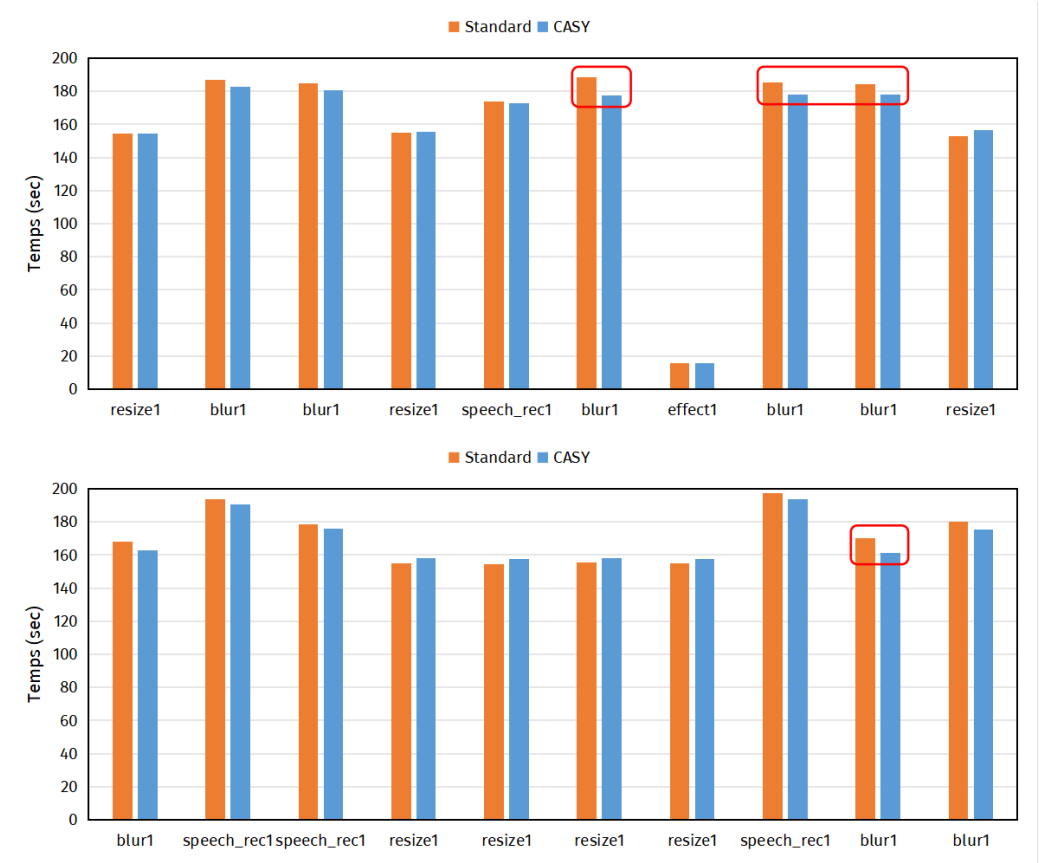


FIGURE 2.8 – Évaluation de l'allocation pour une exécution simultanée de 10 fonctions

indiqué précédemment, nous considérons que la durée de vie d'une fonction est assez longue et qu'une fonction est très probablement invoquée un grand nombre de fois. Nous considérons donc que le temps nécessaire au profilage est compensé par les avantages découlant de l'allocation de la mémoire cache du processeur.

Conclusion

Dans ce travail, nous avons présenté *CASY*, une solution pour améliorer l'allocation du cache CPU aux fonctions dans une plateforme FaaS. *CASY* s'appuie sur l'apprentissage automatique pour construire des profils d'utilisation du cache CPU pour les fonctions sur la base des données d'entrée, et utilise CAT pour partitionner le cache. Le profilage

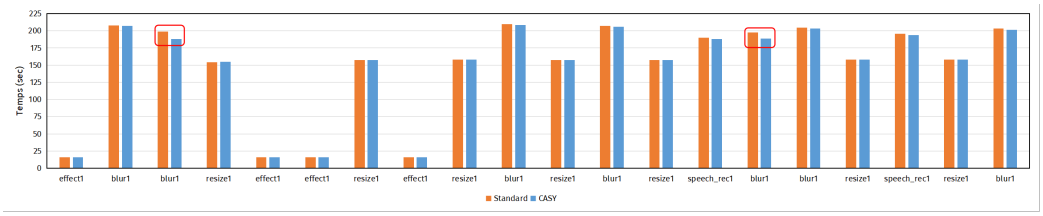


FIGURE 2.9 – Évaluation de l'allocation pour une exécution simultanée de 20 fonctions

des fonctions est effectué en exécutant les fonctions avec différentes données et sur un serveur dédié. Une fois le profilage effectué, *CASY* utilise un classificateur pour construire un modèle capable de prédire les besoins en cache d'une fonction dépendamment de ses données d'entrée. *CASY* propose également un algorithme d'allocation de cache qui permet une répartition équilibrée des charges sur toutes les voies de cache. *CASY* est construit comme un système qui peut facilement être intégré aux plateformes FaaS. Nous avons évalué l'amélioration des performances induite par *CASY* dans plusieurs scénarios différents et avons montré que *CASY* peut réduire le temps d'exécution de certaines fonctions tout en maintenant le même temps d'exécution pour d'autres fonctions.

L'innovation de *CASY* par rapport à l'existant est que nous combinons la gestion du LLC avec l'apprentissage automatique dans un environnement FaaS. Nous profitons du fait que les fonctions FaaS sont des conteneurs avec une utilisation cohérente du cache CPU, pour les classer en fonction de leur profil d'utilisation de ce cache, et ainsi effectuer une allocation intelligente de ce dernier.

Chapitre 3

CADiA, un système d'allocation de cache CPU intelligent et dynamique pour les applications distribuées

Introduction

Aujourd'hui, nous assistons à une variété croissante d'applications dans le domaine des technologies de l'information. Cette variété provient principalement de la généralisation du *Cloud Computing* [26], qui s'appuie sur les centres de données et permet ainsi aux entreprises et aux particuliers d'accéder à des infrastructures informatiques performantes. On assiste à une colocalisation importante de différentes charges de travail dans les centres de données. Parmi ces charges de travail, on trouve les applications parallèles distribuées qui représentent une part importante [27], plus particulièrement les applications SPMD (*Single Program Multiple Data*). Parmi les exemples de charges de travail SPMD populaires figurent les applications big data telles que Hadoop et Spark, ainsi que les charges de travail HPC qui utilisent des bibliothèques telles que MPI. Ces applications présentent généralement un flux d'exécution comprenant une phase de calcul intensif sur les nœuds, suivie d'une phase de communication pour transférer les résultats. Notre contribution porte sur l'allocation de cache CPU pour ces applications. Dans le reste de ce chapitre, nous considérons que les applications SPMD et les applications parallèles distribuées sont les mêmes.

Dans les applications SPMD, le temps d'exécution global est

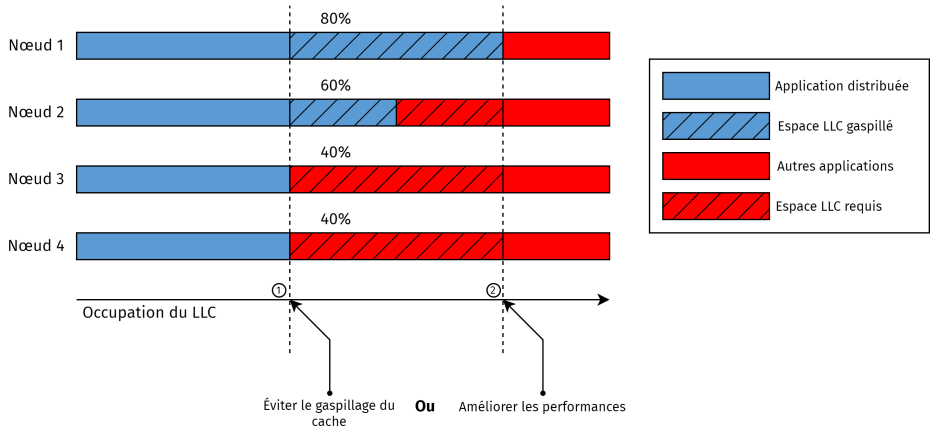


FIGURE 3.1 – Exemple d'occupation du LLC par une application parallèle distribuée s'exécutant sur 4 nœuds.

déterminé par les performances de chaque nœud (la phase de calcul intensif dans le flux d'exécution). Par conséquent, la vitesse de calcul de chaque nœud joue un rôle crucial dans le temps d'exécution global de l'application. Si un nœud est lent en raison d'une mauvaise allocation du LLC, cela affectera les performances globales de l'application. De ce fait, une stratégie d'allocation du cache CPU qui se concentre sur les nœuds individuellement sans avoir une vision globale de l'application distribuée est inadéquate. Par exemple, considérons une application parallèle distribuée s'exécutant sur quatre nœuds homogènes, comme illustré dans la figure 3.1, avec une forte contention de cache sur deux des nœuds (les nœuds 3 et 4 ont la majeure partie du LLC occupée par d'autres applications) et une contention de cache plus faible sur les deux autres nœuds (le LLC du nœud 1 et du nœud 2 est principalement utilisé par l'application distribuée). Premièrement, pour améliorer les performances de l'application, son occupation du LLC sur les nœuds 3 et 4 doit être augmentée jusqu'à atteindre celui sur les nœuds 1 et 2. Deuxièmement, l'utilisation plus importante du LLC sur les nœuds 1 et 2 (comme c'est le cas dans la figure 3.1) alors que les autres nœuds connaissent toujours une forte contention n'améliorera pas les performances de l'application, mais entraînera plutôt un gaspillage de LLC (cases vertes hachurées sur la figure 3.1). Ainsi, pour améliorer les performances ou éviter le gaspillage du LLC, l'allocation doit être coordonnée entre tous les nœuds. Autrement dit, elle doit prendre en compte la nature distribuée de ces applications.

Ce travail aborde ce problème en apportant une double contribution.

Premièrement, nous avons effectué une analyse approfondie de l'impact de la contention du LLC sur les performances d'une application distribuée. Deuxièmement, sur la base de nos conclusions, nous avons introduit CADiA (***C**ache **A**llocation system for **Di**stributed **A**pplications)*), un système qui propose deux stratégies d'allocation du cache du processeur spécialement conçues pour les applications parallèles distribuées.

Dans la première partie de notre contribution, qui porte sur la **contention du LLC sur les applications parallèles distribuées**, nous présentons une analyse approfondie de l'impact de la contention du LLC sur les performances globales des applications distribuées. À cette fin, nous avons mené des expériences en utilisant des benchmarks basés sur MPI avec différents niveaux de contention sur les nœuds. Nos résultats révèlent que :

(a) l'activité du cache du CPU pour l'application est comparable sur tous les nœuds ;

(b) la contention sur un nœud ou un sous-ensemble de nœuds a un impact direct sur le temps d'exécution global de l'application ;

(c) le temps d'exécution est fortement influencé par le niveau de contention le plus élevé sur les nœuds.

Sur la base de ces observations, nous proposons deux nouvelles stratégies d'allocation de cache pour les applications distribuées.

Dans la deuxième partie de notre contribution, qui concerne les **stratégies d'allocation du LLC**, nous proposons, sur la base des observations précédentes, *CADiA*, un système ciblant l'allocation du LLC pour les applications distribuées. Le but de CADiA est d'harmoniser l'allocation du LLC sur tous les nœuds pour les applications distribuées. *CADiA* est conçu en deux phases : le profilage et l'allocation. Dans la phase de profilage, *CADiA* crée un profil d'utilisation du LLC sur tous les nœuds et construit la courbe du taux d'échec [1] (MRC) pour tous les processus s'exécutant sur ces nœuds (l'application distribuée et les autres). Pour ce faire, des mesures de bas niveau sont collectées pendant l'exécution de l'application. Une fois les courbes MRC construites, elles sont utilisées pour déterminer la quantité de LLC à allouer sur tous les nœuds pour l'application distribuée. Cette taille est fixée en fonction de la stratégie sélectionnée. *CADiA* propose deux stratégies nommées : *Top-Down* et *Bottom-Up*.

1. Courbe représentant le taux d'échec (*miss rate*) en fonction de l'occupation de la mémoire cache

Top-Down. Cette première stratégie donne la priorité à l'application distribuée par rapport aux autres applications s'exécutant sur les nœuds et vise à allouer la quantité optimale de cache pour l'application distribuée. Comme son nom l'indique pour cette stratégie, le choix du LLC à allouer provient du maître (en haut) et est défini sur les nœuds (en bas). En utilisant le MRC de l'application distribuée, *CADiA* détermine la quantité minimale de LLC qui doit être allouée pour que l'application fonctionne de manière optimale et alloue cette taille sur tous les nœuds de calcul.

Bottom-Up. La deuxième stratégie accorde la même priorité à l'application distribuée et aux applications colocalisées sur les nœuds. Dans cette stratégie, les contraintes de contention sur les nœuds sont remontées au nœud maître et déterminent la quantité de LLC à allouer à l'application distribuée, d'où le nom "Bottom-Up". L'objectif est d'empêcher l'allocation ou l'utilisation inutile du cache par l'application distribuée. Pour ce faire, *CADiA* utilise le MRC de l'application distribuée et des autres applications sur chaque nœud pour calculer la taille minimale de LLC nécessaire aux deux. Ensuite, pour chaque nœud, il calcule la quantité de LLC à allouer à l'application distribuée pour une allocation **équitable** avec les autres applications. La quantité de LLC qui sera finalement allouée à l'application distribuée sur tous les nœuds sera le minimum des valeurs calculées sur tous les nœuds.

Nous avons mis en œuvre ces deux stratégies d'allocation dans la bibliothèque Open MPI [28] et évalué leurs performances avec des applications de calcul à haute performance (HPC). Nos résultats démontrent que, en utilisant stratégie *Top-Down*, *CADiA* améliore les performances des applications parallèles distribuées jusqu'à 13% dans certains scénarios. Ensuite, lorsque la stratégie *Bottom-Up* est utilisée, *CADiA* diminue le taux d'échec (*miss rate*) sur le LLC pour les applications colocalisées jusqu'à 40%.

Le reste de ce chapitre est organisé comme suit. La section 3.1 motive la nécessité d'un système tel que *CADiA*. La section 3.2 présente la vue d'ensemble de *CADiA* ainsi que sa conception. La section 3.4 présente les résultats de l'évaluation. Et enfin la dernière section sera la conclusion de ce chapitre.

3.1 Les enjeux de la contention de cache pour les applications distribuées

Comme indiqué dans l'introduction de ce chapitre, la plupart des études sur l'allocation du LLC se sont principalement concentrées sur l'analyse d'un seul nœud pour l'allocation, ce qui n'est pas adapté aux applications parallèles distribuées. Dans cette section, nous menons une étude sur l'impact de la contention du LLC sur diverses applications parallèles distribuées. L'objectif est de démontrer la nécessité d'une stratégie globale d'allocation du cache CPU. Le but de cette section est tout d'abord d'étudier l'impact de la contention du cache sur les performances des applications parallèles distribuées et ensuite d'analyser le comportement de l'application envers le LLC sur chaque nœud.

3.1.1 Configuration expérimentale

Dans notre terminologie et pour le reste de ce chapitre, nous définissons l'application *bubble* (une application à plusieurs threads) comme l'application qui contrôle l'utilisation de son LLC. L'application *bubble* nous permet de générer un niveau de contention souhaité sur les nœuds de calcul sur les deux sockets *NUMA*. L'acronyme $bubble_x$ représente une exécution de l'application *bubble*, avec un nombre x de voies utilisées. Ainsi, la notation $bubble_1$ signifie que la bulle a été configurée pour utiliser une voie de cache. Les configurations matérielles utilisées sont spécifiées dans le tableau 3.1 et les benchmarks utilisés sont décrits dans la section 3.4. En ce qui concerne la configuration matérielle, étant donné que notre système possède deux sockets *NUMA*, nous avons donc deux processeurs physiques, chacun ayant 11 voies, ce qui conduit à 22 voies pour la taille totale du LLC. Lors de nos évaluations, les applications sont exécutées sur les deux sockets de tous les nœuds.

Notre procédure expérimentale consiste à faire varier le nombre de nœuds de calcul, le niveau de contention (colocation avec $bubble_2$ à $bubble_{14}$), et le nombre de nœuds soumis à contention pour chaque application distribuée. Nous comparons les résultats obtenus avec les performances de l'application fonctionnant avec $bubble_0$ afin d'intégrer la surcharge du système due à l'exécution de processus multiples (l'application distribuée et l'application *bubble*). Pour éviter toute interférence de l'ordonnancement, nous avons assigné tous les processus des applications distribuées et de *bubble* à des cœurs spécifiques, évitant ainsi tout impact de l'ordonnancement sur les performances. Il est

Composant	Caractéristiques
CPU	2×Intel Xeon Gold 6130
	2 sockets NUMA, 16 cœurs par socket
	8 CLOS supportés
Cache niveau 3 (LLC)	2×22MB, 11-voies
Mémoire RAM	192 GB (96 GB par socket)
Ethernet	Intel Ethernet Controller X710 10GbE
Stockage	240 GB SSD
Système d'exploitation	Ubuntu 20.04.5, Linux kernel 5.4.0

TABLE 3.1 – Configuration matérielle.

important de noter que dans ces évaluations, nous nous sommes assurés de l'homogénéité à la fois du matériel et du niveau de contention entre les nœuds. Les résultats de nos expériences sont présentés dans les cartes thermiques de la figure 3.2, où les nombres représentent la dégradation calculée comme suit :

$$\text{Deg} = \frac{(\text{Perf}' - \text{Perf})}{\text{Perf}} \times 100 \tag{3.1}$$

avec Perf′ et Perf représentant respectivement la performance de l'application distribuée avec et sans contention (exécution avec $bubble_0$) sur les nœuds.

3.1.2 Résultats et analyse

Sur la figure 3.2, l'axe des y représente le nombre de nœuds subissant une contention, tandis que l'axe des x représente le nombre de voies de cache utilisées par l'application *bubble*. Les résultats affichés concernent une exécution avec 8 nœuds. Nous n'avons pas montré les résultats pour l'exécution avec 4 nœuds, car les observations que nous avons faites sont les mêmes qu'avec 8 nœuds. Nous pouvons observer que la contention sur un seul nœud entraîne une dégradation des performances pour toutes les applications (premières lignes inférieures de toutes les cartes thermiques). En outre, la dégradation augmente avec le nombre de voies utilisées par l'application *bubble*, ainsi qu'avec le nombre de nœuds saturés. Cela signifie que la dégradation augmente avec le nombre de nœuds saturés. Ceci est illustré par le fait que les colonnes les plus sombres sont situées en haut à droite des cartes thermiques. En outre, la dégradation n'est pas la même pour tous les benchmarks. Pour

Miniweather et SPH-EXA, la dégradation peut atteindre respectivement jusqu'à 20% et 8%, alors que pour les autres benchmarks, elle est d'environ 5% au pire. Cela montre que la sensibilité au LLC n'est pas la même pour toutes les applications. Nous confirmerons cette hypothèse lors de notre évaluation (Section 3.4).

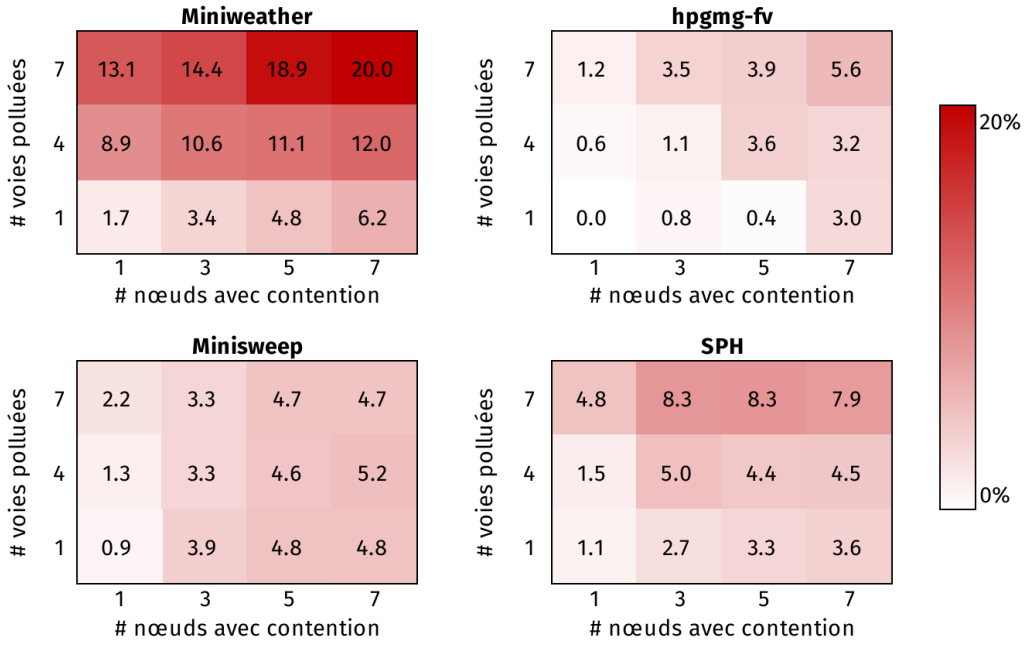


FIGURE 3.2 – Cartes thermiques présentant l'impact sur les performances de la contention du cache du processeur sur des applications distribuées fonctionnant sur 8 nœuds.

Pour comprendre le comportement cache des applications SPMD sur chaque nœud et proposer un système d'allocation, nous avons exécuté tous les benchmarks sans contention sur les nœuds et surveillé l'utilisation du cache sur chaque nœud. Sur les figures 3.3 et 3.4, nous présentons l'occupation LLC et les défauts de cache (*cache misses*) sur 4 nœuds uniquement pour le benchmark Miniweather (les observations sont les mêmes pour les autres benchmarks). Nous pouvons constater que l'occupation du LLC et le nombre de défauts de cache sont identiques sur tous les nœuds. Ce résultat n'est pas surprenant, car ces applications effectuent souvent des traitements identiques sur tous les nœuds. Cette observation est fondamentale pour le système que nous proposons, car elle nous permet d'affirmer que pour les applications parallèles distribuées, le profil d'utilisation du LLC est le même sur tous

les nœuds. Par conséquent, il est possible d'avoir une politique d'allocation de cache centralisée qui évite la **sur-allocation** ou la **sous-allocation** sur les nœuds en allouant la même taille de cache sur tous les nœuds de calcul. *CADiA* permet de définir la quantité de LLC à allouer et alloue cette quantité sur tous les nœuds.

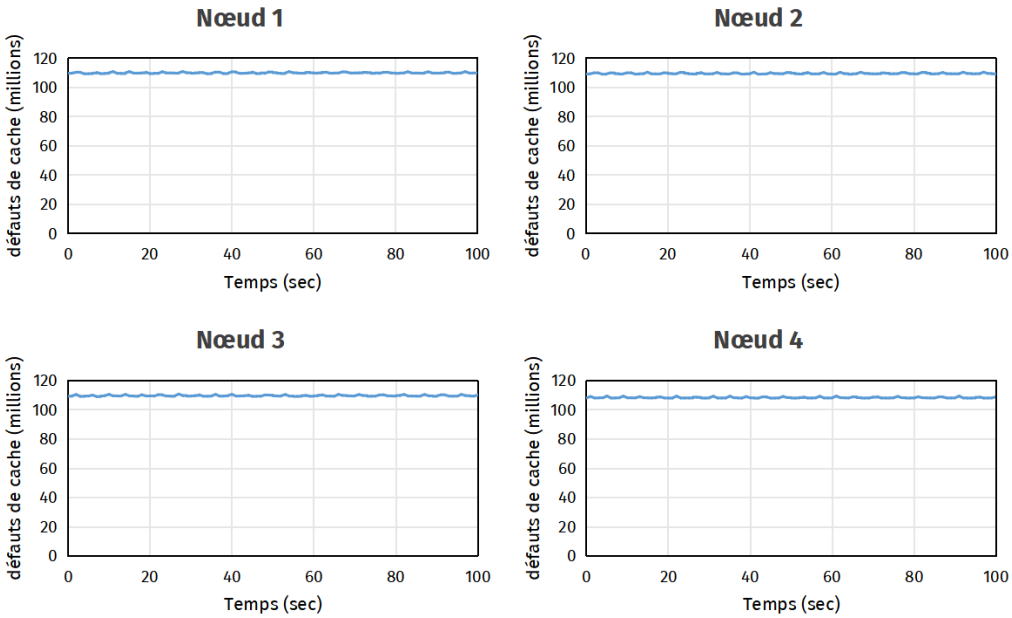


FIGURE 3.3 – Défauts de cache sur 4 nœuds de calcul par Miniweather au cours de son exécution.

Des résultats précédents, nous avons pu tirer trois leçons qui nous serviront à la mise en œuvre de *CADiA* :

(1) la contention sur un ou plusieurs nœuds a un impact sur le temps d'exécution global de toutes les applications parallèles distribuées ;

(2) l'impact sur les applications varie en fonction du profil d'utilisation du LLC de l'application ;

(3) le comportement vis-à-vis du cache CPU des applications parallèles distribuées est identique sur tous les nœuds de calcul.

3.2 Contribution

Dans cette section, nous fournissons une description détaillée de l'architecture de notre système. Nous commençons par une vue

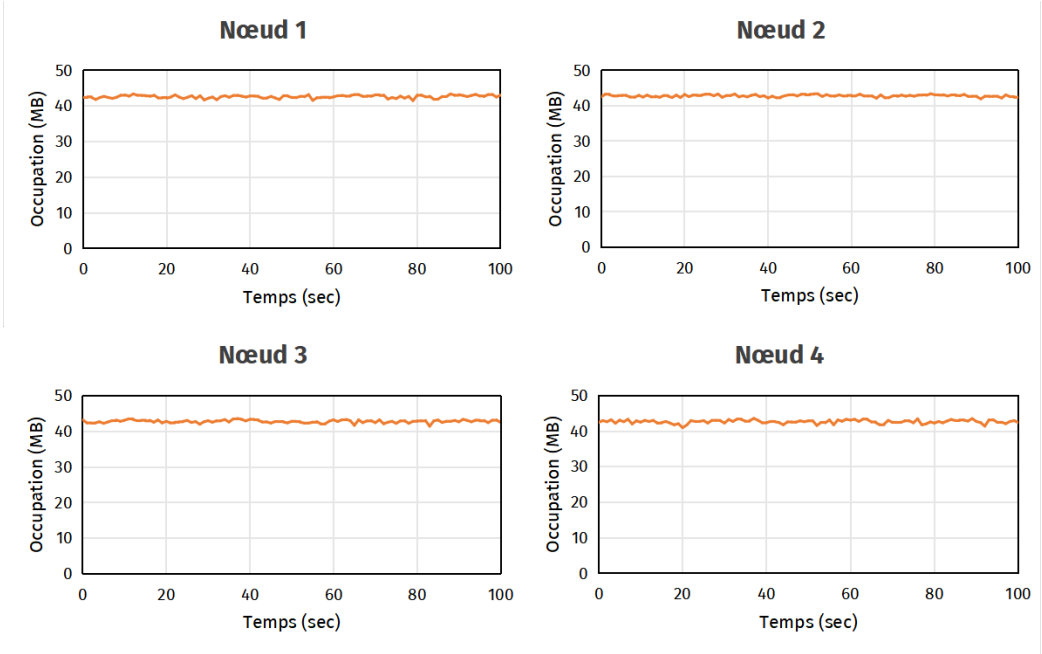


FIGURE 3.4 – Occupation du LLC sur 4 nœuds de calcul par Miniweather au cours de son exécution.

d'ensemble, puis nous détaillons chacun des composants architecturaux et leurs objectifs respectifs.

3.2.1 Vue d'ensemble

Comme indiqué précédemment, *CADiA* est un système d'allocation de la mémoire cache CPU conçu pour les applications parallèles distribuées. Il permet d'harmoniser la stratégie d'allocation du cache CPU sur tous les nœuds de calcul exécutant l'application. La figure 3.5 illustre l'architecture générale de *CADiA* et ses composants : *CADiA Agent*, *CADiA Profiler et CADiA Allocator*.

CADiA fonctionne en deux phases : ❶ le profilage et ❷ l'allocation. Dans la phase de profilage (❶), le profileur recueille des métriques de bas niveau à l'aide de l'agent pour établir un profil de l'utilisation du LLC par l'application distribuée et des autres applications fonctionnant sur les nœuds. L'objectif de cette phase est de générer les courbes de taux d'échecs (MRC). Ces courbes sont utilisées pour déterminer la taille minimale de LLC requise ne causant pas une dégradation des performances pour toutes les applications sur les nœuds. L'étape de

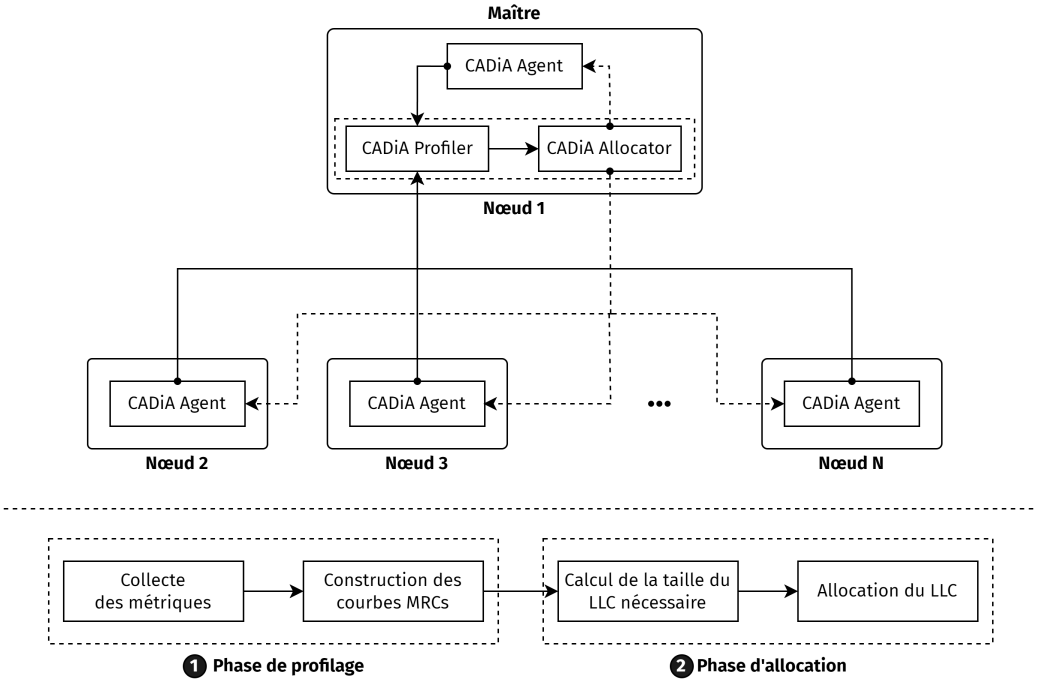


FIGURE 3.5 – Architecture de *CADiA*

profilage est déclenchée périodiquement suivant une période définie par l'utilisateur. L'agent est chargé de collecter des métriques de bas niveau sur chaque nœud et de les envoyer au profileur pour construire les MRCs. L'Agent alloue également la quantité de LLC définie par l'Allocateur par la suite. Dans la phase d'allocation ❷, l'Allocateur utilise les informations des MRCs pour allouer la taille de cache requise à l'application distribuée sur la base de la stratégie d'allocation choisie. Nous introduisons deux stratégies dans *CADiA* : les stratégies *Top-Down* et *Bottom-Up*, qui sont sélectionnées en fonction du niveau de priorité attribué à l'application distribuée. En harmonisant la stratégie d'allocation du cache CPU sur tous les nœuds, *CADiA* améliore les performances des applications distribuées, en particulier en présence de contention sur les nœuds.

3.2.2 Architecture détaillée de *CADiA*

Dans notre approche, nous classons les processus sur un nœud en deux groupes : ceux qui appartiennent à l'application distribuée et ceux qui n'y appartiennent pas. Les sections suivantes décrivent la façon dont nous construisons les MRCs et allouons le LLC pour ces deux groupes.

CADiA Profiler

Ce composant est responsable de la construction des MRCs de l'application distribuée et des autres applications fonctionnant sur les nœuds. Pour construire un profil, le Profileur a besoin de métriques de bas niveau de l'application avec différentes tailles de cache allouées. Ces mesures sont les suivantes : **Le nombre de défauts de cache**, **le nombre de références de cache** et **l'occupation du cache**.

Soit Nb_{noeuds} le nombre total de nœuds de calcul utilisés par une application distribuée, et N_i le nom de chaque nœud avec $i \in \{1, ...Nb_{voies}\}$. En outre, notons Nb_{voies} comme le nombre total de voies de cache qui peuvent être allouées. Au cours de la phase de profilage, le Profileur doit collecter des métriques de bas niveau avec différentes tailles de cache allouées et construire $2 \times Nb_{noeuds}$ MRCs. Pour ce faire, le Profileur demande aux agents de configurer différentes tailles de cache pour l'application distribuée et les autres applications. Cette opération est réalisée selon l'algorithme 3. L'agent commence par construire les deux groupes de processus, ceux de l'application distribuée et les autres (lignes 1-2). Ensuite, les structures de données qui contiendront les valeurs collectées pour chaque groupe (lignes 3-4) sont initialisées. L'agent alloue une partie spécifique du LLC (i voies) aux processus de l'application distribuée, collecte les métriques correspondantes, et assigne le cache restant aux autres applications, en collectant également leurs métriques (lignes 6-16). Cette opération est répétée pour i allant de 1 à Nb_{voies} (ligne 5). Une fois cette opération terminée, les agents de tous les nœuds ont collecté les métriques de l'application distribuée et des autres applications sur tous les nœuds. Les données collectées sont alors utilisées par le Profileur pour construire les MRCs. Le MRC final pour l'application distribuée est construit en combinant les données de tous les nœuds.

CADiA Agent

CADiA Agent joue un rôle crucial dans la collecte de données en rassemblant des métriques sur l'utilisation du LLC sur les nœuds de calcul. Ces mesures sont ensuite transmises au Profileur. L'agent collecte les données sur une période déterminée. Pour construire le MRC d'une application, le Profileur a besoin du nombre de défauts de cache pour différentes tailles de cache allouées. L'agent est chargé d'allouer une partie du cache à l'application sur chaque nœud afin d'obtenir ces mesures. Enfin, une fois que l'allocateur a déterminé la taille de cache

Algorithm 3 Allocation du LLC pour le profilage sur un nœud

```
app1 ← RECUPERER_LISTE_PID("DistApp")
app2 ← RECUPERER_LISTE_PID("OtherApp")
metriques_appli_dist ← HASHMAP( )
metriques_autres_appli ← HASHMAP( )
for i in 1 .. nb_voies − 1 do
    ALLOUER_VOIES(app1, i)
    ALLOUER_VOIES(app2, nb_voies - i)
    evenements ← ["Misses", "References", "Occupancy"]
    mesures ← COLLECTER(app1, evenements)
    occupation ← mesures["Occupancy"]
    taux_d_echec ← mesures["Misses"] / mesures["References"]
    metriques_appli_dist[occupation] ← taux_d_echec
    mesures ← COLLECTER(app2, evenements)
    occupation ← mesures["Occupancy"]
    taux_d_echec ← mesures["Misses"] / mesures["References"]
    metriques_autres_appli[occupation] ← taux_d_echec
end for
```

appropriée, il utilise l'agent pour définir l'allocation en conséquence.

CADiA Allocator

CADiA allocator joue un rôle central en déterminant la taille du cache à allouer sur tous les nœuds, et en définissant la politique d'allocation pour l'application distribuée. Dans notre travail, comme presenté sur la figure 3.6, nous proposons deux stratégies d'allocation, à savoir *Top-Down* et *Bottom-Up*, qui sont sélectionnées en fonction du niveau de priorité attribué à l'application distribuée. La stratégie *Top-Down* est employée lorsque l'objectif est de maximiser les performances de l'application distribuée, ce qui signifie que la priorité absolue est accordée à cette application. En revanche, la stratégie *Bottom-Up* est utilisée lorsque l'objectif est d'effectuer une allocation équitable, accordant ainsi à l'application distribuée la même priorité que les autres applications.

Définissons P_i comme la performance des processus de l'application distribuée sur le nœud N_i. P_i dépend du cache alloué sur le nœud. Plus précisément, P_i est défini comme une fonction de C_i :

$$P_i = f(C_i) \tag{3.2}$$

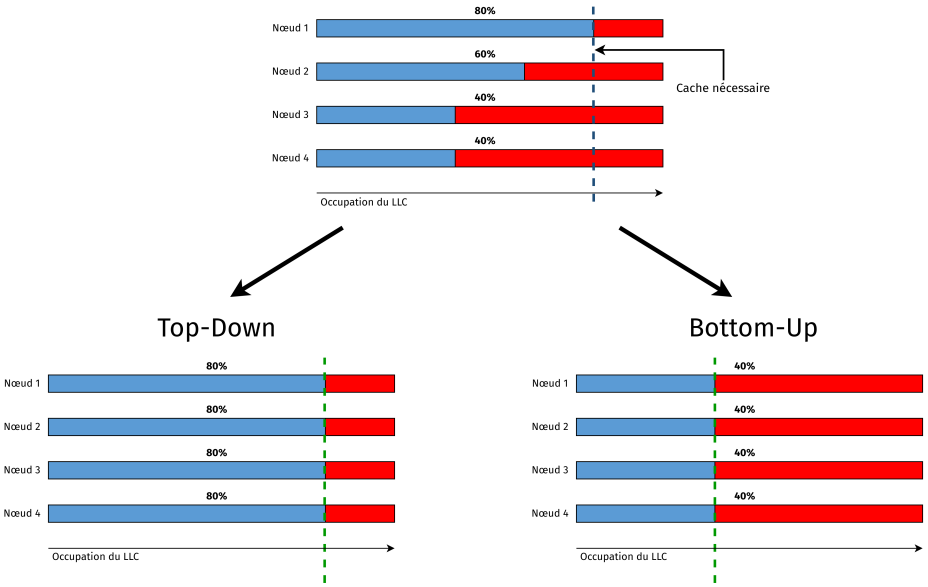


FIGURE 3.6 – Stratégies d'allocation du LLC

Avec C_i représentant la quantité de LLC allouée à l'application distribuée sur le nœud N_i. Les performances globales de l'application distribuée dépendent de différents $P_i \; \forall i \in \{1, .., Nb_{noeuds}\}$. En fonction de la stratégie choisie, une valeur de C_i sera fixée par l'allocateur sur tous les nœuds. Avant de présenter les deux stratégies, il est important de discuter de l'utilisation de MRC pour déterminer la taille minimale de cache requise par une application pour fonctionner sans impact sur le LLC.

Une fois que le Profileur a construit les MRC, il les envoie à l'Allocateur, qui les exploite en fonction de la stratégie à utiliser. La recherche de la taille de cache minimale à partir du MRC s'effectue selon l'algorithme 4. Le MRC et la taille du LLC sont les données d'entrée de l'algorithme (lignes 1-2). Tout d'abord, les tableaux X et Y contenant respectivement les occupations du LLC et les taux d'échec, sont initialisés (lignes 3-7). Ensuite, la méthode des moindres carrés est utilisée pour déterminer la ligne de tendance (droite de régression) pour X et Y (ligne 10), et en déduire la pente (ligne 11). Si cette pente multipliée par la taille du LLC est inférieure à un seuil (ligne 12), cela signifie que la ligne de tendance est assez proche de l'horizontale, le premier point du tableau X est alors retourné (ligne 13). Sinon, les premiers éléments des tableaux X et Y sont retirés (lignes 15 et 16) et l'opération est répétée tant que la taille de X (ou de Y) est supérieure à 1 (ligne 9). Si la taille de X est égale à 1 à la fin de l'opération, cela signifie que l'application a besoin de la totalité du

LLC pour fonctionner de manière optimale. Cet algorithme permet de trouver la quantité minimale de cache nécessaire à une application à partir de son MRC, et est utilisé par les deux stratégies.

Algorithm 4 Calcul de la quantité requise de LLC

```
1: MRC ← RECUPERER_MRC( )
2: taille_llc ← RECUPERER_TAILLE_LLC( )
3: X ← TABLEAU( )
4: Y ← TABLEAU( )                          ▷ X et Y gérés en FIFO.
5: for (occupation, taux_d_echec) in MRC do
6:     X ← ENFILER(X, occupation)
7:     Y ← ENFILER(Y, taux_d_echec)
8: end for
9: while TAILLE(X) > 1 do
10:    droite ← MOINDRES_CARRES(X, Y)
11:    pente ← PENTE(droite)
12:    if pente × taille_llc < Seuil then
13:        return X[0]
14:    end if
15:    DEFILER(X)
16:    DEFILER(Y)
17: end while
18: return X[0]
```

Stratégie *Top-Down* Dans cette stratégie, nous donnons la priorité aux besoins en cache de l'application distribuée par rapport aux autres applications fonctionnant sur les nœuds. Par conséquent, notre objectif est d'allouer suffisamment de cache pour permettre l'exécution correcte de l'application distribuée. Cet objectif peut être formalisé comme suit :

$$P_i = f(C_{meilleur}), \forall\, i \in \{1, .., Nb_{noeuds}\} \tag{3.3}$$

avec $C_{meilleur}$ étant la taille optimale du cache nécessaire sur chaque nœud pour que l'application distribuée ne soit pas affectée par la contention du cache. L'objectif de cette stratégie est de déterminer $C_{meilleur}$ et de l'allouer sur tous les nœuds. Pour ce faire, l'allocateur utilise uniquement le MRC de l'application distribuée et détermine la taille optimale du LLC à l'aide de l'algorithme 4. Une fois que $C_{meilleur}$ est déterminé, l'allocateur demande aux agents de définir cette quantité de LLC sur tous les nœuds pour l'application distribuée et $Nb_{voies} - C_{meilleur}$ voies de LLC est allouée aux autres applications.

Stratégie *Bottom-Up* Ici, l'application distribuée n'est pas prioritaire par rapport aux autres applications en cours d'exécution sur les nœuds. L'objectif de cette stratégie est de mettre en œuvre une allocation équitable du LLC pour tous les processus. Les deux MRC sont utilisés, l'application distribuée et les autres processus. L'algorithme utilisé est celui présenté dans l'Algorithme 5. La première étape consiste à calculer la taille optimale de LLC pour l'application distribuée et les autres applications sur chaque nœud (lignes 3-4). Ensuite, la quantité de LLC à allouer à l'application distribuée est calculée proportionnellement aux besoins des deux groupes d'applications (lignes 6). Enfin, la quantité de LLC allouée à l'application distribuée sur tous les nœuds sera le minimum des valeurs obtenues localement sur chaque nœud (lignes 8-9), et le cache restant sera alloué aux autres processus (ligne 10). Cette stratégie permet la mise en œuvre d'une allocation locale équitable sur les nœuds tout en évitant le gaspillage de LLC par l'application distribuée.

Algorithm 5 Allocation équitable du LLC

1: $C \leftarrow$ TABLEAU( )
2: **for** i **in** 1 .. Nb_{nodes} **do**
3: $C_1 \leftarrow$ UTILISER_MRC("DistApp")
4: $C_2 \leftarrow$ UTILISER_MRC("OtherApp")
5: $C \leftarrow$ AJOUTER($nb_{\text{voies}} \times \frac{C_1}{C_1+C_2}$)
6: **end for**
7: $C_{\text{allouer}} \leftarrow$ MIN(C)
8: ALLOUER_VOIES_SUR_LES_NOEUDS(C_{allouer},"DistApp")
9: ALLOUER_VOIES_SUR_LES_NOEUDS($nb_{\text{voies}} - C_{\text{allouer}}$, "OtherApp")

Notre solution montre de bons résultats dans des environnements hétérogènes, où l'hétérogénéité se manifeste à deux niveaux.

— **Fréquence du processeur** : Les variations de fréquence du processeur n'ont pas ou peu d'impact sur le MRC d'une application. En effet, le comportement du cache, en termes de taux de misses, est plus dépendant des accès mémoire que du rythme d'exécution des instructions.

— **Taille de la mémoire cache** : La taille totale de la mémoire cache affecte uniquement les domaines de définition de chaque courbe MRC. Cela n'est pas problématique, car en centralisant les données sur le nœud principal, nous obtenons un MRC final qui est l'union de tous les MRCs des différents nœuds.

En résumé, le taux d'échec dépend principalement de la taille de la mémoire cache occupée, indépendamment des variations de fréquence des processeurs et de taille des mémoires cache.

3.3 Mise en œuvre de *CADiA*

Nous avons implémenté CADiA comme une option supplémentaire dans la bibliothèque Open MPI, version 3.3.2. Nous avons modifié la bibliothèque pour inclure une option qui permet de spécifier si la gestion de LLC est souhaitée ou non pour notre exécution et également quelle stratégie utiliser. Le code source de CADiA est composé de 1 461 lignes de code en C++. Nous présenterons l'implémentation de chaque composant, et nous avons également prévu de publier le code source du projet une fois l'article publié.

CADiA Agent L'agent est lancé sous la forme d'un démon qui s'exécute sur tous les nœuds. La collecte des métriques de bas niveau nécessaires est effectuée à l'aide de la bibliothèque PQoS [29] fournie par Intel. Cette bibliothèque permet d'obtenir des statistiques sur l'utilisation du cache des processus. Elle est également utilisée pour configurer les CLOS, c'est-à-dire pour leur assigner des voies de cache, et pour lier des processus à un CLOS.

CADiA Profiler Il s'exécute sur le nœud principal en tant que démon et communique avec l'agent en utilisant le protocole TCP pour le transfert des métriques. Il est également chargé de déclencher le profilage des applications à leur lancement et de façon périodique.

CADiA Allocator L'allocateur est également un démon qui s'exécute sur le nœud principal. Il communique avec les agents à l'aide d'une connexion TCP pour indiquer la quantité de LLC à allouer aux processus sur les nœuds. L'élément intéressant de ce composant est la quantité de LLC allouée sur les nœuds. Étant donné que l'allocation avec CAT est effectuée avec la voie comme unité d'allocation, il est difficile d'allouer exactement la quantité de LLC obtenue après l'utilisation de MRC. Par conséquent, dans notre implémentation, nous allouons à chaque fois la quantité de LLC la plus proche qu'il est possible d'allouer avec les voies. Par exemple, si le MRC détermine que 3 MB sont nécessaires sur chaque

nœud pour l'application distribuée, puisque la taille de la voie est de 2 MB, l'allocateur alloue 2 voies qui font 4 MB sur chaque nœud.

3.4 Évaluation

Dans cette section, nous évaluons *CADiA*. Les principaux objectifs sont les suivants :

(a) évaluer la capacité de *CADiA* à construire des profils d'application, c'est-à-dire à construire leur MRC et à déterminer leurs besoins en cache ;

(b) étudier la capacité de *CADiA* à améliorer les performances des applications parallèles distribuées ;

(c) estimer le surcoût de *CADiA* ;

3.4.1 Environnement et procédure expérimentale

Nous avons réalisé l'évaluation sur des clusters homogènes de 4 et 8 machines sur la plateforme Grid5000. Grid5000 [30] est une plateforme des universités françaises qui donne accès à des serveurs pour des évaluations de recherche. Les caractéristiques des machines utilisées sont listées dans le tableau 3.1. Il est important de noter que notre cluster est homogène, c'est-à-dire que les serveurs sont identiques.

La procédure expérimentale a consisté à exécuter des benchmarks sur le cluster avec et sans l'activation de *CADiA*, avec différentes stratégies de gestion du LLC. Nous avons ensuite comparé les performances obtenues pour estimer les améliorations apportées par *CADiA*. Les benchmarks utilisés proviennent de la suite SPEChpc [31], une suite de benchmark HPC reconnue dans la communauté effectuant des calculs dans divers domaines scientifiques. Le tableau 3.2 présente les benchmarks et une brève description de chacun d'eux. La métrique utilisée pour l'évaluation des performances est le temps d'exécution, et pour chaque expérimentation, nous avons exécuté le test cinq fois. Les résultats présentés sont les valeurs moyennes de ces cinq exécutions. Nous avons également calculé les écarts types qui étaient très faibles, c'est pourquoi ils ne sont pas représentés. Nous avons fixé la valeur de la période de déclenchement du profilage de *CADiA* Profiler à 5 minutes.

Nom	Description
Miniweather	Calculs météorologiques
HPGMG-FV	Cosmologie, Astrophysique, Combustion
Minisweep	Ingénierie nucléaire - Transport des radiations
SPH-EXA	Astrophysique et cosmologie

TABLE 3.2 – Liste des benchmarks et leur description

3.4.2 Évaluation du profilage de *CADiA* et de l'utilisation du MRC

Dans cette première évaluation, nous nous sommes concentrés sur le profilage offert par *CADiA*. L'objectif est d'évaluer la capacité de *CADiA* à construire le MRC d'une application distribuée, ainsi que de déterminer le LLC minimal requis. Nous avons exécuté nos applications distribuées sur 8 nœuds de calcul, 4 nœuds étant soumis à la contention par l'application *bubble*, et nous avons laissé *CADiA* extraire le MRC et calculer le LLC requis. La figure 3.7 présente les MRCs obtenus.

On constate que les quatre applications n'ont pas la même sensibilité au cache, ce qui valide les observations de la section 3.1. Miniweather et SPH-EXA semblent plus sensibles au LLC (leurs performances varient avec la quantité de LLC allouée) que Minisweep et HPGMG-FV, qui ne voient pas leurs performances impactées par la quantité de LLC allouée. Les points rouges sur les courbes représentent les tailles minimales de cache obtenues par notre algorithme (Algorithme 4). Les tailles de LLC requises sont respectivement de 38 Mo pour Miniweather, 7 Mo pour HPGMG-FV, 8.3 Mo pour Minisweep, et 18.5 Mo pour SPH-EXA. Nous pouvons en observant ces graphes remarquer que ces points sont bel et bien ceux à partir desquels l'augmentation du cache n'entraine pas une baisse du taux d'échec, ce qui démontre l'efficacité du profilage de *CADiA*. Cela est également vérifié par l'expérimentation, car lorsqu'on exécute ces benchmarks en leur allouant ces tailles minimales de cache, on n'observe aucune dégradation (moins de 5%) de leur temps d'exécution.

3.4.3 Impact de *CADiA* sur les performances des applications

Dans cette section, nous allons présenter l'évaluation de la stratégie *Top-Down* de *CADiA* dans le cas d'une contention homogène et

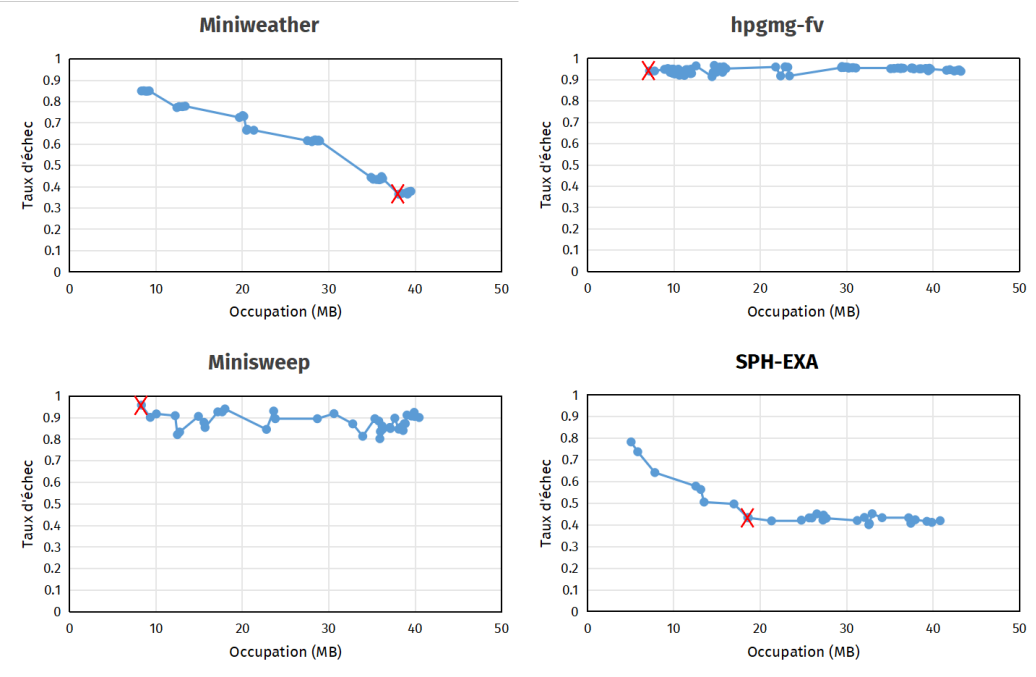


FIGURE 3.7 – Courbes du taux d'échec (MRC) des benchmarks. Construites à partir d'une exécution sur 8 nœuds. Les points rouges représentent les tailles minimales de LLC estimées par notre algorithme.

hétérogène, ainsi que la stratégie *Bottom-Up* dans le cas d'une contention homogène uniquement.

Top-Down : contention homogène

Dans cette évaluation, la contention sur les nœuds est homogène. Nous avons utilisé la stratégie *Top-Down* dans l'évaluation, ce qui signifie que l'application distribuée a une priorité plus élevée que les autres applications sur le nœud. Les figures 3.8 et 3.9 présentent les résultats obtenus pour une exécution sur 8 et 4 nœuds. Sur ces figures, l'axe des x représente le nombre de nœuds sur lesquels il y a de la contention de cache, et l'axe des y représente la performance normalisée par rapport à l'exécution de base. L'exécution de base correspond à une exécution sans contention et sans *CADiA*, tandis que l'exécution standard correspond à une exécution avec contention et sans *CADiA*.

La première observation concerne le gain de performance. Nous pouvons observer que l'amélioration des performances est d'environ 13% pour Miniweather et 4% pour SPH-EXA par rapport à une exécution

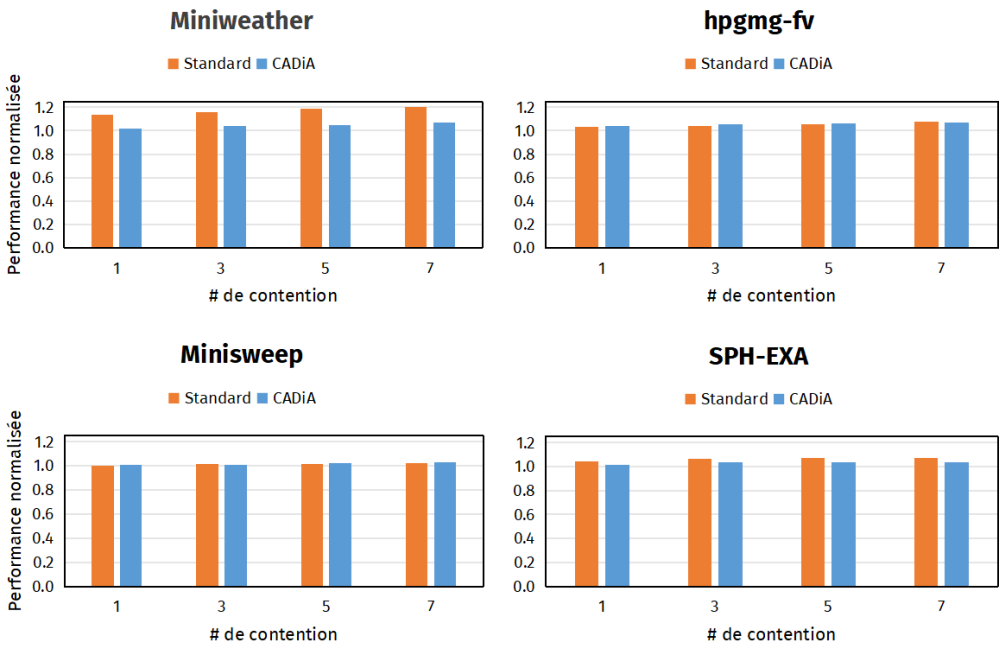


FIGURE 3.8 – Exécution sur 8 nœuds avec la stratégie *Top-Down*

standard. En revanche, nous n'observons aucune amélioration pour Minisweep et HPGMG-FV. Ceci est cohérent avec les observations faites sur la sensibilité LLC du benchmark. L'amélioration attendue dépend du nombre de nœuds utilisés par l'application; plus il y a de nœuds de calcul, plus l'amélioration de CADiA est importante. Pour les deux autres benchmarks pour lesquels nous n'observons pas de gain, il est important de noter que leur performance n'est pas impactée par CADiA, mais que celle de l'application *bubble* l'est.

En effet, la limitation de la quantité de LLC utilisée par les applications distribuées permet à l'application *bubble* de bénéficier de cette quantité de LLC libre qui aurait été gaspillée. Pour illustrer cela, nous avons ré-exécuté les benchmarks et collecté le nombre de défauts (*misses*) générés par l'application *bubble* dans les deux scénarios d'exécution (de référence et avec *CADiA*). De 1 à 130 secondes, l'application *bubble* est exécutée seule, puis à 130 secondes, le benchmark SPEChpc est lancé. La figure 3.10 montre le nombre moyen de misses durant l'exécution de l'application *bubble*. Attardons-nous sur les benchmarks Minisweep et HPGMG-FV. Nous pouvons observer qu'entre les points 1 Sec et 130 Sec, le nombre de misses est le même pour les deux scenarios. Mais par la suite, nous observons une réduction des défauts de

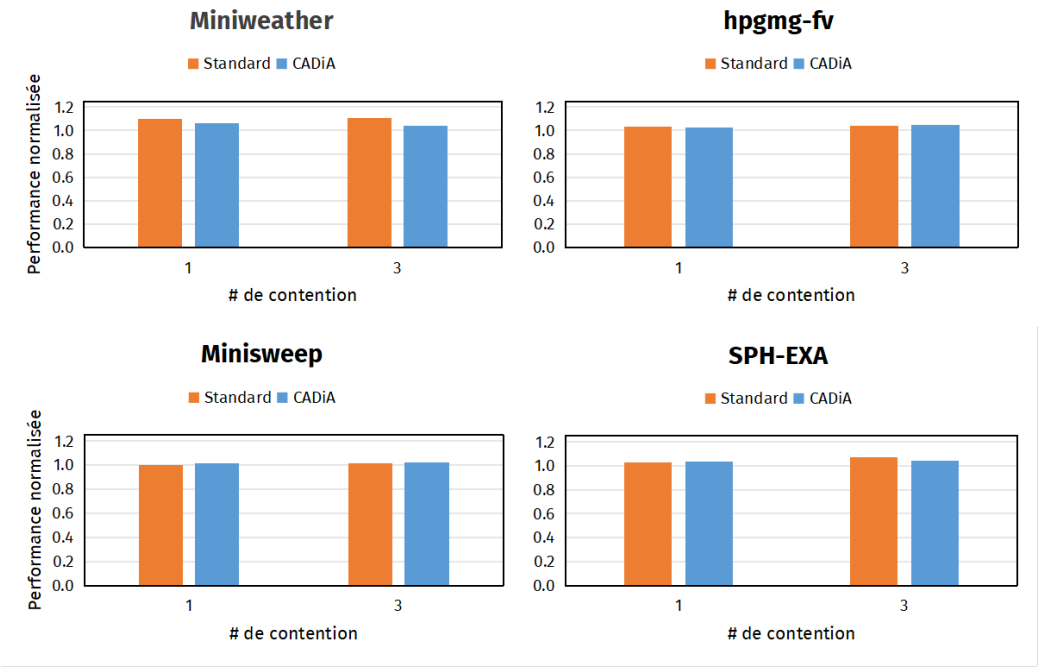


FIGURE 3.9 – Exécution sur 4 nœuds avec la stratégie *Top-Down*

cache avec *CADiA* par rapport à l'exécution standard. Ceci est simplement dû au fait que *CADiA* a détecté que les applications distribuées n'avaient pas besoin de beaucoup de LLC et a alloué ce cache à l'application *bubble*. Cela réduit le nombre de misses de l'application. Alors que durant l'exécution standard, le cache est inutilement utilisé par les applications distribuées. Nous observons le contraire pour les deux autres benchmarks (Miniweather et SPH-EXA) où le nombre de misses a tendance à augmenter ou stagner pour l'application *bubble*.

Top-Down : contention hétérogène

Dans ce scénario, nous avons configuré des contentions hétérogènes sur les nœuds, ce qui signifie que les nœuds ne sont pas soumis au même niveau de contention. Pour cette évaluation, les applications distribuées sont exécutées sur 8 nœuds dont 7 sont en contention : le premier nœud exécute $bubble_2$, le second $bubble_4$, le troisième $bubble_6$, et le dernier nœud exécute $bubble_{14}$. Les résultats obtenus sont présentés dans la figure 3.11. On constate que les gains de performance sont toujours intéressants avec les benchmarks Miniweather et SPH-EXA, de l'ordre de 12% pour le premier et de 3% pour le second. En revanche, aucun gain n'est observé

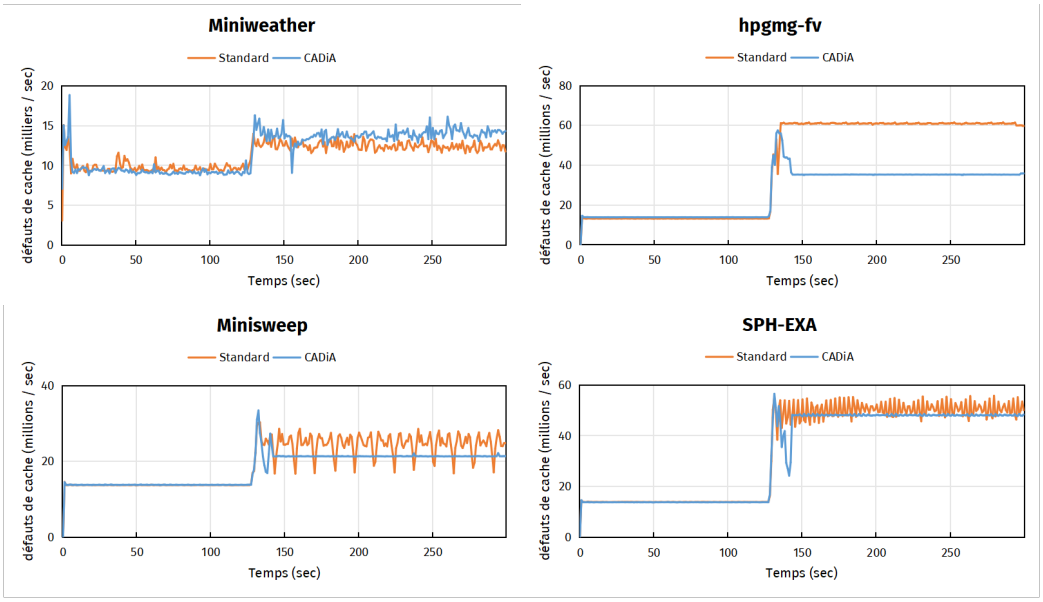


FIGURE 3.10 – Taux d'échec de l'application de *bubble* lors des évaluations.

pour les autres benchmarks. Ce résultat démontre la capacité de *CADiA* à allouer la bonne taille de cache dans le cas d'une pollution hétérogène sur les nœuds.

Bottom-Up : contention homogène

Cette évaluation concerne la stratégie *Bottom-Up*. Pour rappel, l'objectif de cette stratégie est d'allouer équitablement le LLC aux applications s'exécutant sur les nœuds et non d'obtenir la meilleure performance pour l'application distribuée. La figure 3.12 montre les résultats obtenus avec nos benchmarks. On constate que le gain de performance pour Miniweather et SPX-EXA est moins important. Ceci est compréhensible. Intéressons-nous à Miniweather. Miniweather a besoin de 20 voies de cache pour s'exécuter sans impact sur les performances. Avec une allocation équitable et une colocation avec $bubble_{14}$, elle n'aura accès qu'à 13 voies, car $bubble_{14}$ utilise les 8 autres : $13 = 22 \times \frac{20}{(20+14)}$ (voir Algorithme 5 ligne 6). En conséquence, Miniweather a moins de LLC et donc de moins bonnes performances, seulement 5% d'amélioration des performances, ce qui est inférieur aux 13% observés avec la stratégie *Top-Down*. Il en va de même pour SPH-EXA où le gain est plus faible (2%). Cependant, les performances des deux autres applications, HPGMG-FV et Minisweep, ne sont pas affectées par cette

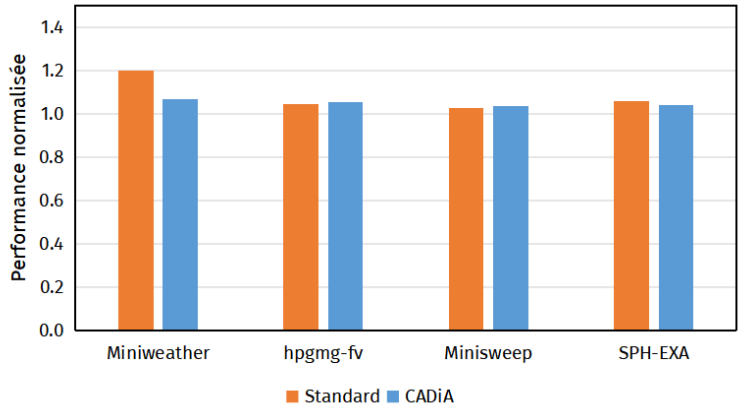


FIGURE 3.11 – Évaluation de la stratégie *Top-Down* avec une contention hétérogène sur les nœuds.

stratégie, ce qui est normal puisqu'elles sont insensibles au LLC.

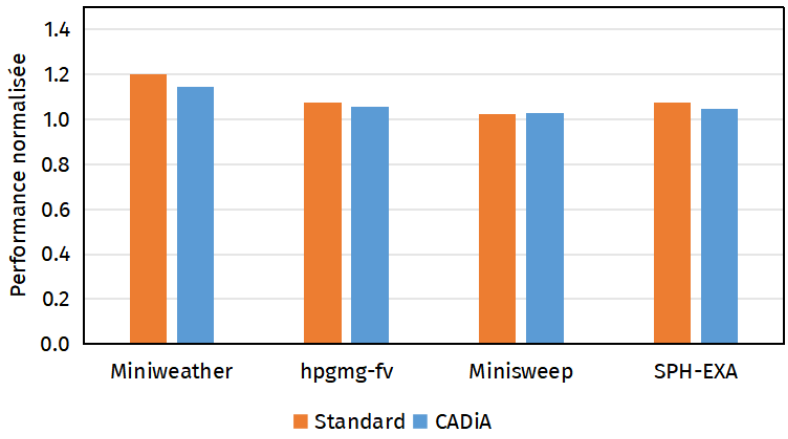


FIGURE 3.12 – Évaluation de la strategie *Bottom-Up*.

3.4.4 Surcoût de *CADiA*

Dans cette section, nous évaluons le coût en ressources matérielles lié au fonctionnement de *CADiA*. Ce coût est divisé en deux catégories :

(1) l'utilisation des ressources CPU et mémoire sur les nœuds de calcul ;

(2) et l'impact sur la performance des applications distribuées, en d'autres termes la charge de profilage de l'application.

En ce qui concerne la catégorie CPU et mémoire, les composants de *CADiA* s'exécutent en tant qu'agents sur les nœuds en utilisant principalement le protocole TCP. Ils sont en mode passif, et la plupart du temps, ils attendent les paquets du réseau. Cela signifie qu'ils ne consomment pratiquement pas de ressources de CPU ou de mémoire sur les nœuds. La quantité de mémoire utilisée est de 3,98 MB pour l'Agent et l'Allocateur et de 3,15 MB pour le Profileur. En ce qui concerne l'impact sur les performances de l'application, *CADiA* a besoin d'agir sur l'allocation du LLC pendant la phase de profilage pour construire les MRCs. Durant cette phase, l'application est impactée par les tailles de LLC qui lui sont allouées. Cependant, le profilage est effectué sur une très courte période, et le surcoût induit n'est pratiquement pas perceptible puisque les applications HPC sont conçues pour s'exécuter pendant une très longue période.

Conclusion

Dans ce chapitre, nous avons présenté *CADiA*, un nouveau système qui alloue le LLC aux applications parallèles distribuées en utilisant les technologies CAT fournies par les processeurs Intel. *CADiA* construit des profils d'utilisation de LLC pour les applications distribuées en collectant des métriques de bas niveau sur les nœuds de calcul et utilise CAT pour partitionner le cache et harmoniser l'allocation de LLC sur tous les nœuds de calcul. Le profil de l'application, qui correspond à sa courbe du taux d'échec (MRC), est construit à partir des métriques extraites des nœuds. Une fois le profilage effectué, *CADiA* estime la quantité de LLC à allouer à l'application en fonction de la stratégie choisie. *CADiA* introduit deux stratégies d'allocation, *Top-Down* et *Bottom-up*. La première donne la priorité à l'application distribuée et alloue la taille optimale de LLC nécessaire pour fonctionner sans impact sur les performances, tandis que la seconde applique une allocation de LLC équitable entre tous les processus sur les nœuds de calcul. Nous avons implémenté *CADiA* dans la bibliothèque Open MPI et l'avons évalué avec la suite de référence SpecHPC. Les résultats obtenus montrent un gain de 13% sur certains benchmarks tout en évitant le gaspillage de LLC sur les nœuds de calcul, et une diminution de 40% du nombre d'échecs sur le LLC pour les applications colocalisées.

Comparé aux solutions existantes, *CADiA* est la seule solution qui propose d'harmoniser l'allocation du cache de dernier niveau (LLC) sur tous les nœuds de calcul pour une application parallèle distribuée. Les

solutions existantes se concentrent sur l'analyse centralisée avec des applications standard ou proposent des techniques de planification des tâches pour les applications distribuées. Aucune d'entre elles ne coordonne l'allocation du LLC entre tous les nœuds de calcul, ce qui est fondamental pour les applications parallèles distribuées.

Chapitre 4

État de l'art

Comme nous l'avons précisé à l'introduction, le sujet de cette thèse porte sur les enjeux de l'allocation de la mémoire cache du processeur dans les environnements distribuée et les environnements cloud de FaaS. Pour cette raison, nous avons structuré notre état de l'art en trois catégories :

(i) la première catégorie, qui comprend les travaux portants sur les techniques de partitionnement et d'allocation du LLC, au moyen de la technologie *CAT* d'Intel ou d'autres mécanismes ;

(ii) la deuxième, qui concerne les techniques de gestion des fonctions cloud en vue de la réduction de leur temps d'exécution et leur temps de démarrage ;

(iii) et la troisième, qui rassemble les méthodes logicielles ayant pour objectif l'allocation des ressources dans le domaine des applications distribuées.

Nous structurerons donc notre état de l'art en ces trois sections.

4.1 Partitionnement et allocation du cache

La technologie d'allocation de cache d'Intel a ouvert la voie à de nombreuses possibilités d'optimisation de la gestion du cache processeur. La majorité des recherches récentes visant à améliorer l'utilisation du LLC s'appuie sur cette fonctionnalité.

Veitch et al. [14] a montré les avantages de l'utilisation de la technologie CAT pour protéger les ressources des fonctions de réseau virtuel (VNF) contre les effets de « voisinage bruyant », en hiérarchisant de manière déterministe les ressources LLC entre les charges de travail concurrentes. Xiao et al. [32] propose CP_{pf}, une approche de

partitionnement du LLC tenant compte du préchargement pour améliorer la gestion du LLC. Noll et al. [33] conçoit un schéma d'allocation de cache à partir d'une analyse empirique de différents opérateurs et intègre un mécanisme de partitionnement de cache dans un système de gestion de base de données commercial. Farshin et al. [34] introduit un système de gestion de la mémoire tenant compte des tranches et propose CacheDirector, une solution d'Entrée/Sortie réseau qui étend DDIO (*Data Direct I/O Technology*) et place l'en-tête du paquet réseau dans la tranche du LLC le plus proche du cœur de processeur concerné. CoPart [35] effectue une analyse dynamique des caractéristiques des applications consolidées sur des serveurs de base et alloue le LLC et la bande passante mémoire de manière coordonnée afin d'améliorer l'équité globale. Xu et al. [36] propose dCAT, une solution de gestion dynamique du cache qui alloue également le LLC sur un serveur aux applications en fonction de leurs besoins en cache à un moment donné. Une famille de politiques de partitionnement du cache basées sur le clustering a été proposée par Selfa et al. [37] pour traiter l'équité à l'aide de la technologie CAT.

Dans [13], une approche est proposée pour construire automatiquement un modèle de prédiction pour les changements de performance des applications en utilisant la technologie CAT, et une technique de gestion de cache dynamique est présentée qui tire parti du modèle de prédiction pour partitionner les ressources de cache et améliorer le débit de l'application. Stewart et al. [38] propose une solution qui consiste à profiler l'utilisation du cache hors ligne, à caractériser les effets des politiques d'allocation du cache à l'aide de techniques d'apprentissage profond et à proposer de nouveaux modèles de performance pour l'allocation à court terme avec des services en ligne. L'objectif est de minimiser le temps de réponse pour tous les services colocalisés. CacheSlicer [39] introduit un support au niveau du cluster pour la gestion du LLC dans le cloud public. DCAPS [40] est un système de gestion de cache dynamique qui surveille et prédit la demande de cache d'une charge de travail multiprogrammée pour réallouer le LLC en fonction d'un objectif de performance, et explore le partage partiel d'une partition de cache entre les programmes pour réaliser l'allocation de cache à une granularité plus fine. Kim et al. [13] propose une approche qui construit automatiquement un modèle de prédiction pour les changements de performance de l'application avec CAT, et une technique de gestion dynamique du cache qui utilise ce modèle de prédiction et partitionne intelligemment le cache afin d'améliorer le débit de l'application.

4.2 Ordonnancement et démarrage des fonctions sans état

Pour réduire la latence de démarrage et le temps d'exécution d'une fonction, certains travaux s'intéressent à l'optimisation du temps de démarrage de l'environnement d'exécution, c'est-à-dire du conteneur, tandis que d'autres se concentrent sur l'ordonnancement des fonctions.

Lee et al. [41] propose un algorithme d'équilibrage de charge optimisé pour FaaS qui fournit une plus grande localité. OpenWhisk [42] utilise également un algorithme d'équilibrage de charge optimisé qui tient compte de la localité et de la réutilisation des conteneurs pour minimiser à la fois la latence de démarrage (démarrage à froid) et le temps d'exécution. Bermbach et al. [43] a mis en œuvre trois approches qui réduisent le nombre de démarrages à froid tout en traitant le service FaaS comme une boîte noire. Dans ces approches, ils utilisent les connaissances sur la composition des fonctions pour déclencher le provisionnement de nouveaux conteneurs avant que le processus d'application n'invoque la fonction correspondante. OFC [20] est un système de cache élastique en mémoire pour les plateformes FaaS, qui estime les ressources mémoire réelles requises par chaque invocation de fonction et conserve la capacité restante pour alimenter le cache, en utilisant des modèles d'apprentissage automatique ajustés pour les catégories de données d'entrée des fonctions typiques. Abad et al. [44] propose un algorithme d'ordonnancement tenant compte des paquets qui tente d'affecter les fonctions qui nécessitent le même paquet au même nœud de travail, augmentant ainsi le taux de succès du cache des paquets et, par conséquent, réduisant la latence de démarrage des fonctions FaaS. Silvia et al. [45] propose une technique de démarrage de fonction, qui restaure les instantanés des processus de fonction précédemment exécutés. FaaSRank [46] est un ordonnanceur de fonctions pour les plateformes FaaS sans serveur basé sur les informations monitorées à partir des serveurs et des fonctions, qui apprend automatiquement les politiques d'ordonnancement par l'expérience en utilisant l'apprentissage par renforcement et les réseaux neuronaux. Barcelona-Pons et al. [47] propose des lignes directrices pour l'orchestration des charges de travail massivement parallèles en utilisant des fonctions sans serveur pour réduire les coûts indirects. FnSched [48] propose un nouvel ordonnanceur centralisé au niveau du cluster et granulaire au niveau du cœur de processeur pour les fonctions sans serveur, ce qui améliore l'élasticité et réduit les interférences.

4.3 Allocation des ressources pour les applications distribuées

Nous avons observé que la plupart des travaux ayant pour objectif l'allocation des ressources pour les applications parallèles distribuées, se concentrent sur l'ordonnancement intelligent des tâches sur les nœuds.

Han et al. [49] abordent la question des comportements d'interférence dans les applications parallèles distribuées. Ils présentent une étude qui caractérise les effets de l'interférence pour diverses applications distribuées sous différents paramètres d'interférence. Ils analysent comment les intensités d'interférence sur plusieurs nœuds affectent la performance globale et, sur la base de cette caractérisation, proposent un modèle basé sur le profilage statique pour la propagation de l'interférence et les comportements d'hétérogénéité. Stavrinides et al. [50] examinent les charges de travail dans un système distribué et utilisent la simulation pour démontrer qu'une stratégie d'ordonnancement qui prend en compte le degré de parallélisme d'un travail surpasse d'autres méthodes qui ne prennent pas en compte les caractéristiques individuelles du travail.

SLURM [51] est un système de gestion des ressources des clusters qui peut s'adapter à des milliers de processeurs. Il est conçu pour être flexible et tolérant aux pannes et fourni un environnement d'exécution de tâches parallèles simple, robuste et hautement évolutif pour les systèmes de clusters. Wang et al. [52] développent SLURM++, un gestionnaire d'application distribué qui est une extension directe de SLURM. SLURM++ met en œuvre une technique de vol de ressources faiblement cohérente basée sur le monitoring pour réaliser l'équilibrage des ressources dans les lancements de tâches d'applications distribuées de calcul à haute performance. D'autre part, Mishra et al. [53] proposent des algorithmes d'allocation de nœuds qui prennent en compte le comportement de communication d'une tâche afin d'améliorer les performances des charges de travail à forte intensité de communication. Les algorithmes proposés visent à minimiser la contention du réseau en allouant des nœuds sur les commutateurs réseaux les moins sollicités. Leur système est également basé sur SLURM.

4.4 Synthèse

Au regard de cet état de l'art, nous pouvons dire que concernant les techniques de partitionnement et d'allocation de cache, les travaux de recherche proposent principalement des politiques d'allocation du LLC faites à l'aide de la technologie CAT ou de systèmes faits à la main, ces politiques étant basées sur des modèles de prédictions obtenus grâce à une analyse empirique ou dynamique des métriques de bas niveau des applications concernées.

En ce qui concerne l'ordonnancement et le démarrage des fonctions sans état (*serverless*), les recherches actuelles proposent deux approches principales. D'une part, des algorithmes d'ordonnancement et d'équilibrage de charge qui exploitent principalement la localité afin de maximiser la réutilisation des environnements d'exécution et des paquets, réduisant ainsi les démarrages à froid et les temps d'exécution des fonctions. D'autre part, divers mécanismes de démarrage de fonction sont également explorés.

Pour ce qui est de l'allocation des ressources dans le cadre des applications distribuées, la plupart des travaux se concentrent sur l'ordonnancement intelligent des tâches sur les nœuds.

Conclusion générale

Nous avons dans cette thèse adressée la problématique de l'allocation de la mémoire cache dans les domaines de cluster et de cloud computing, plus précisément dans les environnements de FaaS et les applications distribuées. Pour ce faire, nous sommes partis de deux hypothèses. La première étant celle selon laquelle il est possible de prédire les besoins en cache d'une fonction FaaS à partir de l'analyse de ses exécutions passées dans le but de lui allouer efficacement le cache. La deuxième, celle selon laquelle il est possible, d'éliminer le gaspillage du LLC et d'optimiser son utilisation par les applications distribuées en harmonisant son allocation sur tous les nœuds de travail sur lesquels s'exécutent ces applications. Nous avons pu vérifier ces hypothèses grâces à des expérimentations qui nous ont servi de motivation par la suite. Cela nous a conduits à la mise en Œuvre de *CASY* et *CADiA*, des systèmes d'allocation intelligents du LLC respectivement pour les fonctions FaaS et pour les applications distribuées. *CASY* utilise l'apprentissage automatique supervisé pour prédire les besoins en cache d'une fonction sans état en se basant sur la taille des données d'entrée, tandis que *CADiA* construit dynamiquement le profil d'une application distribuée et utilise une politique d'allocation qui prend en compte ce profil ainsi que la priorité accordée à l'application. Ces deux solutions nous ont permis d'obtenir des gains de performance allant jusqu'à environ 11% pour *CASY*, et 13% pour *CADiA*.

Dans ce travail, nous explorons un pan de la recherche encore peu exploré. Ainsi, loin d'infirmer les positions des recherches existantes, il s'inscrit plutôt en complément aux limites et aux problèmes non traités par ceux-ci. Nos systèmes peuvent donc parfaitement fonctionner en combinaison avec les autres. Par exemple, un fournisseur de cloud FaaS peut utiliser FaaSRank [46] ou FnSched [48] pour l'ordonnancement de ses fonctions et *CASY* pour l'allocation du LLC aux fonctions qui s'exécutent sur un même nœud. De même, dans un cluster, SLURM [51] ou SLURM++ [52] peuvent être déployés pour le lancement et la gestion des tâches des applications distribuée, et *CADiA* pour l'allocation

harmonisée du LLC à ces applications. Il serait même possible d'utiliser *CASY* et *CADiA* avec une technologie d'allocation de cache autre que la technologie CAT, par exemple DCAPS ou CacheSlicer.

Concernant les limitations de notre travail, nous commencerons tous d'abord par parler de *CASY*. Étant donné qu'il s'appuie sur l'apprentissage automatique supervisé basé sur les données d'entrée, la première limite évidente réside dans l'ampleur du jeu de données d'entraînement. Ainsi, la masse de données utilisée lors de l'entrainement de notre modèle de prédiction limite, du fait de sa taille, l'efficacité de la prédiction de la quantité de cache requise. La deuxième limite concerne les particularités des données d'entrée. En effet, en prenant en compte de nouveaux paramètres spécifiques à chaque type de données, tels que la résolution pour les images et les vidéos, le débit pour les fichiers audio, ainsi que d'autres caractéristiques propres à chaque type de données, il devient possible d'affiner la précision du modèle de prédiction. Par exemple, comme observé sur la figure 2.6, certains fichiers ont une résolution supérieure tout en étant moins lourds, ce qui implique qu'ils nécessitent plus de ressources malgré leur poids moindre. Ensuite, la dernière limite concerne les politiques d'allocation. Nous n'avons pas disposé du temps ni des ressources nécessaires pour entreprendre une évaluation exhaustive de multiples politiques d'allocation et de schémas répartition des voies de cache (par exemple, avec chevauchement et sans chevauchement, statique et dynamique) entre les différents CLOS, puis procéder à leur comparaison.

Pour ce qui est de *CADiA*, la première limite se situe au niveau de la répartition des tâches sur les nœuds de calcul. En effet, notre système ne prend actuellement en compte que les cas où la distribution des tâches est uniforme sur tous les nœuds. On pourrait considérer les situations dans lesquelles la répartition des tâches sur les nœuds de travail serait hétérogène. Ceci afin de réaliser une allocation en fonction du nombre de tâches exécutées sur chacun des nœuds. La seconde concerne le cas où plusieurs applications distribuées seraient exécutées sur un nœud, plutôt qu'une seule. Dans ce cas, on pourrait d'une part faire une allocation équitable (*Bottom-Up*) entre les applications distribuées et les autres applications du système, et d'autre part, d'effectuer l'allocation en fonction d'une hiérarchisation des applications distribuées par ordre de priorité.

Perspectives

Les systèmes d'allocation intelligents de cache, *CASY* et *CADiA*, sont actuellement implémentés en tant que module complémentaire d'OpenWhisk et de OpenMPI respectivement. Cependant, ces systèmes requièrent des ajustements et des compléments à court et à long terme dont nous allons parler dans les sections suivantes.

Perspectives à court terme

En ce qui concerne *CASY* Une première étape d'évolution à court terme consisterait non seulement à enrichir le jeu de données d'entraînement, mais aussi à tester différents modes d'allocation et de répartition des voies de cache (statique et dynamique), dans le but de perfectionner notre système. Ensuite, il est indispensable d'enrichir la base de données de fonctions testées. En effet, bien que nous ayons sélectionné des fonctions dont les profils représentent respectivement à peu près tous les types pouvant être rencontrés dans une situation réelle, il est nécessaire d'éprouver notre système avec un ensemble plus large de fonctions afin d'améliorer sa robustesse.

En ce qui concerne *CADiA*, il est essentiel dans un premier temps de le soumettre à un plus large éventail d'applications afin d'améliorer sa robustesse, tout comme nous le ferrons avec le *CASY*. Ensuite, une évaluation de ce système dans des environnements hétérogènes est envisagée. L'hétérogénéité à considérer se situe notamment au niveau de la fréquence du processeur et de la taille de la mémoire cache. Enfin, nous prévoyons de faire évoluer notre système afin qu'il puisse gérer les situations dans lesquelles plusieurs applications distribuées s'exécutent simultanément sur un même nœud.

Perspectives à long terme

En ce qui concerne les perspectives à moyen et long terme pour *CASY*, notre objectif est de le rendre adaptable afin qu'il puisse être intégré dans n'importe quelle plateforme de FaaS, telles que Google Cloud Functions, AWS Lambda, etc. Ensuite, nous pourrions l'intégrer à l'ordonnanceur ou à l'invoqueur de FaaS, permettant ainsi à l'ordonnancement des fonctions de tenir compte de l'occupation du cache de chaque nœud de travail sur lequel s'exécutent les conteneurs contenant les fonctions, et d'en évaluer l'impact.

En ce qui concerne *CADiA*, nos perspectives à long terme incluent l'objectif de le rendre indépendant de l'intergiciel de cluster. En d'autres termes, nous visons à le rendre compatible aussi bien avec des applications utilisant *OpenMPI* que celles utilisant *Hadoop*, *Spark*, ou d'autres logiciels permettant la mise en œuvre d'applications distribuées. De plus, nous envisageons de tester différentes techniques pour construire la courbe du taux d'échec, afin de l'affiner et de la rendre plus précise. Actuellement, nous nous contentons de modifier l'espace de cache alloué à l'application pour générer cette courbe. Enfin, nous envisageons d'explorer d'autres modes d'allocation de cache pour obtenir une granularité plus fine que celle offerte par la voie de cache.

Bibliographie

[1] G. F. Pfister, *Cluster computing*. GBR : John Wiley and Sons Ltd., 2003, p. 218–221.

[2] H. Liang, W. Chen, and K. Shi, "Cloud computing : programming model and information exchange mechanism," in *Proceedings of the 2011 International Conference on Innovative Computing and Cloud Computing*, ser. ICCC '11. New York, NY, USA : Association for Computing Machinery, 2011, p. 10–12. [Online]. Available : https://doi.org/10.1145/2071639.2071642

[3] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakaroha, "A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2017, pp. 162–169.

[4] A. Kadam, A. Sangale, K. Mahamuni, S. Perle, and S. Kamble, "Advance high performance cluster architecture," in *Proceedings of the International Conference and Workshop on Emerging Trends in Technology*, ser. ICWET '10. New York, NY, USA : Association for Computing Machinery, 2010, p. 999. [Online]. Available : https://doi.org/10.1145/1741906.1742148

[5] P. Somasekaram, R. Calinescu, and R. Buyya, "High-availability clusters : A taxonomy, survey, and future directions," *Journal of Systems and Software*, vol. 187, p. 111208, 2022. [Online]. Available : https://www.sciencedirect.com/science/article/pii/S0164121221002806

[6] C. Ho and H. Ewe, "A hybrid ant colony optimization approach (haco) for constructing load-balanced clusters," in *2005 IEEE Congress on Evolutionary Computation*, vol. 3, 2005, pp. 2010–2017 Vol. 3.

[7] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[8] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker : Lightweight virtualization for serverless applications," in *17th {usenix} symposium on networked systems design and implementation ({nsdi} 20)*, 2020, pp. 419–434.

[9] A. J. Smith, *Design of CPU cache memories*. Computer Science Division, University of California, 1987.

[10] D. Byrne, "A survey of miss-ratio curve construction techniques," *ArXiv*, vol. abs/1804.01972, 2018. [Online]. Available : https://api.semanticscholar.org/CorpusID:4606157

[11] Y. Ye, R. West, Z. Cheng, and Y. Li, "Coloris : a dynamic cache partitioning system using page coloring," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 381–392.

[12] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proceedings of the 4th ACM European Conference on Computer Systems*, ser. EuroSys '09. New York, NY, USA : Association for Computing Machinery, 2009, p. 89–102. [Online]. Available : https://doi.org/10.1145/1519065.1519076

[13] Y. Kim, A. More, E. Shriver, and T. Rosing, "Application performance prediction and optimization under cache allocation technology," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 1285–1288.

[14] P. Veitch, E. Curley, and T. Kantecki, "Performance evaluation of cache allocation technology for nfv noisy neighbor mitigation," in *2017 IEEE Conference on Network Softwarization (NetSoft)*, 2017, pp. 1–5.

[15] P. Sohal, M. Bechtel, R. Mancuso, H. Yun, and O. Krieger, "A closer look at intel resource director technology (rdt)," in *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, ser. RTNS 2022. New York, NY, USA : Association for Computing Machinery, 2022, p. 127–139.

[16] T. Hastie, R. Tibshirani, J. Friedman, T. Hastie, R. Tibshirani, and J. Friedman, "Overview of supervised learning," *The elements of statistical learning : Data mining, inference, and prediction*, pp. 9–41, 2009.

[17] B. Hayes, "Cloud computing," 2008.

[18] D. Kelly, F. G. Glavin, and E. Barrett, "Serverless computing : Behind the scenes of major platforms," *CoRR*, vol. abs/2012.05600, 2020. [Online]. Available : https://arxiv.org/abs/2012.05600

[19] M. Sciabarrà, *Learning Apache OpenWhisk : Developing Open Serverless Solutions*. O'Reilly Media, 2019. [Online]. Available : https://books.google.fr/books?id=gEqgDwAAQBAJ

[20] D. Mvondo, M. Bacou, K. Nguetchouang, L. Ngale, S. Pouget, J. Kouam, R. Lachaize, J. Hwang, T. Wood, D. Hagimont, N. De Palma, B. Batchakui, and A. Tchana, "Ofc : An opportunistic caching system for faas platforms," in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys '21. New York, NY, USA : Association for Computing Machinery, 2021, p. 228–244. [Online]. Available : https://doi.org/10.1145/3447786.3456239

[21] A. AWS, "Amazon lambda pricing," https://aws.amazon.com/lambda/pricing/, 2021.

[22] A. Fuerst and P. Sharma, "Faascache : Keeping serverless computing alive with greedy-dual caching," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2021. New York, NY, USA : Association for Computing Machinery, 2021, p. 386–400. [Online]. Available : https://doi.org/10.1145/3445814.3446757

[23] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My vm is lighter (and safer) than your container," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA : Association for Computing Machinery, 2017, p. 218–233. [Online]. Available : https://doi.org/10.1145/3132747.3132763

[24] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst : Stateful functions-as-a-service," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 2438–2452, jul 2020. [Online]. Available : https://doi.org/10.14778/3407790.3407836

[25] "Intel(r) rdt software package," https://github.com/intel/intel-cmt-cat/, dernier accès : 22 Décembre 2023.

[26] S. Singh, Y.-S. Jeong, and J. H. Park, "A survey on cloud computing security : Issues, threats, and solutions," *Journal of Network and Computer Applications*, vol. 75, pp. 200–222, 2016.

[27] D. Christina and K. Christos, "Bolt : I Know What You Did Last Summer... In The Cloud," in *Proceedings of the Twenty Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2017.

[28] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *et al.*,

"Open mpi : Goals, concept, and design of a next generation mpi implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface : 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11*. Springer, 2004, pp. 97–104.

[29] A. Ihre Sherif, "An evaluation of intel cache allocation technology for data-intensive applications," 2021.

[30] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*, ser. Communications in Computer and Information Science, I. I. Ivanov, M. van Sinderen, F. Leymann, and T. Shan, Eds. Springer International Publishing, 2013, vol. 367, pp. 3–20.

[31] H. Brunst, S. Chandrasekaran, F. M. Ciorba, N. Hagerty, R. Henschel, G. Juckeland, J. Li, V. G. M. Vergara, S. Wienke, and M. Zavala, "First experiences in performance benchmarking with the new spechpc 2021 suites," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022, pp. 675–684.

[32] J. Xiao, A. Pimentel, and X. Liu, "Cp$_{pf}$: a prefetch aware llc partitioning approach," 08 2019, pp. 1–10.

[33] S. Noll, J. Teubner, N. May, and A. Böhm, "Accelerating concurrent workloads with cpu cache partitioning," 04 2018, pp. 437–448.

[34] A. Farshin, A. Roozbeh, G. Jr, and D. Kostic, "Make the most out of last level cache in intel processors," 03 2019.

[35] J. Park, S. Park, and W. Baek, "Copart : Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA : Association for Computing Machinery, 2019. [Online]. Available : https://doi.org/10.1145/3302424.3303963

[36] C. Xu, K. Rajamani, A. Ferreira, W. Felter, J. Rubio, and Y. Li, "Dcat : Dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA : Association for Computing Machinery, 2018. [Online]. Available : https://doi.org/10.1145/3190508.3190555

[37] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, and M. E. Gómez, "Application clustering policies to address system fairness with intel's cache allocation technology," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017, pp. 194–205.

[38] C. Stewart, N. Morris, L. Chen, and R. Birke, "Performance modeling for short-term cache allocation," in *Proceedings of the 51st International Conference on Parallel Processing*, ser. ICPP '22. New York, NY, USA : Association for Computing Machinery, 2023. [Online]. Available : https://doi.org/10.1145/3545008.3545094

[39] M. Shahrad, S. Elnikety, and R. Bianchini, "Provisioning differentiated last-level cache allocations to vms in public clouds," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21. New York, NY, USA : Association for Computing Machinery, 2021, p. 319–334. [Online]. Available : https://doi.org/10.1145/3472883.3487006

[40] Y. Xiang, X. Wang, Z. Huang, Z. Wang, Y. Luo, and Z. Wang, "Dcaps : Dynamic cache allocation with partial sharing," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA : Association for Computing Machinery, 2018. [Online]. Available : https://doi.org/10.1145/3190508.3190511

[41] Y. Lee and S. Choi, "A greedy load balancing algorithm for faas platforms," in *2021 5th International Conference on Cloud and Big Data Computing (ICCBDC)*, ser. ICCBDC 2021. New York, NY, USA : Association for Computing Machinery, 2021, p. 63–69. [Online]. Available : https://doi.org/10.1145/3481646.3481657

[42] "Apache openwhisk is an open source, distributed serverless platform," https://openwhisk.apache.org/, last Accessed : Dec 10, 2021.

[43] D. Bermbach, A.-S. Karakaya, and S. Buchholz, "Using application knowledge to reduce cold starts in faas services," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, ser. SAC '20. New York, NY, USA : Association for Computing Machinery, 2020, p. 134–143. [Online]. Available : https://doi.org/10.1145/3341105.3373909

[44] C. L. Abad, E. F. Boza, and E. van Eyk, "Package-aware scheduling of faas functions," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18. New York,

NY, USA : Association for Computing Machinery, 2018, p. 101–106. [Online]. Available : https://doi.org/10.1145/3185768.3186294

[45] P. Silva, D. Fireman, and T. E. Pereira, "Prebaking functions to warm the serverless cold start," in *Proceedings of the 21st International Middleware Conference*, ser. Middleware '20. New York, NY, USA : Association for Computing Machinery, 2020, p. 1–13. [Online]. Available : https://doi.org/10.1145/3423211.3425682

[46] H. Yu, "Faasrank : A reinforcement learning scheduler for serverless function-as-a-service platforms," Ph.D. dissertation, University of Washington, 2021.

[47] D. Barcelona-Pons, P. García-López, A. Ruiz, A. Gómez-Gómez, G. París, and M. Sánchez-Artigas, "Faas orchestration of parallel workloads," in *Proceedings of the 5th International Workshop on Serverless Computing*, ser. WOSC '19. New York, NY, USA : Association for Computing Machinery, 2019, p. 25–30. [Online]. Available : https://doi.org/10.1145/3366623.3368137

[48] A. Suresh and A. Gandhi, "Fnsched : An efficient scheduler for serverless functions," in *Proceedings of the 5th International Workshop on Serverless Computing*, ser. WOSC '19. New York, NY, USA : Association for Computing Machinery, 2019, p. 19–24. [Online]. Available : https://doi.org/10.1145/3366623.3368136

[49] J. Han, S. Jeon, Y.-r. Choi, and J. Huh, "Interference management for distributed parallel applications in consolidated clusters," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA : Association for Computing Machinery, 2016, p. 443–456. [Online]. Available : https://doi.org/10.1145/2872362.2872388

[50] G. L. Stavrinides and H. D. Karatza, "Scheduling techniques for complex workloads in distributed systems," in *Proceedings of the 2nd International Conference on Future Networks and Distributed Systems*, ser. ICFNDS '18. New York, NY, USA : Association for Computing Machinery, 2018. [Online]. Available : https://doi.org/10.1145/3231053.3231087

[51] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm : Simple linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Berlin, Heidelberg : Springer Berlin Heidelberg, 2003, pp. 44–60.

[52] K. Wang, X. Zhou, K. Qiao, M. Lang, B. McClelland, and I. Raicu, "Towards scalable distributed workload manager with monitoring-based weakly consistent resource stealing," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15. New York, NY, USA : Association for Computing Machinery, 2015, p. 219–222. [Online]. Available : https://doi.org/10.1145/2749246.2749249

[53] P. Mishra, T. Agrawal, and P. Malakar, "Communication-aware job scheduling using slurm," in *Workshop Proceedings of the 49th International Conference on Parallel Processing*, ser. ICPP Workshops '20. New York, NY, USA : Association for Computing Machinery, 2020. [Online]. Available : https://doi.org/10.1145/3409390.3409410

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

Université de Toulouse

Titre : Optimisation de l'allocation de la mémoire cache CPU pour les fonctions cloud et les applications haute performance

Mots clés : Cache CPU, Allocation, FaaS, Application Distribuée Parallèle, Sans Serveur, Calcul Haute Performance

Résumé : Les services informatiques contemporains reposent principalement sur deux paradigmes majeurs : le cluster computing et le cloud computing. Le premier implique la répartition des tâches de calcul entre différents nœuds qui fonctionnent ensemble comme un seul système, tandis que le second se fonde sur la virtualisation de l'infrastructure informatique qui permet sa fourniture à la demande. Dans le cadre de cette thèse, notre attention se porte sur l'allocation du cache de dernier niveau (LLC) dans le contexte de ces deux paradigmes, en se concentrant spécifiquement sur les applications distribuées et les fonctions FaaS. Le LLC est un espace mémoire partagé et utilisé par tous les cœurs de processeur sur un socket NUMA. Étant une ressource partagée, il est sujet à de la contention qui peut avoir un impact significatif sur les performances. Pour pallier ce problème, Intel a mis en œuvre une technologie dans ses processeurs qui permet le partitionnement et l'allocation de la mémoire cache : Cache Allocation Technology (CAT).
Dans ce travail, à l'aide de la technologie CAT, nous examinons d'abord l'impact de la contention du LLC sur les performances des fonctions FaaS. Ensuite, nous étudions comment cette contention dans un sous-ensemble de nœuds d'un cluster affecte les performances globales d'une application distribuée en cours d'exécution. De ces études, nous proposons CASY et CADiA, des systèmes d'allocation intelligents du LLC respectivement pour les fonctions FaaS et pour les applications distribuées. CASY utilise l'apprentissage automatique supervisé pour prédire les besoins en cache d'une fonction FaaS en se basant sur la taille du fichier d'entrée, tandis que CADiA construit dynamiquement le profil d'une application distribuée et effectue une allocation harmonisée sur tous les nœuds en fonction de ce profil. Ces deux solutions nous ont permis d'obtenir des gains de performance allant jusqu'à environ 11% pour CASY, et 13% pour CADiA.

Title: Optimizing CPU cache allocation for cloud functions and high-performance applications

Key words: CPU Cache, Allocation, FaaS, Distributed Parallel Application, Serverless, High performance Computing

Abstract: Contemporary IT services are mainly based on two major paradigms: cluster computing and cloud computing. The former involves the distribution of computing tasks between different nodes that work together as a single system, while the latter is based on the virtualization of computing infrastructure, enabling it to be provided on demand. In this thesis, our focus is on last-level cache (LLC) allocation in the context of these two paradigms, concentrating specifically on distributed parallel applications and FaaS functions. The LLC is a shared memory space used by all processor cores on a NUMA socket. As a shared resource, it is subject to contention, which can have a significant impact on performance. To alleviate this problem, Intel has implemented a technology in its processors that enables partitioning and allocation of cache memory: Cache Allocation Technology (CAT).
In this work, using CAT, we first examine the impact of LLC contention on the performance of FaaS functions. Then, we study how this contention in a subset of nodes in a cluster affects the overall performance of a running distributed application. From these studies, we propose CASY and CADiA, intelligent LLC allocation systems for FaaS functions and distributed applications respectively. CASY uses supervised machine learning to predict the cache requirements of a FaaS function based on the size of the input file, while CADiA dynamically constructs the cache usage profile of a distributed application and performs harmonized allocation across all nodes according to this profile. These two solutions enabled us to achieve performance gains of up to around 11% for CASY, and 13% for CADiA.