# A performance projection approach for design-space exploration on Arm HPC environment

Clément Gavoille




THÈSE

PRÉSENTÉE À

# L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE
DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Clément Gavoille**

POUR OBTENIR LE GRADE DE

# DOCTEUR

SPECIALITÉ : INFORMATIQUE

---

## Approche de projection de performance pour l'exploration de paramètres de conception de l'environnement Arm en HPC

---

Dirigée par **Brice Goglin**

Co-dirigée par **Emmanuel Jeannot**

**Soutenue le : 15 janvier 2024**

**Devant la commission d'examen composée de :**

| | | |
|---|---|---|
| Pr. Raymond Namyst .. | Professeur, Université de Bordeaux ..... | Président du jury |
| Dr. Brice Goglin ..... | Directeur de recherche, INRIA ..... | Directeur de thèse |
| Dr. Emmanuel Jeannot | Directeur de recherche, INRIA, Université de Bordeaux | Co-directeur de thèse |
| Dr. Aleksandar Ilic ..... | Assistant professor, Université de Lisbonne ..... | Rapporteur |
| Pr. Olivier Sentieys ... | Professeur, Université de Rennes ..... | Rapporteur |
| Pr. Estela Suarez ..... | Professeure, Université de Bonn ..... | Examinatrice |
| Dr. Hugo Taboada ..... | Ingénieur-chercheur, CEA, DAM, DIF ..... | Encadrant/Invité |
| M. Conrad Hillairet .. | Ingénieur, Arm ..... | Invité |

# Remerciements

En premier lieu, je remercie mes directeurs de thèse, Brice Goglin et Emmanuel Jeannot, d'avoir accepté de diriger ma thèse mais aussi d'avoir su être présent malgré une première année sans avoir pu avoir d'échanges directs à cause des restrictions sanitaires.

Ensuite, je souhaite remercier de tout cœur Hugo Taboada de m'avoir encadré, accompagné et conseillé tout au long de ma thèse mais aussi sur mon stage de fin d'études d'ingénieur. Je suis très heureux d'avoir été ton premier doctorant.

Je voudrais aussi remercier Fabrice Dupros, Patrick Carribault et Conrad Hillairet qui sont de belles rencontres faites au cours de ma thèse et les remercie pour leurs conseils, nos discussions ainsi que leur participation dans mon encadrement.

Je remercie également l'ensemble des membres de mon jury pour avoir accepté, pour leurs questions et leurs retours sur mon travail.

Je souhaite aussi remercier les stagiaires, doctorants ainsi que l'ensemble des collègues que j'ai eu la chance de côtoyer à Teratec mais aussi au sein de l'équipe TADAAM de l'INRIA Bordeaux pour les échanges, pauses cafés, parties de jeux, ...

Sur un plan plus personnel, je voudrais remercier mes amis de Clématie pour avoir toujours été là pour moi même si je n'ai pas rendu la chose facile.

Ensuite, je veux remercier mes parents pour tout ce qu'ils ont fait pour moi, pour avoir éveillé mon esprit scientifique et encouragé dans cette voie. Sans eux, je n'aurais jamais pu être celui que je suis aujourd'hui et je leur dédie donc ce manuscrit.

Je souhaite aussi remercier mes trois frères: Nicolas, Antoine et Guillaume pour tout ce qu'on a partagé et ce qu'on partagera dans le futur. Je pense aussi à mes neveux et nièces: Cassandre, Marc et Constance dont l'arrivée a égayé mes années d'études.

Enfin, je veux remercier Cassandra qui fut surement la plus belle rencontre que j'ai pu faire pendant ces années de thèse. Sa présence et son soutien sont, sans aucun doute, ce qui m'a permis d'accomplir ce travail. J'espère compter sur ton soutien pour longtemps encore, tout comme tu sais que tu pourras compter sur le

mien.

Bien sur, il y a encore beaucoup de personnes que je n'ai pas pu remercier, je le fait donc ici: Merci à tous d'avoir rendu pas si solitaire cette aventure qu'est le doctorat.

**Titre** Approche de projection de performance pour l'exploration de paramètres de conception de l'environnement Arm en HPC.

**Résumé** La science d'aujourd'hui utilise de plus en plus la simulation pour modéliser et comprendre le monde qui nous entoure. Pour permettre à celles-ci d'être plus rapides, précises et modélisant de plus grand phénomènes, les scientifiques utilisent des supercalculateurs, domaine d'expertise du *Calcul Haute-Performance*. Or, à mesure que la demande en puissance de calcul grandit, ces machines se doivent d'être de plus en plus performantes. Seulement, la réduction de la taille des transistors prévue par la loi de Moore ne suffit plus à diriger l'évolution des processeurs, noyau de la puissance des supercalculateurs. Ainsi, pour continuer à être capable de répondre à cette demande, ces machines deviennent de plus en plus complexes. Et les performances des applications HPC dépendent des interactions entre les nombreux comportements des applications, les architectures des processeurs de plus en plus complexes et des choix faits par les différentes piles logicielle. Les efforts à fournir pour l'optimisation des performances des applications sur les machines sont donc de plus en plus importants.

Une solution pour simplifier ces efforts d'optimisation et obtenir de meilleures performances des applications est de rassembler l'ensemble des acteurs du HPC dans un environnement de **codesign** pour la conception des futures machines. Ainsi, dans un tel environnement où les choix faits par les concepteurs sont dirigés par les intérêts des applications, les processeurs et la pile logicielle seront adaptés aux besoins des futurs utilisateurs. Cela est encore plus important depuis la récente arrivée de l'environnement Arm en HPC, représentant 10% de la puissance de calcul totale du Top500 avec seulement 6 machines. En effet, celui-ci offre une plus grande liberté aux constructeurs dans les choix des caractéristiques des processeurs. Seulement, dans un tel environnement de codesign, il est nécessaire d'utiliser une approche de prédiction de performance prenant en compte l'impact des choix faits par l'ensemble des acteurs pour pouvoir effectuer une exploration viable de l'espace de conception.

Au cours de cette thèse, nous mettons en place une approche de projection de performance adaptée à notre définition d'un environnement de codesign regroupant les acteurs et les aspects des performances des applications en 3 groupes : l'application, la pile logicielle et le matériel. Ce modèle se présente en trois étapes pour effectuer la projection d'un triplet application/pile logicielle/matériel source, et accessible, vers un futur triplet cible d'intérêt, et inaccessible. Ces étapes sont : la caractérisation des performances sur nos trois aspects, suivi de l'analyse des performances sur le triplet source qui va enfin conduire à une projection des performances vers le triplet cible en fonction des différences entre les paramètres de celui-ci et du triplet source. Cette approche est ensuite implémentée à l'aide

d'une représentation fondée sur le modèle Roofline dans laquelle on se concentre sur le maximum de performance atteignable par les triplets et on projette les performances avec une hypothèse de conservation de l'efficacité architecturale. Nous utilisons ensuite ce modèle pour l'analyse et l'exploration de paramètres matériels tels que la taille des vecteurs ou le choix du type de mémoire sur différentes architectures de cœurs Arm. Enfin, nous étendons cette exploration à des architectures multicœurs en affinant la caractérisation de la bande passante et le travail effectués par chaque cœur. L'utilisation de cette approche se concentre sur l'exploration de paramètres applicatifs et de pile logicielle sur une future architecture d'intérêt pour le HPC : le processeur EPI (pour European Processor Initiative).

**Mots-clés** Projection de performance, Environnement Arm, Exploration de l'espace de conception, Codesign

**Laboratoires d'accueil** CEA,DAM,DIF, F-91297 Arpajon, France
Inria Bordeaux Sud-Ouest, 200 Avenue de la Vieille Tour, 33400 Talence

**Title** A performance projection approach for design-space exploration on Arm HPC environment

**Abstract** Today's science increasingly uses simulation to model and understand the world around us. To improve their speed, accuracy, and modeling capabilities, scientists rely on supercomputers, the domain of expertise of *High-Performance Computing*. As the demand for computing power keeps growing, these machines must become ever more powerful. However, the reduction in transistor size predicted by Moore's Law is no longer sufficient to drive the evolution of processors, the core of supercomputer power. Hence, these machines are becoming increasingly complex to answer this increasing demand. The performance of HPC applications depends on interactions between varied application behavior, a complex processor architecture, and the choices made by the software stack. As a result, optimizing applications' performance on these machines is a tedious task.

One solution to simplify optimization efforts and improve applications' performance is to bring together all HPC actors in a **codesign** environment for designing future machines. In an environment where the interests of applications drive the choices made by constructors, the processors and software stack will be adapted to the needs of future users. It is all the more vital with the recent arrival of the Arm environment in HPC, already representing 10% total computing power of the Top500 with just six machines, because this environment offers manufacturers great freedom in their choice of processor characteristics. However, in such a codesign environment, it is mandatory to use a performance prediction approach that accounts for the impact of the choices made by all players to drive the design-space exploration.

In this thesis, we implement a performance projection approach adapted to our definition of a codesign environment that groups the actors and aspects of application performance into three groups: the application, the software stack, and the hardware. This model takes the form of a three-step process for projecting an accessible application/software stack/source hardware triplet onto a future target triplet of interest, which is inaccessible. These steps are performance characterization of our three aspects, followed by performance analysis on the source triplet, which finally leads to a projection of performance towards the target triplet as a function of the differences between its parameters and those of the source triplet. Then, we implement this approach using a Roofline model representation, in which we focus on the maximum performance attainable by the triplets and project performance with an assumption of architectural efficiency conservation. We then use this model to analyze and explore hardware parameters such as hardware vector size and choice of memory type on different Arm core architectures. Finally, we extend this exploration to multi-core architectures by refining the characterization

of the bandwidth and the workload of each core. Then, we use this extension for the exploration of application and software stack parameters on a future HPC architecture of interest: the EPI (European Processor Initiative) processor.

**Keywords** Performance projection, Arm environment, Design-space exploration, Codesign

**Hosting Laboratories** CEA,DAM,DIF, F-91297 Arpajon, France
Inria Bordeaux Sud-Ouest, 200 Avenue de la Vieille Tour, 33400 Talence

# Résumé étendu en français

Dans le but d'apporter des solutions aux grands enjeux contemporains, la science et l'industrie ont besoin de plus en plus de puissance de calcul. De nombreux domaines scientifiques, tels que la climatologie, la biologie ou la physique quantique, utilisent cette puissance à travers la mise en place de simulation sur des supercalculateurs. On parle alors de Calcul Haute-Performance (*High Performance Computing*, ou HPC, en anglais). Et l'architecture de ces supercalculateurs évolue pour devenir de plus en plus performants pour nourrir cette demande croissante en puissance de calcul. Aujourd'hui, on peut les définir comme un ensemble d'un ou plusieurs processeurs dans des nœuds de calculs interconnectés par un réseau rapide. Seulement, avec la fin de la Loi de Moore, les processeurs de ces machines se doivent de devenir de plus en plus complexes pour continuer à être capable de répondre à la demande croissante en puissance de calcul. La principale métrique pour mesurer les performances d'une application HPC est le nombre d'opérations à virgule flottante par seconde ou FLOPS, mais le nombre de FLOPS d'une application ne dépend pas seulement de la puissance de calcul pure offerte par la machine. En effet, elles dépendent de nombreux facteurs touchant des domaines d'expertise différents tels que les choix faits par la pile logicielle, lors du développement du code source ou de leur interaction avec l'architecture du processeur. Ainsi, l'optimisation des applications sur de nouvelles machines est une tâche complexe nécessitant de long temps de développement et une expertise dans de nombreux domaines tels que l'ingénierie logicielle, l'architecture des ordinateurs ou la modélisation de problèmes physiques.

Une solution pour simplifier les efforts d'optimisation des applications ou obtenir de meilleures performances est le développement des futures machines dans un cadre de codesign. L'objectif d'un tel environnement est de construire une machine et de développer une pile logicielle adaptées aux applications des futurs utilisateurs. En outre, avec la récente arrivée de l'environnement Arm dans le monde du HPC, ce besoin de codesign est encore plus fort du fait de la grande liberté offerte par celui-ci dans le choix des paramètres matériels. Seulement, cette liberté dans le choix des paramètres architecturaux se traduit aussi par un besoin de la possibilité d'étudier l'interaction de la pile logicielle et des applications avec

le matériel. Un tel modèle n'est possible que si l'ensemble des acteurs impliqués dans la construction et l'utilisation des supercalculateurs Arm de demain peuvent discuter et avoir des retours sur les choix de conception fait par les autres acteurs. Ainsi, dans notre cas, le point de discussion choisi est sur la **performance** des applications HPC. Il est donc indispensable d'utiliser un modèle de prédiction de performance pour diriger de telles initiatives de codesign.

Idéalement, une telle approche se doit d'être rapide, précise et générique. Seulement, avec la complexité des machines, des logiciels et des codes de simulation d'aujourd'hui, aucun modèle ne peut coupler ces trois caractéristiques. On se doit donc de faire des compromis. Il existe donc de nombreuses approches possibles en fonction des compromis que les acteurs de l'environnement de codesign peuvent se permettre d'effectuer ou non. Les trois principales approches sont :

- La *Simulation cycle-par-cycle*, précise et générique. Elle fait le choix de simuler le comportement d'une machine exécutant une application sur une autre machine. En effet, cette précision se fait au détriment de la rapidité de la prédiction du fait des nombreuses interactions à modéliser ;

- Les *Modèles dépendants d'une application* qui sacrifie l'aspect générique de l'approche pour modéliser le comportement d'une application en particulier et ainsi effectuer des prédictions rapides et précises sur un grand nombre d'architectures ;

- Les *Modèles analytiques* font, eux, le choix d'accepter de perdre en précision des prédictions pour la rapidité et la généricité. Ils se fondent sur une modélisation mathématique, statistique ou des méthodes d'apprentissage pour obtenir une estimation de l'évolution de la performance en fonction des paramètres d'entrée.

Du fait de ces nombreuses approches possibles, effectuer le choix d'une approche ou d'une autre va dépendre des besoins de l'environnement de codesign.

C'est l'objectif de cette thèse : **Mettre en place une approche de prédiction de performance adaptée à l'exploration des paramètres de conception dans un cadre de codesign**.

Ce manuscrit présente les contributions effectuées pendant trois années de thèse dans le but de mettre en place un environnement de codesign, définir une approche de projection de performance adaptée et l'implémenter pour l'exploration de paramètres à différents niveaux. En premier lieu, l'environnement de codesign est défini en séparant les acteurs et les aspects des performances des applications HPC en trois groupes distincts: les applications, la pile logicielle et le matériel. Suite à cette définition, le besoin de pouvoir analyser les impacts des choix fait par les acteurs de chacun de ces aspects entraîne le choix d'utiliser une **approche de**

**projection de performance** pour diriger l'exploration des parametres. Ensuite, nous avons effectué une première implémentation de cette approche avec pour objectif d'étudier l'impact de paramètres matériels sur les performances mono-cœurs des applications HPC. Cette implémentation se fonde sur une modélisation de type Roofline des performances pour l'analyse et la projection de performances. Les explorations de paramètres accomplies avec cette approche concluent sur différentes optimisations matérielles adaptées aux choix des applications et de la pile logicielle. Enfin, cette implémentation est ensuite naturellement étendue pour la projection des performances multi-cœurs des applications HPC. Cette extension conserve l'aspect Roofline mais raffine la caractérisation de la bande-passante et du travail de chaque cœur pour s'adapter aux difficultés apportées par une exécution en environnement multicœur. Grâce à cette approche, nous avons pu établir et reproduire une voie d'optimisation logicielle ou applicative possible des performances sur une architecture de processeur d'intérêt pour le futur proche de l'environnement Arm en HPC.

## Définition de l'environnement de codesign et mise en place de l'approche de projection de performance

Afin de faire le choix d'une approche de prédiction de performance adaptée aux besoins de notre environnement de codesign, il faut avant tout définir cet environnement et ses besoins. Ainsi, nous avons fait le choix de séparer les acteurs et les aspects des performances des applications HPC en trois groupes :

- L'*application* qui comprend le code source, les structures de données utilisées, les schémas numériques, ... ;
- La *pile logicielle* comprenant l'impact des compilateurs, des librairies d'exécution, ... ;
- Le *matériel* qui correspond à l'architecture de la machine, sa hiérarchie mémoire, ...

Du fait du besoin de caractériser indépendamment les impacts de chacun de ces aspects rapidement et de façon générique, nous avons fait le choix d'utiliser un modèle analytique pour la prédiction de performance. Nous définissons ce modèle comme une approche de projection de performance.

En effet, celle-ci vise à estimer l'impact des différences entre un triplet application/pile logicielle/matériel source, accessible, et un tel triplet cible, inaccessible, sur les performances finales. Cette méthodologie peut se séparer en trois étapes

: la caractérisation des paramètres de chacun des aspects, l'analyse des performances du triplet source et la projection des performances vers le triplet cible. La caractérisation consiste à récupérer des métriques d'intérêt représentatives des performances sur les trois aspects des deux triplets. Ensuite, ces métriques sont utilisées pour l'analyse des performances du triplet source afin de conclure quant à leur impact sur les performances. Enfin, suite à cette estimation de l'impact de ces métriques, la projection de la performance mesurée sur le triplet source se fait grâce aux différences entre les métriques des deux triplets. Une fois que l'environnement et la méthodologie de projection utilisée pour l'exploration des paramètres de conception, nous l'implémentons ensuite dans un premier environnement simple : l'exécution monocœur d'une application.

## Exploration matérielle en environnement monocœur

Même si les applications HPC sont actuellement exécutées sur plusieurs cœurs en parallèle, l'étude de leur performance monocœur reste capitale. En effet, selon la loi d'Amdahl, les performances multicœurs d'une application sont limitées par le temps de calcul qu'elle effectue en monocœur. Ainsi, l'optimisation des performances monocœurs permet d'améliorer la capacité d'une application à passer à l'échelle.

Or, dans le cas d'une exécution séquentielle, les performances dépendent notamment des paramètres matériels et de comment la suite d'instruction interagit avec celui-ci. Ainsi, l'exploration de l'impact de paramètres matériels sur les performances monocœurs des applications HPC est vitale pour les applications d'aujourd'hui.

Dans cet objectif, nous implémentons la méthodologie définie précédemment à l'aide d'une représentation de type Roofline des performances. Cela se traduit par une modélisation des limites imposées par le matériel sous forme de *rooflines*, obtenus à l'aide des benchmarks STREAM et High Performance Linpack. Dans le cas d'une machine cible inaccessible, nous supposons qu'ils sont, soit fournis par le constructeurs, soit extrapolés. Ensuite, l'analyse du binaire (représentatif des aspects application et pile logicielle) permet de pondérer ce roofline pour être plus représentatif de la limite actuelle du mix d'instructions. Enfin, nous caractérisons le comportement de l'application à l'aide de son Intensité Opérationnelle (OI) pour les différents niveaux de mémoire. La projection se fait naturellement avec une hypothèse de conservation de l'efficacité par rapport à son roofline pondérée. Les différentes OI et rooflines (pour chaque niveau de mémoire) peuvent conduire à l'obtention d'un intervalle de performance.

La validation de cette approche entre 3 cœurs d'architecture Arm (Marvell ThunderX2, Neoverse N1 et Fujitsu A64FX) met en avant la capacité de cette

approche à obtenir des prédictions justes dans le cas de machines présentant une micro-architecture proche. Elle montre aussi la limitation de cette approche dans le cas d'une projection vers une machine trop éloignée en termes de micro-architecture, démontrant que l'hypothèse de conservation de l'efficacité est trop grossière dans ce cas.

Enfin, l'exploration de paramètres matériels est accomplie en utilisant nos trois cœurs de référence en machine source et en modifiant la taille des vecteurs, le type de mémoire ou en combinant ces deux modifications. Cette étude est menée sur trois proxy-applications de référence de la suite CORAL: MiniFE, Quicksilver et LULESH. Cette exploration montre que le choix de la valeur d'un paramètre va dépendre de l'application que l'on cible ainsi que de l'architecture de référence et du choix du compilateur. Par exemple, une application comme Quicksilver ne bénéficie aucunement de l'augmentation de la taille des vecteurs alors que LULESH peut observer un gain de performance allant jusque $\times 2$ si on part d'une architecture type ThunderX2.

## Exploration applicative et logicielle en environnement multicœur

Dans ce dernier chapitre, l'objectif est d'étendre l'approche de projection monocœur à l'aide d'une modélisation Roofline précédente à une situation en exécution multi-cœur. En effet, au sein d'un nœud de calcul, les applications d'aujourd'hui essayent d'utiliser au maximum les ressources offertes par l'ensemble des cœurs et de la mémoire à l'échelle du nœud. Cette projection correspond donc à un cadre plus réaliste de l'exécution d'une application sur un processeur d'un supercalculateur.

Cela est accompli en analysant le travail de chaque cœur en concurrence sur les mêmes ressources de calcul. Cela nécessite aussi d'affiner la caractérisation de la bande-passante, celle-ci ayant un impact encore plus important sur les performances des applications en multicœurs. L'approche consiste donc à une analyse du travail de chaque cœur, suivis d'une projection sur les cœurs du triplet cible puis à une sommation de l'ensemble des intervalles de projection obtenus. Une limitation de cette approche est que le nombre de cœurs entre le triplet source et le triplet cible doit être égal.

Pour les travaux de validation de cette approche, nous utilisons deux processeurs similaires mais séparés d'une génération : les processeurs Graviton 2 et Graviton 3 d'Amazon Web Services. Ce ne sont pas des processeurs directement HPC mais l'évolution de ceux-ci est représentative de la future évolution des processeurs HPC Arm suivant la roadmap Neoverse. Dans ce cas de figure, la projection entre ces processeurs admet une erreur d'au plus 20% pour deux kernels de la

suite de validation NAS. Cela s'explique notamment par un résultat similaire à la validation monocœur : une différence de comportement de la micro-architecture entre les deux générations, ce qui n'est pas considéré dans notre hypothèse de conservation de l'efficacité. Ces travaux de validations sont donc encourageants et justifient l'exploration de paramètres logiciels et applicatifs vers un processeur d'intérêt pour le HPC : l'European Processor Initiative (EPI).

Les caractéristiques précises de celui-ci étant inconnues au moment de l'écriture de ce manuscrit, nous utilisons les informations disponibles pour représenter l'EPI à partir d'un Graviton 3 avec une mémoire HBM rapide, obtenue sur un nœud d'A64FX, et une taille de cache partagé L3 plus importante. Les seules améliorations se situent donc au niveau des caractéristiques mémoires du processeur. L'objectif est donc d'étudier comment il serait possible d'utiliser des paramètres applicatifs et logiciels pour utiliser ce gain en mémoire sur deux applications : LULESH et LAMMPS DIFFUSE. En effet, l'étude de différents compilateurs et jeux d'instructions sur LULESH montre une piste contre-intuitive : la réduction artificielle de l'intensité opérationnelle pour mieux utiliser les ressources mémoires. Nous avons donc choisi de reproduire ce comportement à l'aide d'un code synthétique dans un environnement permettant d'utiliser deux mémoires différentes avec la même architecture : le processeur Knights Landing d'Intel. Ces travaux ont montré qu'une telle piste d'optimisation, quoique contre-intuitive, est viable et montre que l'utilisation de projection de performance pour le codesign peut ouvrir la voie à de nombreuses possibilités d'optimisation des applications HPC pour les futures machines Arm.

# Conclusion

Le développement de futurs supercalculateurs dans un environnement de codesign est une solution pour simplifier l'optimisation des applications HPC sur ces machines. Dans ce cas-là, il est nécessaire d'utiliser un modèle de prédiction de performance adapté à l'environnement de codesign pour que l'ensemble des acteurs impliqués puissent discuter sur les meilleurs paramètres de conception à choisir dans le domaine d'expertise de chacun.

Cette thèse montre qu'il est avant tout nécessaire de définir un tel environnement pour ensuite choisir une approche de prédiction adaptée. Grâce à différentes implémentations de la méthodologie de projection de performance définie dans la première contribution, les travaux effectués ont permis de montrer différentes possibilités d'optimisation et que cela doit intégrer les choix de l'application, de la pile logicielle et du matériel pour pouvoir avoir une direction claire quant aux choix effectués lors de la conception des machines HPC de demain.

# Contents

# List of Figures

# List of Tables

# Introduction

To understand our world, scientists use computer simulations to model phenomena from the infinitely small to the infinitely large. Delivering the computing power needed for these simulations is the role of High-Performance Computing (HPC) through the development of supercomputers. As science needs more and more computing power, HPC supercomputers are becoming more and more complex to deliver enough performance. Moreover, a faster application execution time opens the possibility of simulating phenomena more accurately or at a bigger scale. However, the execution time of an application running on a supercomputer depends on many complex interactions at different levels. As such, optimizing a complicated application on a complex supercomputer is a tedious task that requires deep knowledge in many different expertise fields. As the Arm HPC environment is recent but already represents a good share (10%) of the top computing power, optimizing applications in this environment is one of the challenges of today's HPC.

One solution to simplify optimization efforts and better performance of applications is to design future machines according to the specific application needs of their future users thanks to codesign. However, a codesign environment is only possible if each actor has feedback on the impact of their and the other's choices on the performance of applications. Hence, using a performance prediction model is mandatory to enable codesign initiatives as a discussion opener between every actor involved in a codesign initiative. Many prediction approaches are used in today's HPC, but, they are not necessarily adapted to a codesign environment. Proposing such a prediction approach adapted to our needs is the main contribution of this manuscript.

The first part of this manuscript presents the context of this study. After a general presentation of the HPC context, applications, and machines in Chapter 1, we describe the hardware and software stack impact on the performance of HPC application in Chapter 2 and 3. Then, we propose the solution of codesign from this context presentation and present the possible state-of-the-art prediction methods for codesign in Chapter 4.

In the second part of this manuscript, we present a performance projection

approach that is coherent with our vision of codesign and apply this methodology to explore the design space around various Arm architectures at different levels. In our definition of such an environment in Chapter 5, we split the performance of HPC applications into three aspects: the application, the hardware, and the software stack. According to these needs, relying on a mechanistic projection approach between a source application/software stack/hardware triplet and a target triplet is a viable solution for our problem. Our projection workflow can be divided into characterization, performance analysis on the source triplet, and projection on the target triplet.

With such a workflow, we first explore the impact of hardware parameters on single-core performance with analysis and projection relying on a Roofline representation of performance. Chapter 6 sets the Roofline single-core projection approach and uses our implementation to explore various hardware parameters such as hardware vector length or the change of memory type on three source Arm cores architectures: Neoverse N1, Fujitsu A64FX, and Marvell ThunderX2.

Finally, we adapt the roofline projection approach to a multicore environment thanks to a finer characterization of the bandwidth and the workload of each thread of the applications. Chapter 7 defines this multicore extension in addition to a preemptive study on the possible impact of different compilers and numerical methods when targeting a node supposedly close to the European Processor Initiative (EPI), unavailable at the time of the writing.

# Part I

# Context

# Chapter 1

# Context of High Performance Computing

Nowadays, science relies more and more on simulation and modelization to have a better understanding of our environment. One such example is climate science researchers that use this computing power for climate simulations. Furthermore, more computing power available translates to faster and more accurate simulations. But this need for computing power is not limited to climate science as, today, every science field such as aerospace or quantum physics relies on computing methods and simulation. And providing this computing power is the objective of High-Performance Computing (HPC) supercomputers.

The first supercomputers were introduced in the 1960s. But their design has little similarity to today's supercomputers. Due to the increasing demand for computing power, their design has changed a lot to adapt. Figure 1.1 presents the evolution of the performance (in GFLOPS or billions of floating-point operations per second) of the sum of all 500 supercomputers of the Top500 ranking between 1993 and June 2023. As such, the available computing power has been growing exponentially since the first Top500 ranking in 1993.

Looking at the evolution of the top-ranked supercomputer, we can link its growth to the Empirical Moore's Law. However, this law is not viable anymore [93] yet the performance is still increasing and getting over the exaflops barrier ($10^{18}$ floating-point operations per second) with the American supercomputer Frontier. Moreover, overcoming performance barrier does not only depend on the computing speed of supercomputers. They also need to resolver the constraints imposed by the difference between memory and computing speed [81] and the power consumption and heat dissipation of CPUs [28]. It is, respectively, the *memory wall* and the *power wall*. Accordingly, to keep attaining higher performance and overcoming the barriers, the machines are becoming increasingly complex.

However, as supercomputers become increasingly complex, the HPC applica-

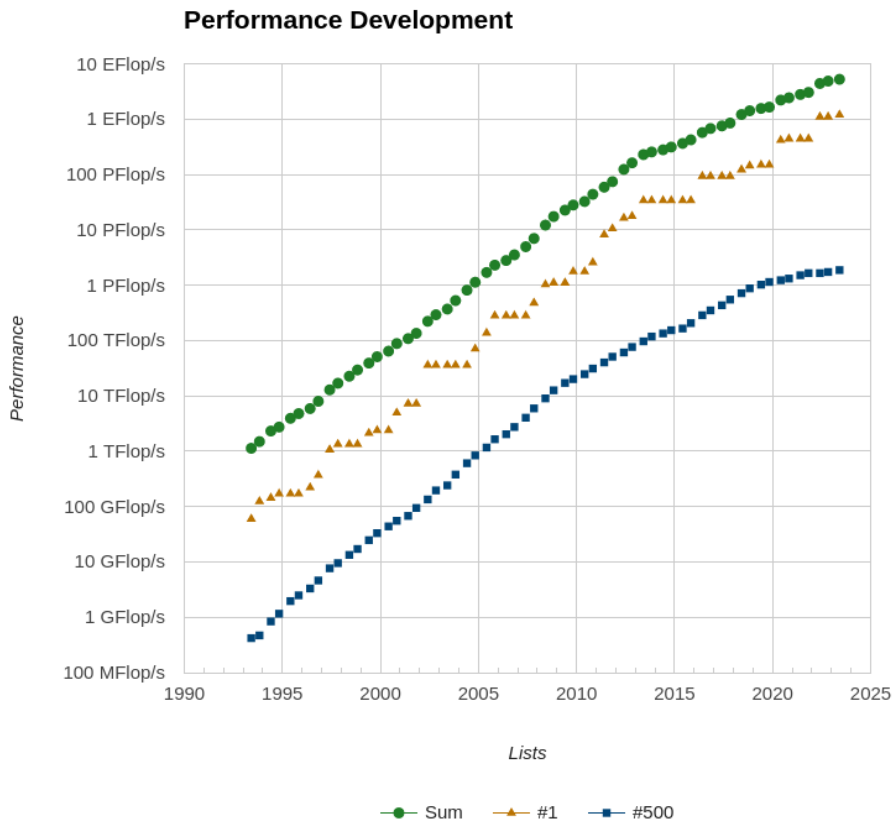


Figure 1.1: Evolution of the performance of the Top 500 #1 and #500 supercomputer and the sum of all ranked machines from 1993 to June 2023
Source: [19]

tions also need to adapt to these machines to be efficient. Hence, in today's HPC context, **how are HPC applications optimized for performance on these supercomputers?**

## 1.1 HPC applications

HPC applications are complex workloads that simulate many interactions between numerous physics phenomena. These workloads depend on a complex workflow with several steps before the final simulation results. Making optimization efforts

on the entire workflow is a difficult and tedious task because of the complex behavior and their long runtime. One such example is the WarpX Particle-In-Cell code awarded by the 2022 Gordon Bell prize [53]. Optimization efforts on this application were accomplished over several layers with many techniques such as load balancing, mesh refining, and parallelization. These optimization efforts on four supercomputers required an international team of researchers specialized in various domains.

One solution used to fasten and simplify these optimization efforts is the development of mini-apps and proxy apps. These codes aim to reproduce the main application's behavior or some parts of this application at a smaller scale. Hence, as the analysis time is much shorter and the execution flow is less complex, optimization of these codes is a faster and easier task than working on the full application. Then, optimization patterns found in these codes can be translated into the main application.

One of the main initiatives of this kind is the CORAL Benchmarks suite [18]. This benchmark suite consists of numerous mini-apps and proxy apps representing the HPC workload of the Lawrence Livermore National Laboratory (LLNL). It aims to converge the optimization and codesign efforts on these representative HPC workloads.

This suite is now widely used in HPC research and also in the work presented in this manuscript. We focus our validation and exploration efforts on some applications of this benchmark suite: LULESH, Quicksilver, MiniFE, and LAMMPS. All these applications are open-source through public git repositories [10, 16, 12, 15].

### 1.1.1 LULESH

LULESH (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics)[58] is a widely used proxy application in HPC for performance prediction, analysis, and benchmarking. It is a simplification of the behavior of shockwave propagation through solid materials. It requires modeling hydrodynamics with a simple analytic answer. LULESH approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a cartesian mesh. Figure 1.2 depicts a 2D visualization of the phenomenon modelized in 3D by LULESH.

As a proxy application, LULESH represents only a minimal fraction of the LLNL's Ale3D [82] workflow described in Figure 1.3

It is a simplified application but a good representation of the numerical algorithms, data motion, and programming style typical in complex multi-material system deformation scientific simulations. This simplicity enabled LULESH to be developed using numerous programming models such as OpenMP, MPI, CUDA, or in several languages like Fortran, C/C++, or Rust. Hence, optimizing perfor-

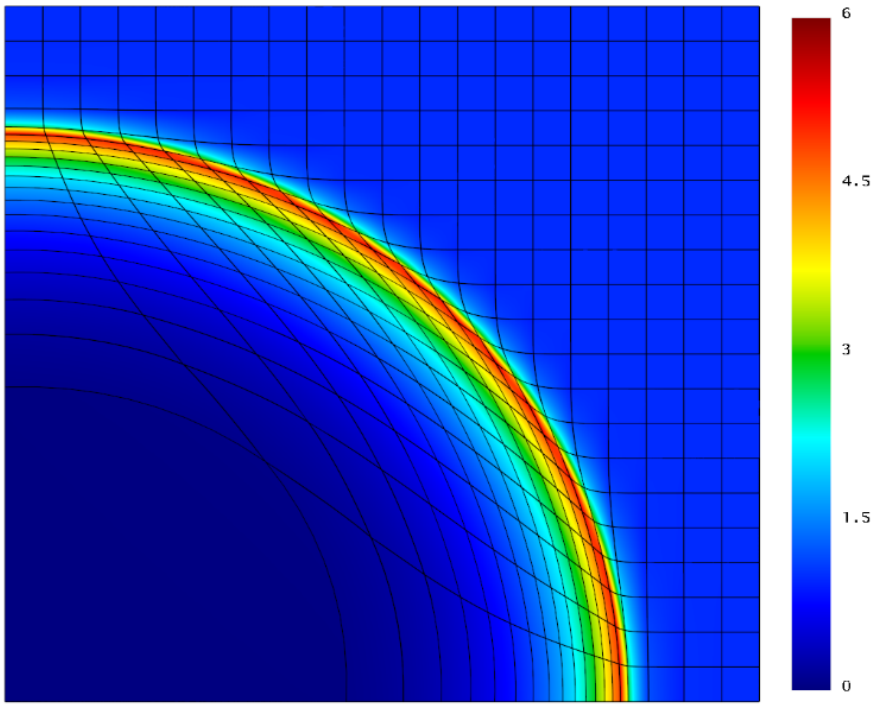


Figure 1.2: Simple Sedov Blast wave problem 2d modelization. LULESH represents this problem on a 3d cartesian mesh
Source: [58]

mance and exploring different programming paradigms on LULESH is easier and faster than on the whole Ale3D application while still being representative of the behavior of one of the challenge problems of HPC.

### 1.1.2 Quicksilver

Quicksilver is a proxy application of the Mercury [29] application that simplifies a dynamic Monte Carlo particle transport problem. Monte Carlo methods [73] are also widely used in HPC applications for statistical sampling or integration approximation. Particle transport is one of its many uses.

Quicksilver modelizes three possible interactions of particles with matter: Absorption, Scattering, and Fission (See Figure 1.4).

The different possible behavior and the randomness of the Monte Carlo method generate a very conditional code with complex memory access patterns that is difficult to optimize and predict. This particular behavior justifies developing a proxy application with very flexible inputs.

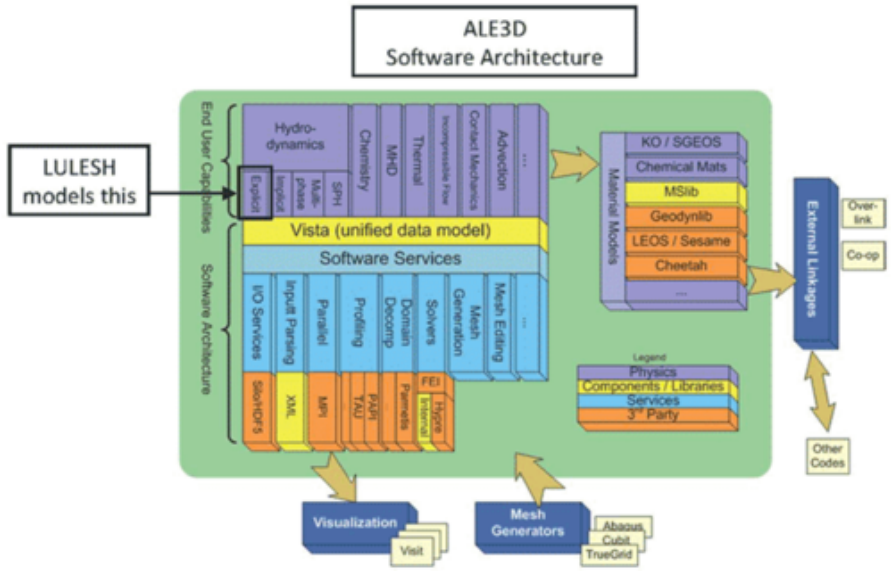


Figure 1.3: Ale3D complex workflow. LULESH only represents a small fraction of this workflow by modelizing hydrodynamics equations with an explicit method.
Source: [58]

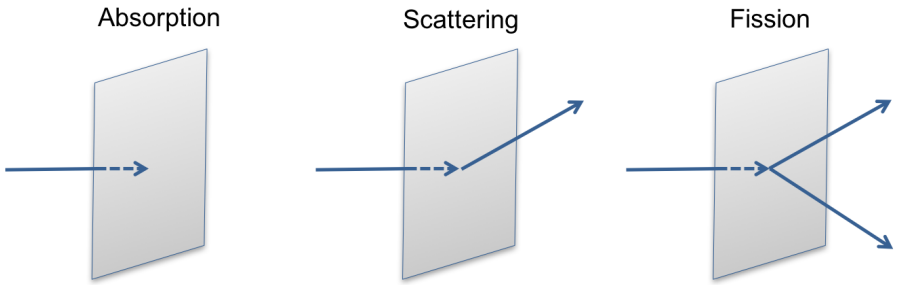


Figure 1.4: The three possible interactions with matter of the randomly positioned particles modelized by Quicksilver.
Source: [86]

### 1.1.3 MiniFE

MiniFE [11] is a mini-application implementing representative kernels of implicit finite-element applications part of the Mantevo Project of Sandia National Laboratory [26]. Here, the mini-application does not model real physics problems. However, many applications rely on an implicit solution of a nonlinear equations system. And they often use a variation of a conjugate gradient solver to resolve such systems.

MiniFE implements the conjugate gradient algorithm in four steps:

- Element-operators computation (source vector, diffusion matrix);
- Assembly (scattering element-operators into sparse matrix and vector);
- Conjugate Gradient solve (sparse matrix-vector product);
- Vector operations with BLAS (Basic Linear Algebra Subprograms).

Hence, finding optimization patterns on the various computing steps of such a linear system-solving application helps reduce the computing time of a widely used computing method and impacts the performance of numerous HPC applications.

### 1.1.4 LAMMPS

LAMMPS [94], for Large-scale Atomic/Molecular Massively Parallel Simulator, is not a single mini-application in itself. It is a classical molecular dynamics code with a focus on materials modeling. It can compute many systems such as solid-state materials (metals, semiconductors), soft matter(polymers, biomolecules), or coarse-grained systems with a variety of interatomic potentials. Figure 1.5 presents three model examples simulated with LAMMPS.

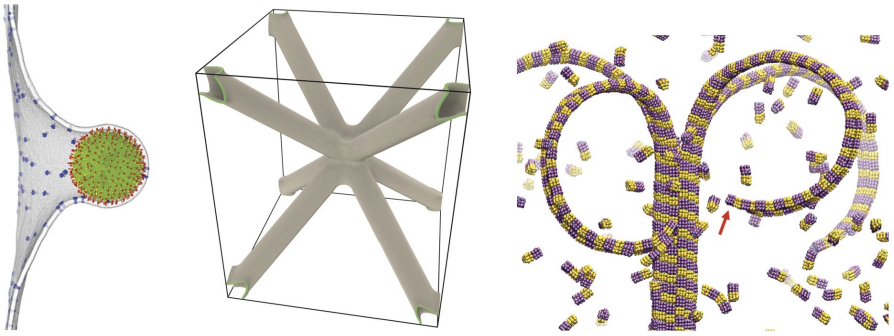

Figure 1.5: (Left) Coarse-grained model of a 100 nm virus-like particle budding through interaction with a cell membrane. Transmembrane proteins are shown in blue. (Middle) A hollow metal strut (200 nm on a side) is used to measure the mechanical strength and stiffness of ultra-lightweight nanoengineered materials. (Right) Coarse-grained simulation of catastrophic depolymerization of alpha-beta-tubulin.
Source: [94]

The flexibility of this framework allows us to use it to simulate simple phenomena and design many mini-apps. For example, we study a small particle diffusion

benchmark DIFFUSE obtained with LAMMPS in our experiments. It allows us to study the impact of different numerical schemes implemented in LAMMPS to compute the same solution.

To conclude, we have seen that real HPC applications rely on a complex workflow. The optimization of these workflows requires a large amount of effort from experts in different domains. Hence, using mini-apps and proxy apps simplifies this work of performance optimization and exploration around different environments. Now that we have defined a representative panel of the HPC applications of interest through mini-apps and proxy apps, we present the current situation of today's HPC supercomputers that execute the workloads represented by these applications in the next section.

## 1.2 HPC supercomputers landscape

Supercomputers have changed a lot through their histories. Nowadays, all the Top500 supercomputers share a similar structure: a massively parallel architecture of computing nodes stored in racks linked by a high-performance interconnect.

The following Figure 1.6 presents the composition of a Fugaku supercomputer rack. It is composed of 432 racks of 384 nodes split into 8 shelves with a 48 cores Fujitsu A64FX CPU per node. Fugaku consists of 158,976 nodes and 7,630,848 cores. With all these cores, Fugaku attained a performance of 442.01 PFLOPS, ranking 2nd in the Top500 of June 2023.

Using every node efficiently is primordial to obtain most of the computing power delivered by these massively parallel machines. Using parallelism in a machine is translated to many communications between computing nodes for some applications. However, communication between nodes is slow, and interconnect network performance can limit these applications' performance. Moreover, because the performance of applications is measured in FLOPS, or floating point operation per second, communication time is not seen as useful for the application. Furthermore, if a node is waiting for data because of communication time, it is also not doing any computation. Hence, optimizing the inter-nodes efficiency by implementing load balancing algorithm [101] or reducing communications time [47] of applications leads to significant performance gain on some applications. This is one of the many optimization patterns that have been found on the WarpX code by implementing particular parallelization strategies and load balancing adapted to the different machines that executed the code for the Gordon-Bell submission [53].

However, applications's performance is not only limited by inter-node commu-

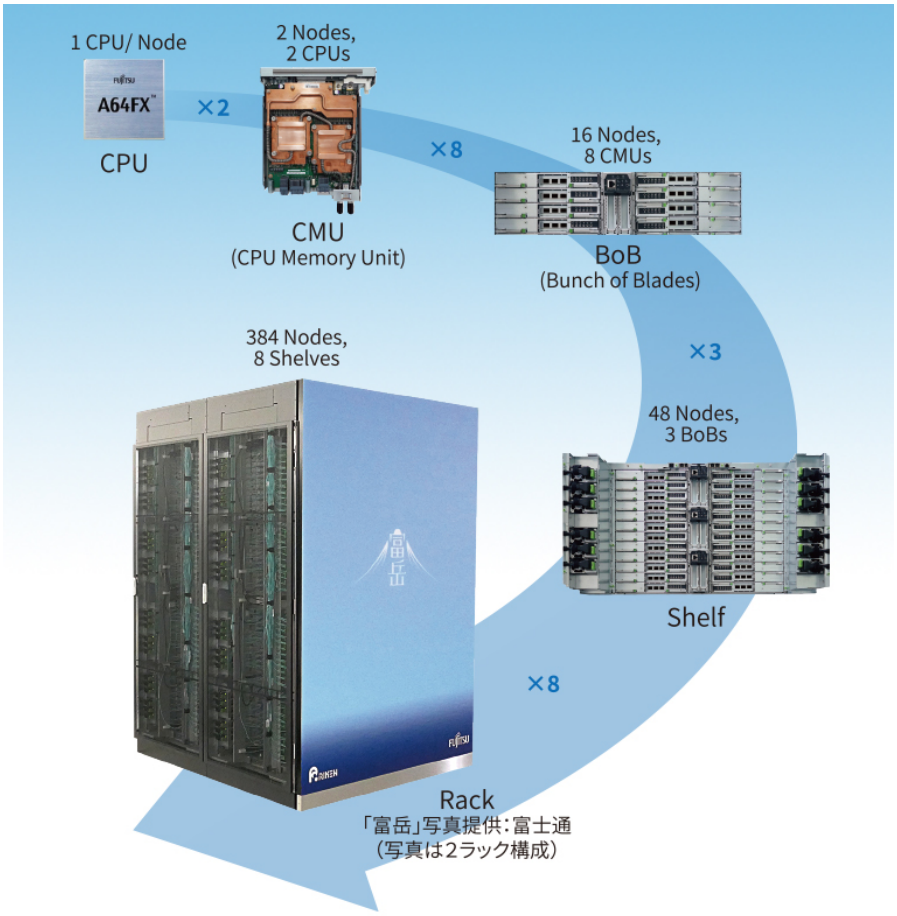


Figure 1.6: Composition of a Fugaku rackSource: [17]

nications. The computing nodes' performance and efficiency can also be a limiting factor in applications' performance. Thus, an efficient usage of the node architecture is mandatory for applications to attain the highest performance on a particular machine.

In this work, we focus our study on the optimization efforts of homogenous nodes that rely on one or more general-purpose CPUs for their computing power. At first glance, with more than 95% of the June 2023 Top500 systems relying on x86 architecture (see Table 1.1), the architectural landscape of HPC CPUs seems homogenous hence simplifying optimization efforts on node efficiency.

Although present in only 1.2% of systems, the Arm architecture accounts for 9.8% of total Top500 performance. Its significant performance share is an accomplishment of the recent introduction of the Marvell ThunderX2 processors and, especially, the Fujitsu A64FX CPUs in Fugaku. Moreover, with future projects

| Architecture | Number | System Share | Performance Share |
|---|---|---|---|
| x86 | 483 | 96.4% | 82% |
| Power | 7 | 1.4% | 5.9% |
| Arm | 6 | 1.2% | 9.8% |
| Others | 5 | 1% | 2.3% |

Table 1.1: Share of CPU architectures of the June 2023 Top 500 machines

relying on this Arm Neoverse roadmap [5] such as the European Processor Initiative (EPI) or NVIDIA Grace, the performance share of Arm in HPC will not be an accomplishment of a single machine.

Hence, optimization and exploration efforts in this environment are essential with the future Arm HPC machines in mind. Furthermore, the position of Arm and its architecture is particular because of the liberty allowed on the architecture design possibilities. Indeed, because Arm is only selling Intellectual Property and not a constructed CPU, several design choices depend on the constructor according to his or future clients' needs.

Hence, Arm HPC machine constructors aim to direct their design choices with the HPC users' workloads in mind. Then, it is vital to understand how a workload interacts with hardware and its characteristics to make the best choices according to its performance needs. The next chapter focuses on how a workload seen as an instruction flow interacts with the hardware architecture and how these interactions impact the intra-node performance of HPC applications.

# Chapter 2

# Hardware architecture impact on performance

This chapter focuses on the hardware features that affect the node performance analyzed in the prediction model developed during this thesis. To understand today's processor performance, we first need to describe the basics of the functioning of a computer. The processor sees an application as a sequences of instructions to execute. And the smallest unit of a processor that can execute such instructions sequentially is the core.

## 2.1 Functioning of a core

Every action of today's computers depends on the sequence of instructions they receive. To process this instruction flow, they rely on the Von Neumann architecture.

### 2.1.1 The Von Neumann architecture

Following this concept proposed by John Von Neumann in 1945, a computer system consists of four components summarized in Figure 2.1:

- The **Arithmetic and Logic Unit (ALU)**: it is responsible for executing instructions and performing computations;
- The **Control Unit**: it manages the instruction flow, decodes them, and ensures they are executed in the correct sequence by the ALU. It forms the Central Processing Unit (CPU) with the ALU;
- The **Memory Unit**: it is where both data and instructions needed by the processor are stored;

- And the **Input/Output (I/O)**: it enables communications with external devices such as peripherals or storage such as network or hard disk.

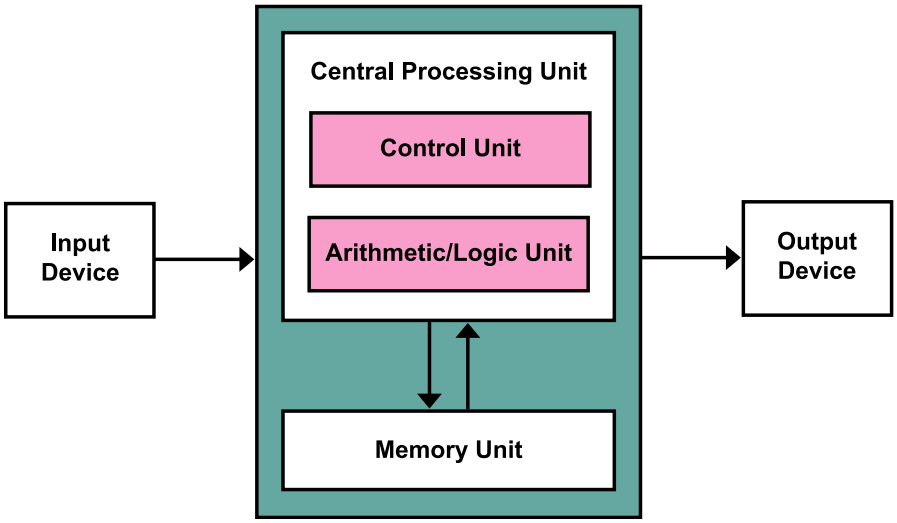


Figure 2.1: The four components of Von Neumann Architecture - Source : [21]

The usage of all these components is vital to take into account for a computer to reach high performance as one of these components lagging would cause a performance bottleneck that would reduce the speed of every other component. During the execution of a HPC application, all these components interact with each other to execute the many instructions per second. And a high instruction processing speed translates to a high performance of applications.

One way to increase this speed is to increase the processor frequency. As presented in Figure 2.2, this is one of the choices made by constructors over time. But, since 2000, CPU frequency has stagnated around 2 GHz because of power consumption and efficiency constraints.

### 2.1.2 The instruction pipeline

However, the processors' instruction processing speed has not stagnated since 2000. Hence, it is the introduction of new techniques that caused this increase. One such technique is the instruction pipeline. An instruction pipeline works similarly to a montage line in a factory. The processing of an instruction is divided into several steps with the processor issuing one step of the pipeline per cycle. When used efficiently, it allows for a faster throughput of instructions than an in-order execution of the instruction flow.

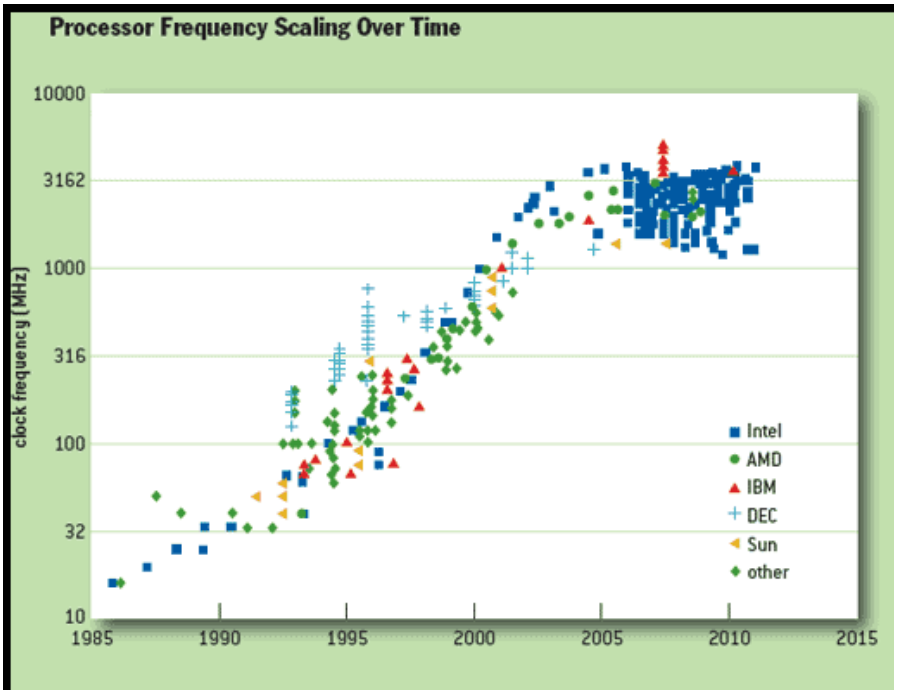


Figure 2.2: CPU frequency scaling over time extracted from Stanford CPU DB
Source : [42]

A simple example is the 4-stage pipeline presented on the left of Figure 2.3. It issues each instruction in four steps: the fetching, the decoding, the execution, and the write-back. It would need $4 \times 4 = 16$ cycles to complete four instructions without pipelining. Whereas, with the correct use of the instruction pipeline, it needs only 9 cycles.

In this example, if each instruction issued is equal to 1 floating-point operation, the performance of the pipelined flow is 0.44 floating-point operations per cycle, whereas it is 0.25 floating-point operations per cycle with an in-order execution. Hence, the use of the pipeline allows to gain performance. But, because of the dependency between instructions, it can hinder the usage of a pipeline with the apparition of a "pipeline bubble". On the right side of Figure 2.3, a bubble appears because the purple instruction execution needs the completion of the green instruction. This wait causes a "stall" in the pipeline leading to an execution of the instruction flow in 10 cycles instead of 9.

Then, the instruction flow has to adapt to these dependencies to use the pipeline to its maximum efficiency. The execution of other independent instruction can cover this stall caused by instruction dependency. Today, processors are out-of-order as they do not necessarily execute instructions in the same order they receive

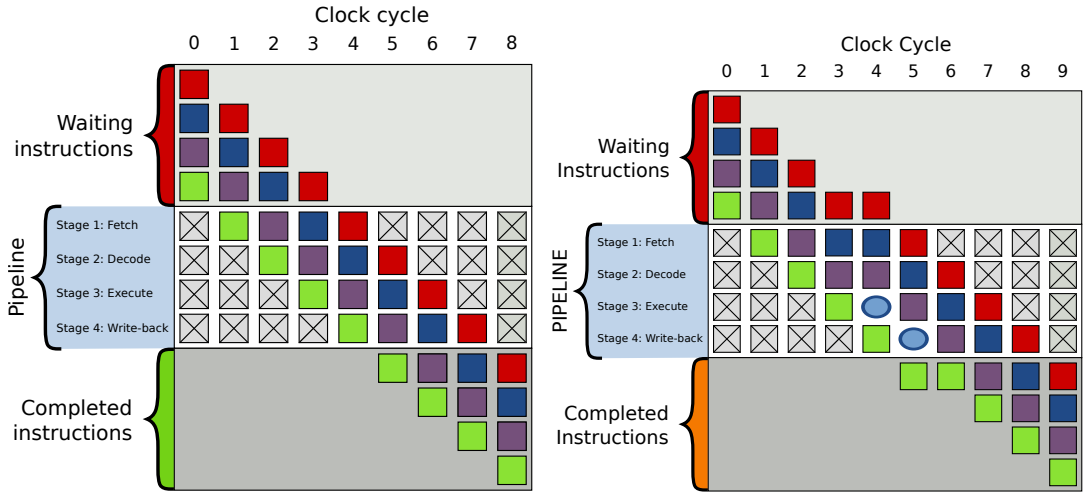


Figure 2.3: Generic 4-stage instruction pipeline without a bubble (left) and with a bubble(right) - Source : [20]

the instructions. They let the instructions wait in a Reordering Buffer (ROB) and issue them in their respective pipeline according to a protocol to resolve their dependencies or independencies. Moreover, with more than one pipeline in today's CPU, they are more proficient in executing many instructions in parallel to cover stalls thanks to this Instruction-Level Parallelism (ILP). The readers can refer to [23] for more information on these protocols and the ILP optimization methods as this is not a focus of this work.

In HPC, the metric for performance application is the FLOPS or floating-point operations per second. It means that only instructions issuing floating-point operations such as *fmadd* or *fsub* in Aarch64 [6] are considered useful. Hence, with this focus on floating-point performance, an efficient usage of the pipelines is translated into more floating-point instructions executed per second. The components of the core that executes these floating-point instructions are the Floating-Point Units or FPUs. Then, a core achieves high performance when its FPUs execute floating-point operations on the most data per second. And Single Instruction Multiple Data (SIMD) and Fused Multiply Add (FMA) instructions have been implemented in today's CPU to increase the computing speed of FPUs.

## 2.2 Impact of SIMD and FMA on performance

Taking the Arm Neoverse V1 architecture [4] as a reference, its FPU can compute one floating-point addition or multiplication in 2 cycles. One core has four FPUs

which means it can issue two floating-point operations per cycle. With a clock frequency of 2.6 GHz implemented in the AWS Graviton 3 CPU, one of its cores can attain $2 \times 2.6 = 5.2$ GFLOPS on scalar floating-point addition and multiplication.

We have seen that increasing the processor frequency is not a reliable way of optimization. Hence, the other possible solution is to increase the number of floating-point operations per cycle. And this is achieved thanks to the usage of **Fused Multiply-Add** (FMA) and **Single Instruction Multiple Data** (SIMD).

### 2.2.1 SIMD and FMA description

A FMA is a floating point instruction (*fmadd* or *fmsub* for Aarch64) that combines one multiplication and one addition. Hence, one single FMA instruction counts as two floating-point operations.

SIMD, in Aarch64, may be summarized as an instruction executed on a vector register (such as *Q0-Q31* registers). A vector register is a fixed-size register that can contain more than one element. Hence, the number of floating-point operations of such an instruction depends on the type of floating-point instruction (FMA or not), the data size, and the hardware vector size. As an example, an addition of 64-bit double-precision floats on a 256-bit vector register counts as $256/64 = 4$ FLOPs. Figure 2.4 presents the count of FLOPs of different scalar and vector instruction.

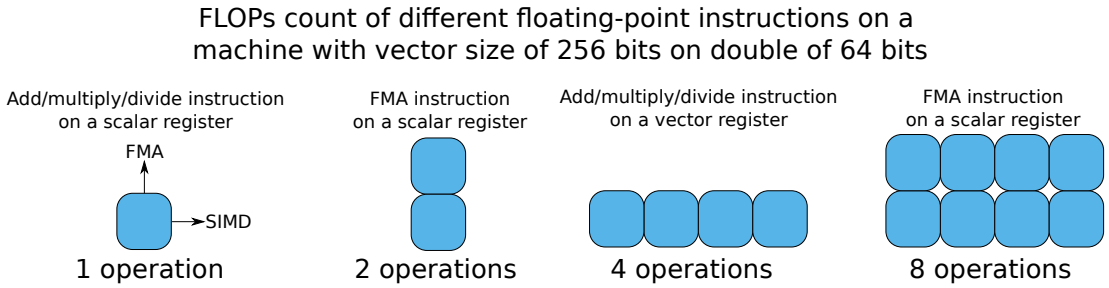


Figure 2.4: Number of FLOPs issued with different type of floating-point instructions

### 2.2.2 Vector ISA in the Arm HPC environment

We focus our study on 2 Arm Instruction Set Architecture (ISA) that implements SIMD: NEON [3] and Scalable Vector Extension (SVE) [89]. The first technology uses 128-bit vector registers. Hence, a single instruction can achieve a maximum number of 4 floating-point operations using double-precision floats (FMA on a 2-element vector). The second technology is a "vector length agnostic" approach

as this ISA allows for many different vector register sizes. SVE instructions can process vector registers whose size varies between 128 and 2048 bits (with a step of 128 bits). This particularity opens up design liberties on the hardware vector size as it is now a choice implemented by the constructor and not dependent on the architecture. Currently, two machines implement this ISA with 256-bit vectors on AWS Graviton 3 or 512-bit vectors on Fujitsu A64FX. Moreover, another characteristic of this ISA is the use of predicates to govern active elements involved in vector operations, acting as a bitmask. It opens the vectorization on loops with complex control flow. With the possibility to vectorize more loops than NEON, SVE can have more impact on performance. As explained before, the performance gain of SIMD can be linked to vector size. Hence, with a vector length agnostic approach such as SVE, the choice of vector size becomes an important question for constructors in the Arm HPC environment.

Consequently, if we take the Arm Neoverse V1 core as a reference, it can execute FMA instructions on two 256-bit vectors in parallel translating to 16 operations on 64-bit double-precision floats per cycle. Then, the use of SIMD and FMA leads to a maximum core performance of $16 \times 2.6 = 41.6$ GFLOPS on vector floating-point addition and multiplication. However, to achieve this high number of floating-point operations per second, CPUs first need to load the data processed by these instructions in these vector registers. Hence, the data movement is critical to achieve a high FLOPS count with FMA and SIMD.

## 2.3 Memory organization

As explained before, the performance of HPC applications is counted in FLOPS and the usage of SIMD and FMA in today's CPU increases their floating-point processing speed. But, as the number of floating-point operations executed on data per second increases, this quantity of data needed per second also increases. During the execution of a program, the main memory or Random Access Memory (RAM) stores these data. Consequently, data needs to be read from memory and loaded in CPU registers before being processed by floating-point instructions. Hence, because the CPU makes computations on more data per second, this data also needs to be loaded in registers faster. As a result, the use of SIMD and FMA often places a heavier burden on the memory bandwidth of the machines.

Because of the Dynamic RAM (DRAM) technology, reading and writing data in main memory is slow, both in latency and bandwidth, compared to the processor speed [37, 79]. This is called the *memory wall* [81]. Therefore, constructors introduced memory optimization techniques to balance out the need for faster main memory bandwidth and lower latency.

The main one is the introduction of cache memory. It consists of small, high-

speed memory closer to the core than the main memory. It acts as a buffer between the CPU and the main memory. When the CPU loads or stores data, it checks the cache memory from the lowest (L1) to the highest level (Last Level Cache, LLC). If it finds the data, it is immediately fetched with a lower latency and faster bandwidth than from the main memory, issuing a cache hit. If the data is not stored in the cache, it copies the data from the main memory in cache memory as a cache line (this cache line can be larger than the data needed), issuing a cache miss. Finally, if a cache level is full, it chooses a line to evict according to an eviction protocol. Most known protocols are Least Recently Used (LRU) or Least Frequently Used (LFU). In the following Figure 2.5, we present the memory hierarchy of a four-stage memory: 3 cache levels and the main memory.

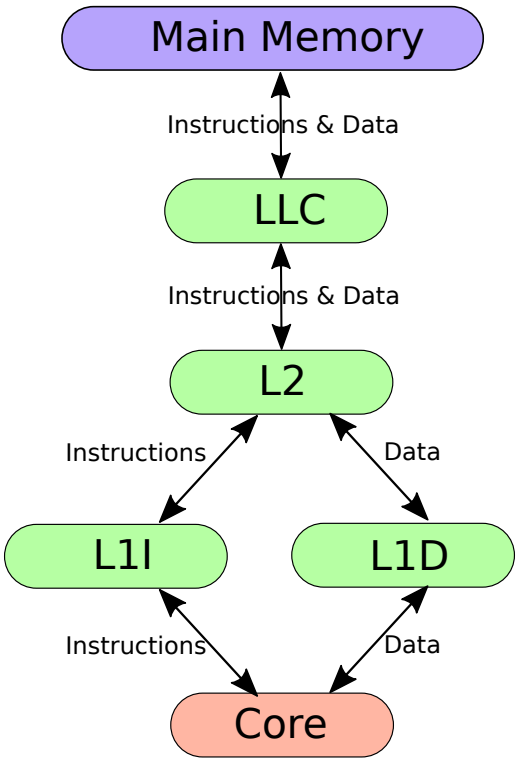


Figure 2.5: Hierarchy of a four stage memory architecture: 3 cache levels and the main memory with 2 L1s, one for data and one for instructions.

Cache memory uses two proprieties, that depend on the program's behavior, of the data processed during a program's execution:

- Its **Spatial Locality**: The accessed data are contiguous in memory;
- And its **Temporal Locality**: The accessed data will be reused later in the execution.

Hence, because of these proprieties, having small but fast data storage leads to much fewer memory accesses in the main memory. Reducing the number of slow memory accesses to the main memory leads to faster bandwidth and lower latency on memory loads and stores to permit faster computation of FPUs. As an example, if we consider the effective bandwidth ($BW_{eff}$) of an application on a one-level cache machine, we obtain:

$$BW_{eff} = h * BW_c + (1 - h) * BW_m \tag{2.1}$$

with $h$ being the hit ratio in cache memory, $BW_c$ its bandwidth and $BW_m$ the main memory bandwidth. If we consider $BW_c$ to be ten times faster than $BW_m$, which is realistic, the application has a 5.5 bandwidth speedup with a cache hit ratio of only 50%.

In today's processor, because of the memory wall, it is crucial to make efficient use of the cache memory for performance. But, making efficient use of cache is not straightforward [50] as it is transparent even for the processor. This problem of needing high cache efficiency for the performance of today's applications is even further exacerbated by the use of multiple cores in parallel.

## 2.4 Impact of memory on multi-core performance

We have seen that SIMD, FMA, and multiple FPUs on a core lead to a computing speed gain. Furthermore, with the cores becoming smaller and smaller in size, another way to multiply the computing power of a processor is to increase its cores number. As today's processors are not single-core, we call them multi-core processors or even manycore as the number of cores per processor has been rapidly increasing over the recent years. Figure 2.6 presents the evolution of the number of cores per socket, and its core and sockets computing speed of the ranked #1 machine of the Top500 over time.

In an ideal scenario of linear scaling, the performance of applications is multiplied by the number of cores used during their execution. However, multiple factors can hinder the scalability of HPC applications. One of these main factors lies in the memory constraints imposed by a computing node. The first constraint is the main memory and shared cache bandwidth limitation.

This bandwidth limitation is caused by the non-linear scaling of the bandwidth when increasing the number of cores due to a finite number of memory channels in memory controllers. As an example, in Figure 2.7, we can see that the maximum of the total main memory bandwidth is attained at 14 cores on a 64 cores AWS Graviton 2 node. Then, it reaches a plateau until 64 cores. Similarly, the bandwidth available per core ($\frac{\text{Total bandwidth}}{\text{Number of cores}}$) rapidly decreases.

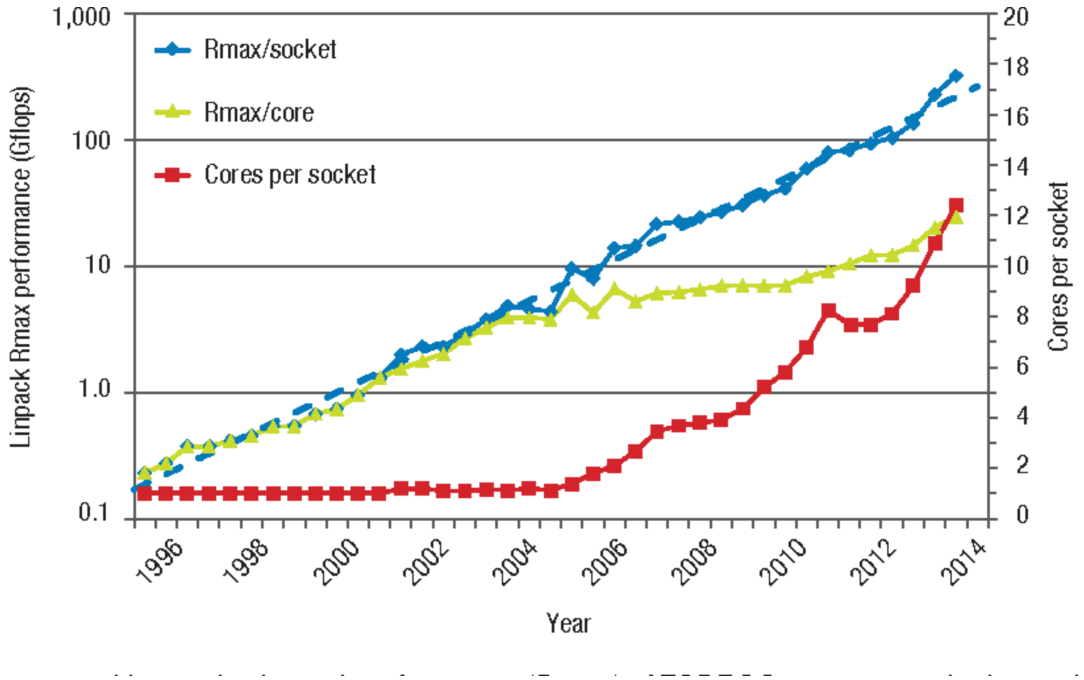


Figure 2.6: Maximal computing speed of a core (Rmax/core) and a socket (Rmax/-socket) and number of cores per socket of Top500 systems

As a result, when the workload's performance is limited by the data movement, the application's scalability on a full node can be hindered. Moreover, applications' performance relies more on main memory bandwidth as the cache levels are less effective in multicore execution due to phenomena like false sharing or cache contention. Finally, maintaining cache coherency in most architectures adds to the burden on the cache and main memory bandwidth of HPC applications running on multiple cores because most of today's CPUs ensure that every cached copy of data has the same value. And ensuring this coherency generates more memory traffic.

In this work, we will study the impact of some mechanisms implemented in multicore CPUs to alleviate this bandwidth burden and help the scalability of applications. These mechanisms are Non-Uniform Memory Access (NUMA) techonology and shared cache.

### 2.4.1 NUMA effect on performance

The bandwidth per core is limited when increasing the number of cores, and faster main memory bandwidth is expensive. Hence, one solution is to increase the number of memory channels by adding more memory units. However, not every

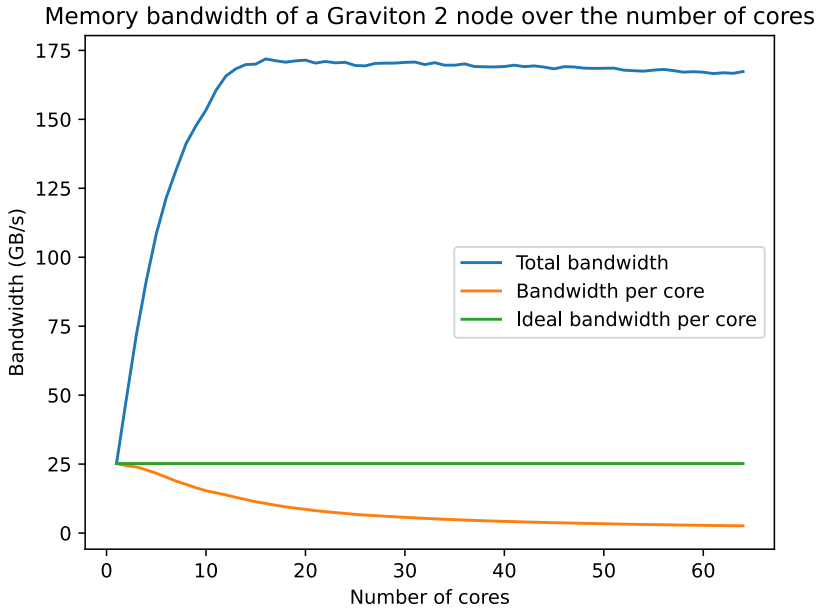


Figure 2.7: Main memory bandwidth obtained with STREAM iterating over the number of cores of an AWS Graviton 2 node.

core has direct access to these added memory banks. Cores having direct access to the same memory bank are in the same **NUMA node**. Hence, when a core accesses data in the main memory in another memory bank, the access will be longer than from its memory bank. This is a NUMA effect.

NUMA effects can impact the latency and bandwidth of memory accesses depending on the memory bank the data is stored in. This technology allows for higher bandwidth at a node level (because of the higher memory channel count) translated to higher memory bandwidth per core. However, the developer needs to be careful of where the data he's working on is stored as accesses between memory banks are slow. Hence, a bad thread placement policy can cause a lower memory bandwidth because of the many accesses between NUMA nodes. Hence, the efficient use of NUMA technology and reduction of NUMA effects rely on the thread placement policy according to the application's behavior.

### 2.4.2 Shared cache impact on performance

Shared caches are cache levels shared between multiple cores. This allows for faster data movement between cores because they do not have to go through the main memory to exchange data. Hence, some cores exchange data together faster

than others leading to a reduction of the communication time. This can lead to performance gain in communication-bound application as it can be seen in the Binding 2 on Figure 2.8. However, if many cores access differents data in the shared cache level as in a memory bandwidth-bound application, it can cause memory contention as the cache size is limited. This can be seen on the Binding 2 on Figure 2.9. Then, the developers need to be aware of such cache sharing in its thread placement policy to allow for better cache usage and faster communication.

### 2.4.3 Topology impact on bandwidth performance

Following both of these effects, it is clear that some cores work together better than others. As an example, two cores sharing a cache level will communicate faster than two cores sharing only a NUMA node. They also communicate faster than two cores on two different NUMA nodes. However, load sharing and communication depend on the application. In this case, an application with lots of exchange between cores needs to be bound accordingly to make the best usage of the machine topology (Binding 2 on Figure 2.8).

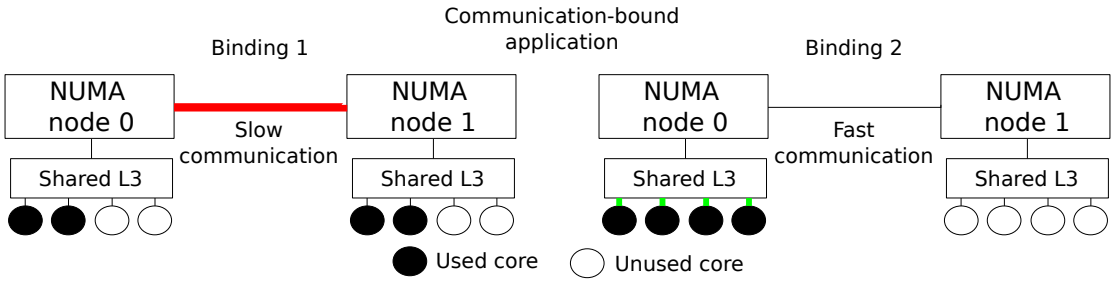


Figure 2.8: Impact of two different placement policies on the performance of an application limited by communication running on four cores. Communications between cores in Binding 1 are limited by the Memory Bus bandwidth between NUMA nodes and can hinder the application's performance.

On the other hand, an application without communication between its running cores but a lot of data movement between main memory and core registers, such as STREAM, will make better usage of all the memory channels if the binded cores are distributed on different NUMA nodes (Binding 1 on Figure 2.9).

Hence, making the best choice on core binding of a parallel application requires a deep knowledge of the topology of the underlying machine and the application behavior for the application to scale efficiently.

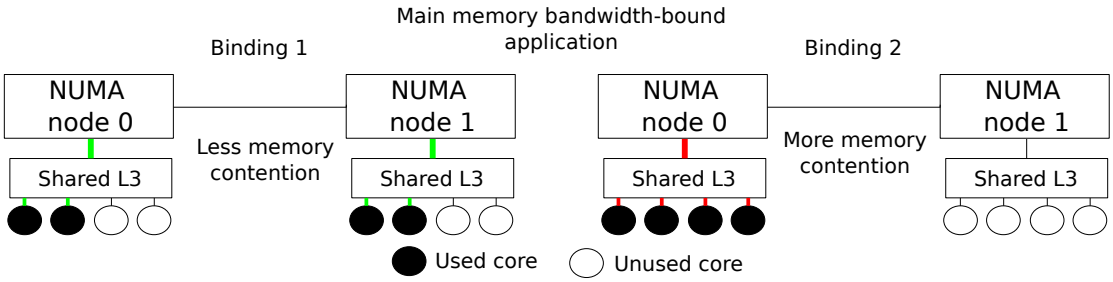


Figure 2.9: Impact of two different placement policies on the performance of an application limited by main memory bandwidth running on four cores. Bandwidth in Binding 2 is limited by the contention on the NUMA node 0 memory channels and can hinder the application's performance.

## 2.5 Conclusion

To conclude this chapter, we have seen that today's CPUs use lots of mechanisms to increase their computing power with SIMD, FMA, and more cores. However, these mechanisms place a burden on memory bandwidth to move the data fast enough according to the processing speed. Furthermore, this burden is exacerbated by the multiplication of the number of cores per processor. Hence, cache memory and NUMA were implemented to help tackle the memory wall and allow HPC applications to have better scalability with the number of cores. However, making efficient use of them requires the supercomputer users to be careful about their applications' memory and communication behavior. It requires them to have deep knowledge of the underlying architecture of the hardware and the application's behavior but also to adapt their application to the machine's evolution. However, these users often do not have a background in computer science, and keeping the full HPC applications up to date with the machines' evolution is long and tedious. Simplifying these optimization efforts and the portability of applications on machines is the role of the software stack.

# Chapter 3

# Software stack and programming environment

In this work, we define the software stack as the software abstraction layer used to help application developers to attain the highest performance on a machine in a minimal optimization time.

We will focus our study on the choices made by different layers of software stack impacting the applications' performance at execution. First, we describe the possible compilers optimizations on the static binary before execution. Then, during executions, the choices made by runtimes, programming models and dynamic libraries also leads to performance differences. Finally, one non-neglectable aspect of the software stack is the ability to help developers understanding the performacne of their application thanks to profinling tools . We will also take a look at on one model used in performance analysis: the Roofline Model.

## 3.1 Compiler impact on performance

As seen in the previous chapter, the cores of a CPU see the executing application as sequences of instructions. Hardware optimizations such as cache usage or use of SIMD and FMA depend on this sequence of instructions. In this case, directly optimizing the sequence of instructions expressed by the executed binary is a way to obtain performance. During compilation, the compiler can directly optimize the instruction flow. Hence, to understand the choices made by a compiler that lead to differences in performance, we first must understand the role of a compiler.

The compiler's role is to transform "human-readable" programming language into "machine-readable" language contained in an executable binary. Modern compilers split this translation into three stages: front end, middle end, and back end.

Figure 3.1 presents these three stages and the representation of the code given to each stage:

- **Front End:** This stage is responsible for the initial processing of the source code. It performs lexical analysis to convert the code into tokens, syntax analysis to create a parse tree, and semantic analysis to check for correctness and generate an Intermediate Representation (IR).
- **Middle End:** In this stage, the compiler applies various optimization phases on the generated IR. They aim to improve the efficiency and performance of the resulting machine code without any knowledge of the target architecture. This results in an Optimized Intermediate Representation (OIR).
- **Back End:** The final stage of the compiler is the back end, which takes the OIR and translates it into the target machine code or assembly language. This stage involves mapping the IR to specific processor instructions, performing architecture-dependent optimizations, and generating the final executable or object file.

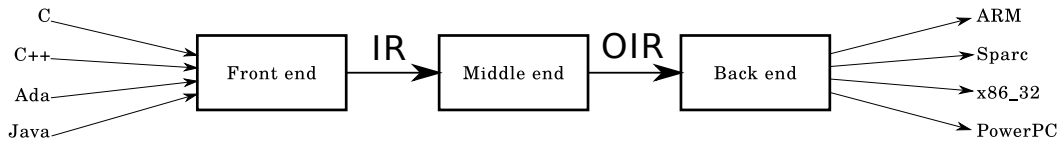


Figure 3.1: The three stages of a compiler from source code (in different languages) to Machine code (for different architectures) through Intermediate Representation (IR) and Optimized IR (OIR)

Whether it's a compiler based on the LLVM (Low-Level Virtual Machine) backend or GCC (Gnu Compiler Collection), they all follow this organization. However, the IR and OIR of the code will be different. During every stage, the compiler needs to ensure the correctness of the code. While keeping this correctness, it can make optimization transformations depending on its understanding of the application and machine behavior. In practice, these optimizations are transparent to the scientific programmers who typically only choose the optimization's aggressiveness level, or which transformations to apply, through the *-O* flag [92].

One example of such optimizations is the auto-vectorization of loops. If the developer does not order the compiler to vectorize a loop, he will vectorize it according to its expected impact on performance and the optimization level. As seen before, vectorization of a loop can lead to crucial performance gain if the

bandwidth is fast enough to sustain it. Hence, whether a loop is automatically vectorized or not ultimately depends on the heuristics of the compiler if he considers it a performance gain. And, every compiler has its own heuristics, whether it is Gnu Compiler (GCC), LLVM-based (clang, armclang, ...), or the Fujitsu Compiler (FCC). Consequently, on the same loop written in C code, one compiler can choose to vectorize and the others not to according to its heuristics and knowledge of the underlying architecture.

Auto-vectorization of loops is an example of one of the many choices the compiler makes while translating source code. Any of these choices impact the performance and the machine's usage efficiency of the compiled binary. Hence, if the programmer has enough knowledge of the underlying architecture, the best practice is to direct the compiler's optimizations thanks to intrinsics.

However, the impact of the software stack on performance does not only rely on the compilers, as the behavior of an application binary at execution is dynamic. Today's HPC applications often use runtimes through dynamic libraries for communication, offloading, or even optimization of linear algebra operations, ...

## 3.2 Programming model and runtime optimization

At execution, the instruction flow of an HPC application does not depend only on the static instruction flow generated by the compiler. It also depends on the software stack at execution through calls to libraries, Operating System (OS), ... In this section, we will focus on the impact of parallelization runtimes on intra-node performance through two programming models: shared and distributed memory. We will also present different linear algebra libraries and their impact on performance.

Both of these programming models differ in how they handle parallelism. Shared memory programming, such as Open Multi-Processing (OpenMP), allows multiple threads to share a common memory space, making it efficient for communication and synchronization within a single computing node. Communications between threads are implicit by directly accessing data of another thread.

On the other hand, distributed memory models like Message Passing Interface (MPI) involve multiple MPI processes, each with its separate memory. The data exchanges and synchronization between these processes are made through explicit communications. Another advantage is that processes can be binded on more than one core instead of OpenMP threads that only runs on a single logical or physical core.

Both of these models have pros and cons, but they are usually combined with OpenMP for intra-node parallelization and MPI for inter-node parallelization. Fig-

ure 3.2 sums up the functioning of MPI and OpenMP and Figure 3.3 presents the combination of both.

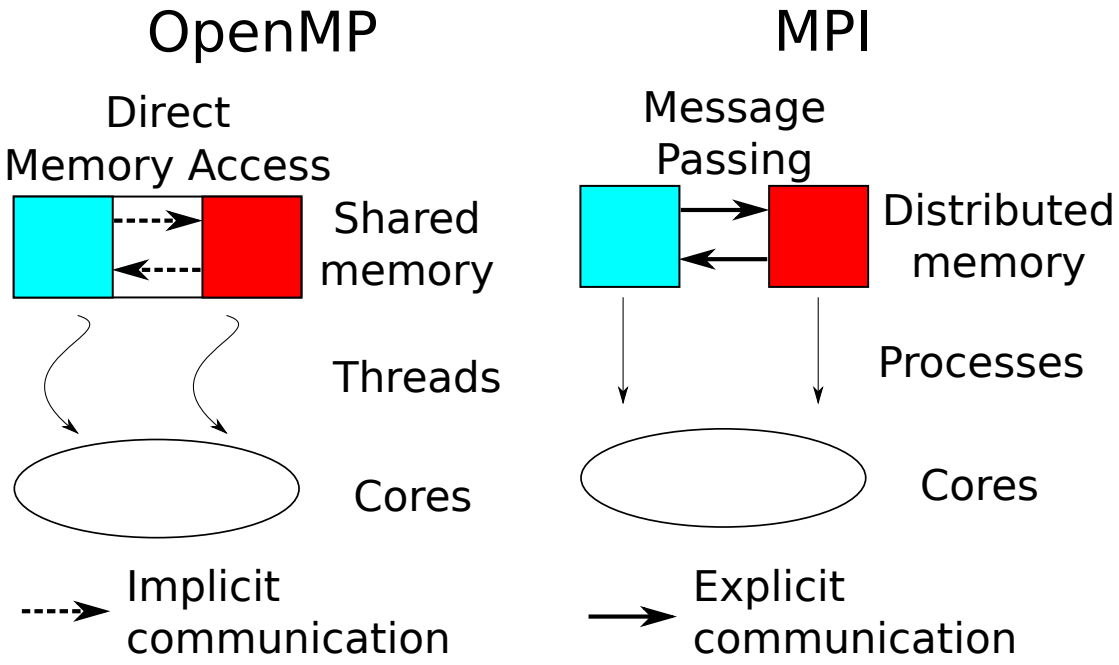


Figure 3.2: Functioning of a Shared Memory programming model (OpenMP) and a Distributed Memory programming model (MPI).

Nevertheless, as we have seen before, with the introduction of NUMA nodes, it can also be interesting to map the MPI processes to each NUMA region to explicitly express communications between cores of different NUMA regions.

Hence, the communication behavior of the application depends on the programming model used but also on the way it is used. Moreover, the positioning of processes and threads also relies on the choices made by these parallelization models and by the topology informations gathered by an underlying software such as *hwloc* [30]. As explained before, the positioning and the use of the topology directly impact the effective bandwidth and the applications' performance [31, 54]. Their impact on the use of hardware topology is one of the many sources of performance changes brought by the software stack at execution.

Another example is the use of optimized linear algebraic libraries. As many HPC applications rely on linear algebraic libraries for computation through the Basic Linear Algebra Subprogram (BLAS) interface implemented in libraries such as OpenBLAS [96] or Arm Performance Libraries (ArmPL) [25]. Each of these libraries presents the same interface to the applications but with a different implementation. These different implementations lead to changes in the instruction

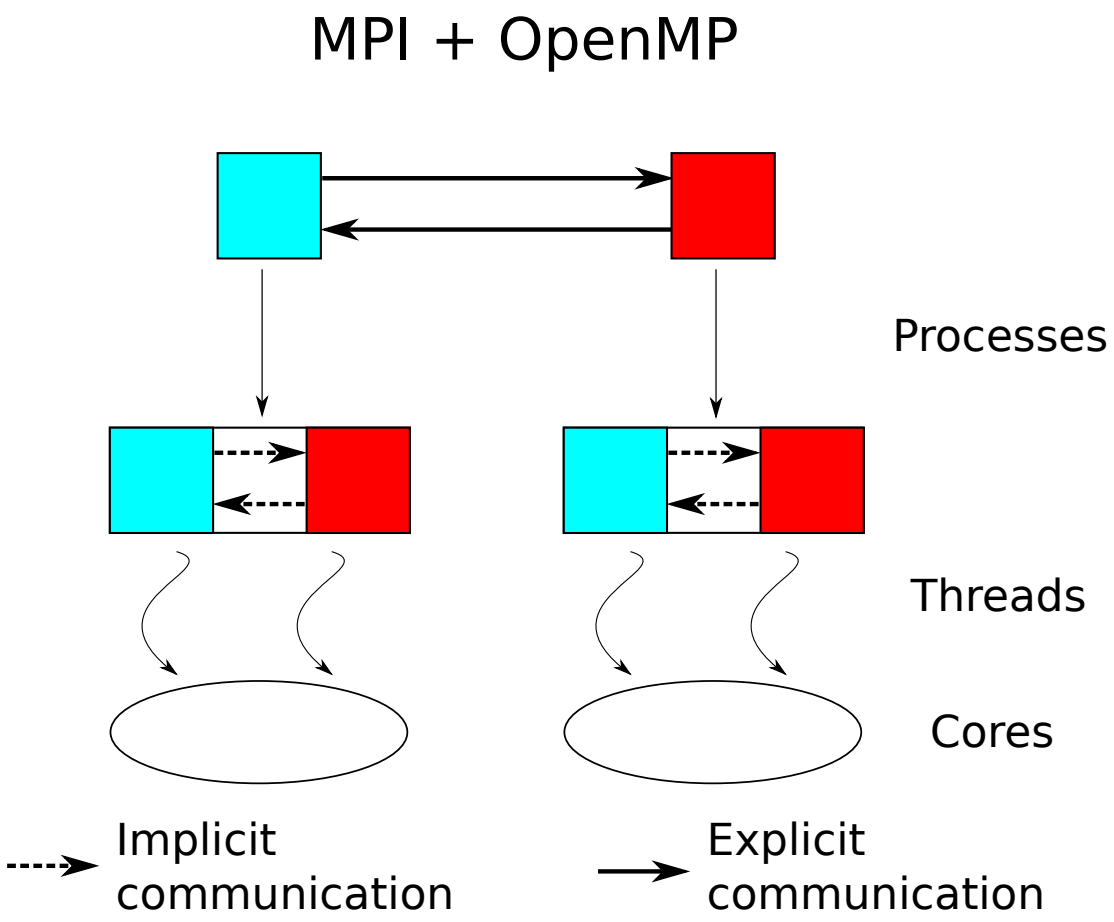


Figure 3.3: Combination of MPI and OpenMP to combine the strengths and cover the weaknesses of each programming models.

flow and, consequently, in observed performance [22].

We have seen that numerous factors impact the performance of HPC applications. Whether it depends on the underlying architecture or the software stack, the developer needs to have a deep understanding of both of these aspects to identify performance bottlenecks in the HPC environment and optimize applications. Hence, the use of a performance analysis tool is mandatory to identify performance bottlenecks, solve them, and, thus, reduce the optimization time of HPC applications in today's HPC environment.

## 3.3 Application performance analysis

Understanding the performance bottlenecks of applications is mandatory to help developers in optimizing HPC applications. However, as we have seen before, numerous factors can hinder the performance of an application in the HPC environment. Hence, they rely on profiling tools to help them identify performance bottlenecks. However, developing such software also necessitates a deep understanding of the environment. Hence, no generic tool is able to identify all the possible performance bottlenecks of a specific application running on a particular hardware with a certain software.

### 3.3.1 Profiling tools

A wide variety of performance analysis tools, such as Intel VTune [8], analyze the code execution, revealing time-consuming functions and memory usage patterns to give source code optimization advice. There are also trace-based tools like NVIDIA Nsight Systems [13] or Tau [88] to capture the code execution flow, aiding in understanding thread interactions, synchronization, and GPU utilization. Performance analysis in HPC also involves tools like Linux Perf [44] and PAPI [41], which offer low-level hardware performance counters to understand the hardware behavior. The uniqueness of each tool lies in its focus, whether on application code details, software insights, or hardware-specific metrics, enabling developers to pinpoint the performance bottlenecks of their applications.

In this ecosystem, a wide variety of tools implement the Roofline Model as a performance analysis tool and visualization, thanks to its simplicity and generality.

### 3.3.2 The Roofline model as an analysis tool

We have seen that, to reach high performance and a high FLOPS count, the bandwidth needs to be fast enough to sustain the cores of a processor. This idea is the main philosophy behind the Roofline Model initially described by S. Williams et al. [97]. This model characterizes the maximum limitations imposed by the hardware, whether it is the bandwidth or the peak computing speed of the machine. The limits imposed by the hardware is the "roofline" shape on Figure 3.4. This is described by the following equation (3.1):

$$\text{roofline(OI)} = min(\text{Peak Bandwidth} \times \text{OI}, \text{Peak computing speed}) \tag{3.1}$$

Then, it characterizes the application's behavior according to its Operational Intensity in FLOP/Byte (OI), or the mean number of floating point operations

per byte moved. This OI is positioned on the x-axis, allowing us to see if the processor's bandwidth (memory-bound) or computing speed (compute-bound) limits the application's performance.

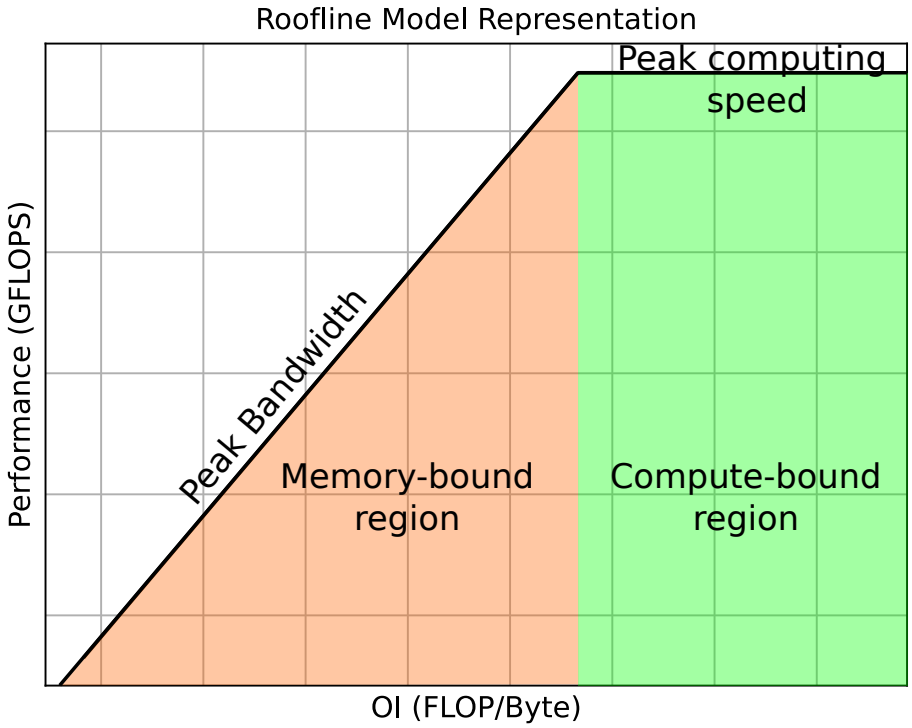


Figure 3.4: Roofline representation with the performance (in GFLOPS) on the y-axis and the Operational Intensity (OI) on the x-axis. The black roofline represents the maximum performance attainable at a certain OI.

Finally, once the application performance is measured and placed on the roofline chart, it allows for visual analysis of the possible performance limitations [59]. This visual analysis allows to identify different optimizations patterns according to the application's behavior (its OI) and the hardware limitations (the Roofline). On the one hand, if the application is in the memory-bound zone, the focus should be on memory patterns optimizations to increase the OI (through better cache efficiency or data structure changes) and reach the compute-bound zone or the peak performance allowed by the bandwidth. On the other hand, if the OI is in the compute-bound zone and not reaching the peak, the focus should be on FMA and SIMD usage but also on the use of the Instruction-Level Parallelism.

The first iteration of the Roofline Model, developed by Williams et al. [97], only used the main memory bandwidth in the roofline and only counted the data exchanged (loaded or stored) in this memory level in the OI. Today, there are numerous iterations of this roofline model with tweaks on the OI and rooflines to describe different possible performance bottlenecks. One such example is the Cache-Aware Roofline Model (CARM) by Ilic et al. [61] that adds rooflines for every memory level to be able to add every memory level in the analysis of possible performance bottlenecks. Another well-used iteration of the roofline model is the Hierarchical roofline model used in Intel Advisor [63] that defines one OI per memory level but can also characterize the FMA and vector usage in the peak computing speed of the hardware. Finally, the Roofline model analysis has also been used to characterize accelerators performance of GPUs [48] or FPGAs [40].

To conclude, the roofline model and its variants are well-used in the performance analysis domain thanks to its simplicity and versatility. However, this simplicity also comes with a cost, as no generic version of the roofline model can pinpoint every possible performance bottleneck of today's HPC applications.

## 3.4 Conclusion

To conclude, a software abstraction layer is mandatory because of the complexity of the interactions between the application workflow and the underlying machine, as HPC users cannot directly write applications in machine language. Whether through compiler optimizations of the binaries before execution, or by adapting the application to the underlying hardware through optimizations at executions and the use of linear algebra libraries, this software stack layer often needs to make many choices transparent to the user that impact application performance and machine usage. And the best practice for developers is to direct these choices by understanding and identifying the performance bottlenecks of application running on a hardware. However, this requires a deep understanding of the underlying architecture and its interaction with the particular application flow. The software stack also comes with performance analysis tools to help applications' developers identify performance bottlenecks brought by the interaction between all the components on current machines.

# The problem

In this part, we have seen that HPC applications rely on complicated workflows. When running on supercomputers, these workflows interact with complex machine architectures through a varied software stack that needs to make choices. From all these interactions, we only see one metric: the performance of this application or the number of floating-point operations per second. However, identifying the performance bottlenecks that hinder the machine's efficiency is long and tedious work that needs a deep understanding of every interaction in the execution of an HPC application on an HPC machine. Even if profiling tools and models can alleviate these efforts, it is a heavy task. Additionally, there is a significant gap in the lifetime of applications and machines, leading to running one version of an application on several generations of machines with different hardware and software stacks.

In this case, one solution is to design machines in a codesign environment where each actor involved in the conception and usage of future HPC machines shares its needs and choices in their respective domain. Such a conception environment would lead to higher performance for the user's applications and reduce the optimization time and costs. Furthermore, the Arm HPC environment is attractive for such codesign initiatives, opening up many design liberties and possibilities. Even if each actor involved in the conception and usage of HPC machines is not knowledgeable in the other's field, they can discuss a standard metric and goal of optimization: the performance of their applications.

In this context, using a performance prediction model that can give feedback on the impact of the design choices of each actor is a great way to open up the discussion in a codesign environment. Hence, the problem that we will try to solve in this manuscript is the following:

**How can we predict the impact on HPC application performance made by the choices of each actor involved in the design and usage of future Arm HPC machines ?**

As performance prediction is not a recent problem in CPU design and HPC, many approaches have been designed over time. The next Chapter 4 presents an overview of performance prediction methodologies, emphasizing their pros and

cons. It also gives insight into what questions could direct the choice of an approach. A need to clarify the codesign environment emerges from this overview, leading to the definition of our codesign environment and the solution of relying on a mechanistic model to predict performance.

Chapter 5 aims to present this first contribution: a performance projection workflow between a source and target application/software stack/hardware triplet adapted to our needs, arising from the definition of a codesign environment.

Then, we implement this workflow in Chapter 6 using an adaptation of the Roofline Model for single-core projection in the Arm HPC environment. This implementation aims to explore the design possibilities of various hardware parameters on a panel of HPC proxy applications with a pool of Arm core architectures.

Finally, our final contribution is the natural extension of the single-projection approach to a multicore environment on a compute node. This extension relies on a finer bandwidth characterization and a separate analysis and projection of each thread. The experiments in this Chapter 7 aim to explore optimization patterns around application and software stacks targeting a node architecture of interest.

# Chapter 4

# State of the art

Performance prediction approaches have been used since the early beginning of HPC to help optimize applications and direct the design of future machines. Ideally, such a prediction workflow must be generic, precise, and fast. However, as interactions between machines and applications become increasingly complex, they constantly evolve. Furthermore, nowadays, there is no fast, generic, and accurate approach.

Hence, these approaches compromise one of these aspects according to their usage. We have chosen to split these models into three groups according to the aspect they want to make a compromise on:

- **Cycle-accurate simulators**: They simulate a CPU's executing an application cycle-by-cycle. It is the most accurate generic approach, but it is also the slowest to perform. Hardware designers mostly use it as they require extreme accuracy to fine-tune the design of cores and processors.
- **Application-dependent models**: These models focus on specific applications or benchmark suites, leveraging the understanding of the application's characteristics and resource usage patterns. Predictions are based on observed behaviors and performance metrics, and are fast and accurate at the cost of the genericity of the approach.
- **Analytical models**: It is the most used approach because of its versatility and the speed of its generic prediction. The accuracy and speed of execution of these approaches depend on the time and implementation used to compute the metrics used by mathematical equations to predict performance.

Figure 4.1 presents an overview of these three approaches with their respective advantages and limitations. They are presented in more detail later in this chapter, discussing how these advantages and limitations directed our choice to rely on an analytical model.

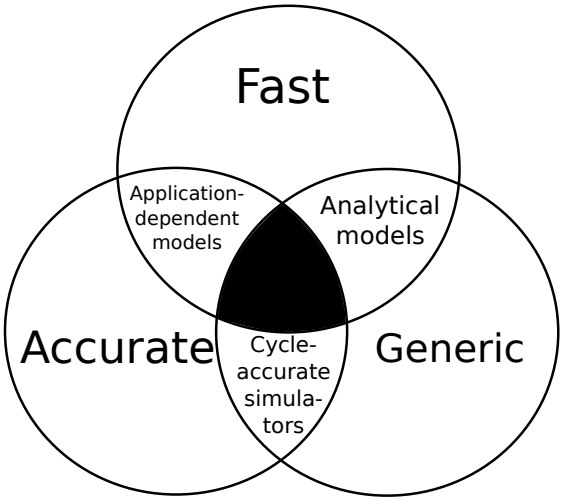


Figure 4.1: Overview of the three different types of approach with the aspect they choose to focus on

## 4.1 Cycle-by-cycle simulators

The use of software to replicate the functioning of a processor executing an instruction flow is called simulation. It allows researchers, developers, and HPC constructors to gain valuable insights into their applications' performance on CPUs without building a prototype. These workflows allow for accurate instruction flow prediction at the cost of an important overhead computing time.

One example in the Arm HPC environment is the A64FX simulator developed by RIKEN and Fujitsu [72]. This simulator is based on the gem5 software [27].

### 4.1.1 The gem5 simulation tool

It has become a staple in the CPU simulation field with many recent simulators based on the gem5 software, such as the RIKEN A64FX simulator or Elastic-SimMATE [83]. Hence, we present some cycle-accurate simulators based on this open-source software. However, other similar approaches not relying on gem5, such as Sniper [36], are also used by Heirman et al. for design space exploration [57].

Like other simulators, gem5 functions by emulating the behavior of a computer system at various levels of abstraction, from individual hardware components like processors and memory to the overall system organization. It is used to evaluate and analyze the performance, power consumption, and other characteristics of new hardware designs or software optimizations without the need for physical hardware. Its modularity allows users to configure and customize simulated systems, making

it a great tool for exploring the design possibilities of future machines. Some of its possibilities are:

- Many different CPU models: in-order, out-of-order, ...
- Many possible systems: Arm, x86, RISC-V, SPARC, ...
- A flexible memory hierarchy through event-driven or trace-driven simulation that allows for multicore simulation

Hence, with the possibility to tweak parameters on all these hardware features, gem5 simulation opens up numerous design exploration possibilities. However, such a precise and flexible prediction comes with a cost: a high execution time overhead.

### 4.1.2 Overview of gem5-based simulators

With these design possibilities, many simulators use gems5 as a back-end, even in the Arm HPC environment. All these simulators share a common point: the accuracy of their predictions with an important overhead.

An example of such a simulator is the one designed by the RIKEN to simulate a Controller Memory Group (CMG) or a group of 12 cores of a Fujitsu A64FX processor [72]. Figure 4.2 presents the architecture of an A64FX processor obtained with *lstopo*. One CMG is a group of 12 cores linked by an L2 cache and a NUMANode.

With their modification of gem5, they achieve around 80% accuracy on single-core test kernels before full processor availability. They were also able to model the STREAM Triad Memory and L2 throughput on up to 12 threads with differences to the test chip due to the lack of implementation of some functions such as prefetching. But, their simulator has an execution time that is 10000 times slower than on the actual machine [72]. In this case, simulating a single 300-second Quicksilver run on a CMG would take around 35 days, which is not exploitable for fast design-space exploration.

There are ways of reducing the execution time of such simulators by using trace-driven simulation in which a first execution collects traces replayed through gem5 with different architectures.

This is the approach chosen by Nocua et al. in ElasticSimMATE [83] to fasten the simulation time for multicore systems with trace-driven simulation. Thanks to the combination of Elastic Traces [65] and SimMATE [34], they speed up the simulation by at least a factor of 3 compared to full-system simulation of up to 128 cores.

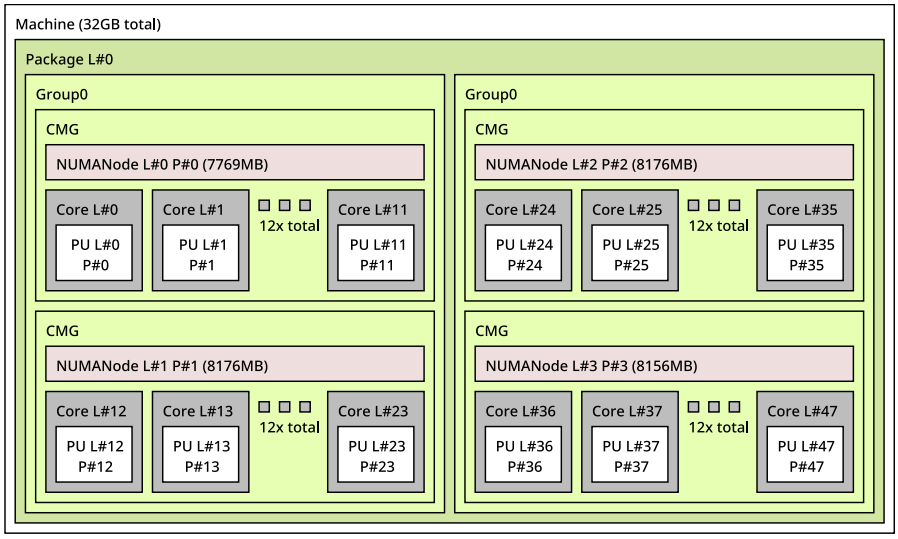


Figure 4.2: Architecture of an A64FX processor obtained with *lstopo*. There are 4 CMG per processor, with 12 cores and one NUMA node each.

### 4.1.3 Discussion

Using cycle-accurate simulation would be the best choice to make design choices based on the most accurate predictions. Nevertheless, we want to make these design choices based on the behavior of HPC applications represented by proxy apps. And, the overhead of these approaches limits the possibility of studying many parameters by running many iterations of the simulations on complete proxy applications. Even with execution time optimization with approaches such as trace-driven simulation, the overhead is still important. Hence, relying only on a simulator to fully predict the behavior of proxy applications on a full machine node would limit the possibility of exploring the design-space.

## 4.2 Application-dependent models

We can group HPC applications into families with similar behaviors. Hence, with a deep knowledge of the application behavior, it is possible to use these similarities to predict their performance and scaling through various architectures. With such a modelization, it is possible to have a fast performance prediction approach tailored to rapidly explore a vast hardware and software design-space with the target application.

### 4.2.1 Hydrodynamics application models

Computer simulations of various science and engineering problems require modeling hydrodynamics. Many laboratories rely on these types of applications, such as the Lawrence Livermore National Laboratories (LLNL) with Ale3D [82] and Hydra [75], or the Los Alamos National Laboratory (LANL) with SAGE [70]. Hence, making design choices for HPC machines according to the needs of this family of applications is of interest to HPC users.

It is the objective of Davis et al. in their modelization of an Atomic Weapons Establishment (AWE) hydrodynamic benchmark scaling behavior on large clusters [43]. This benchmark, named Hydra, is a different code from LLNL's Hydra but model similar hydrodynamics phenomena. They use their knowledge of the application to split each iteration of Hydra into 3 phases: local computation, near-neighbor communication, and collective communication. Then, they use analytical modelization of the first two steps and a machine-specific modelization of collective communications through micro-benchmarks to compute the wall execution time of Hydra with any input size.

Thanks to this approach, they predict the weak scaling of the application with different mesh sizes up to 2048 cores at an 85% accuracy. They are also able to project this prediction up to 8192 cores. Finally, they project the execution time with increased core density (more cores per node) with the source machine core architecture.

These results show what a Hydra user can expect from the evolution of performance on more nodes and cores without running the application on a high node count. It also assesses component costs' impact on performance, including computation, point-to-point communications, and collectives.

### 4.2.2 Discussion

Using application-dependent models would lead to precise prediction and allow the possibility of rapidly exploring the design-space around the modelized applications. However, there are various families of applications in our pool of applications of interest presented in Section 1.1 that we include in our studies. For example, LULESH and Quicksilver both represent very different behavior of applications, with one representing hydrodynamics-solving applications and the other modelizing particle transport with a Monte-Carlo simulation. In this case, we need to develop and validate a model for each family of applications. Furthermore, it would limit the possibility of exploring parameters around new families of applications because it would require a new model. Hence, the genericity of our model is a crucial characteristics of the model we need to enable codesign around the different HPC applications of interest modelized by the proxy apps presented in

Section 1.1.

## 4.3 Analytical models

In this manuscript, we call analytical models the performance prediction approaches that use generic equations and metrics in order to get a prediction of the target application performance on a machine. We split these analytical models into statistical, learning-based, and mechanistic approaches.

- **Statistical approaches**: They regroup all models using observations runs on a machine to feed a statistical model used to predict performance;
- **Learning-based methods**: All the methods using "black-box" learning-based model such as machine learning or neural network on a wide dataset of benchmarks and machines;
- **Mechanistic models**: They regroup all models using the characterization of the workload, with or without profiling with metrics, to assess performance on a machine.

### 4.3.1 Statistical approaches

We consider a prediction model using a statistical approach if they automatically derive an equation of the application's performance through a small number of runs. These approaches use regressions to define algebraic expressions with constants, variables, and operators. More information on the mathematics used in these models can be found in [56].

Historically, these models were used on single-core processors to evaluate the effects of pipeline and ILP through linear regression [84, 51]. Even if this approach has been adapted to superscalar processors by Karkhanis et al. [69] and single-core performance is still important even in parallel applications, nowadays processors have multiple cores.

Furthermore, as the behavior of multicore processors is nonlinear (because of cache effects, bandwidth limitations, ...), using a linear regression approach is not doable in this case. Moreover, solving these nonlinear systems is a mathematically complex task. Some of these approaches rely on iterative methods such as stepwise regressions to [68], whereas others rely on polynomial approximations [76].

The common point in all of these approaches is that they use varying observations as reference points to derive an equation. This function depends on a fixed number of design parameters to describe the performance. Furthermore, once this

equation is defined and accurate enough for an application, the explorations of the design-space through parameters impact on performance is fast.

However, they need to rely on simulation or prototypes to obtain these reference points. Moreover, the number of parameters that impact an application's performance increases as the machine becomes more and more complex, leaving some of these models obsolete for computing applications' performance. Furthermore, developing a new model that considers these parameters will use even more complex equations.

Following this trend, the use of methods relying on learning has increased to speed up and allow the exploration of a more complex design space.

### 4.3.2 Learning-based methods

As machines become more and more complex, performance prediction using learning-based approaches has taken over traditional statistical generic approaches to deal with the non-linear behavior of applications' performance.

We call learning-based approaches the "black-box" approaches consisting of training a model on a vast dataset. There are many approaches in learning-based methods as some rely on Neural Network [64], others on Random Forest [91] or supervised learning [100, 99].

As with every learning-based method, they consist of two phases: a training and a test phase. The validity and accuracy of these approaches are highly dependent on the choice of the training set. A good training set should follow the following properties:

- It should cover the whole application space of interest;
- Every workload should be representative of the applications of interest;
- It needs to have a large number of instances to avoid overfitting.

In [99], Zheng et al. have chosen 157 programs of the many solutions to the problems of the ACM-International Collegiate Programming Contest (ACM-ICPC) [60] to cover many different applications.

After the training phase, the testing phase validates the predictions made by the trained model. These models show a high accuracy on the testing set on various architectures (around 90% for [99]) with a much faster speed than a reference cycle-accurate simulator. These characteristics are ideal for a fast exploration of the design-space.

However, as we said before, these approaches' accuracy and exploration possibilities highly depend on the training set. Hence, the most challenging part and longest part of these methods lies in acquiring this training set on the prototypes or simulators of the hardware of interest.

### 4.3.3 Mechanistic models

Mechanistic models rely on application profiling to compute metrics to represent its behavior. The accuracy and speed of these models rely on the chosen metrics and the way to obtain them. With these metrics, it extrapolates the application performance on a target machine.

A straightforward example of such a model is the following equation (4.1) to compute the execution time of a program [71]. It sees it as a set of basic blocks of fixed execution time ($T_{BB}$) connected by an execution path with a set execution frequency ($F_{BB}$) of each basic block. In this case, the total time is equal to :

$$T_{program} = \sum_{i=1}^{i=n} (T_{BB_i} \times F_{BB_i}) \tag{4.1}$$

A single execution and profiling allows one to obtain the frequency and execution time of each basic block. This model is not accurate anymore as today's processors are not in-order, and the execution time of basic blocks can vary due to pipelining and cache effects.

With today's complex interactions between machines and applications, it needs to make simplifications and hypotheses in this performance extrapolation according to the application's metrics. One such simplification is the hardware-independent representation of the application used by Jongerius et al. in [67, 66].

With this approach, they use an architecture-independent LLVM Intermediate Representation to represent the application and analyze its behavior with a machine of fixed parameters. This analysis results in an Instruction Per Cycle (IPC) metric to estimate performance. As the focus of this model is to help hardware constructors explore a large design space at the early stage of system design, they neglect architecture-focused software optimization to simplify the extrapolation. This choice leads to prediction error up to 50% but with a high enough correlation with observed performance to enable a first exploration of a large design-space.

Hence, to use such a mechanistic model, the first step is to define its usage and use environment. Such a model cannot make predictions as accurate as simulations or application-dependent models. However, it can allow for fast and accurate enough generic prediction if it is used to explore the parameters it has been designed for.

### 4.3.4 Discussion

If we want to rapidly explore a vast design-space on various applications, relying on analytical models to predict performance is a good candidate. However, we have split many possible approaches into three groups: the statistical approaches,

the learning-based methods, and the mechanistic models. Statistical and learning-based approaches are usually easier to implement with fast prediction and good accuracy, but they require a good amount of data available for their training. Hence, without access to many data, relying on a mechanistic model is the best choice. In this case, the first step to choosing an approach is to define the usage of the prediction approach and the environment it will be used in. These models need to make assumptions or simplifications because the interaction between today's applications and machines is too complex to model mathematically.

## 4.4 Conclusion

To conclude this chapter, there are many possible approaches to predict the performance of HPC applications in the Arm environment. However, there is no fast, accurate, and generic approach. Hence, we need to choose one approach according to how it will be used for design-space exploration. The following Figure 4.3 sums up what characteristics of the approach would direct the choice of a specific approach to enable design-space exploration in a codesign environment.

With these directions, as we want to explore a large design-space with a vast pool of applications, a good solution is to use analytical models. Moreover, because Arm is a recent actor in the HPC environment, we cannot rely on statistical or learning-based methods as we cannot gather enough data in the design-space to train such models. Then, the most appropriate solution would be to use a mechanistic model to predict performance. To design such a model, we first need to define its usage and the codesign environment it will be used in to best design such a model. It is the aim of the first chapter of our contribution.

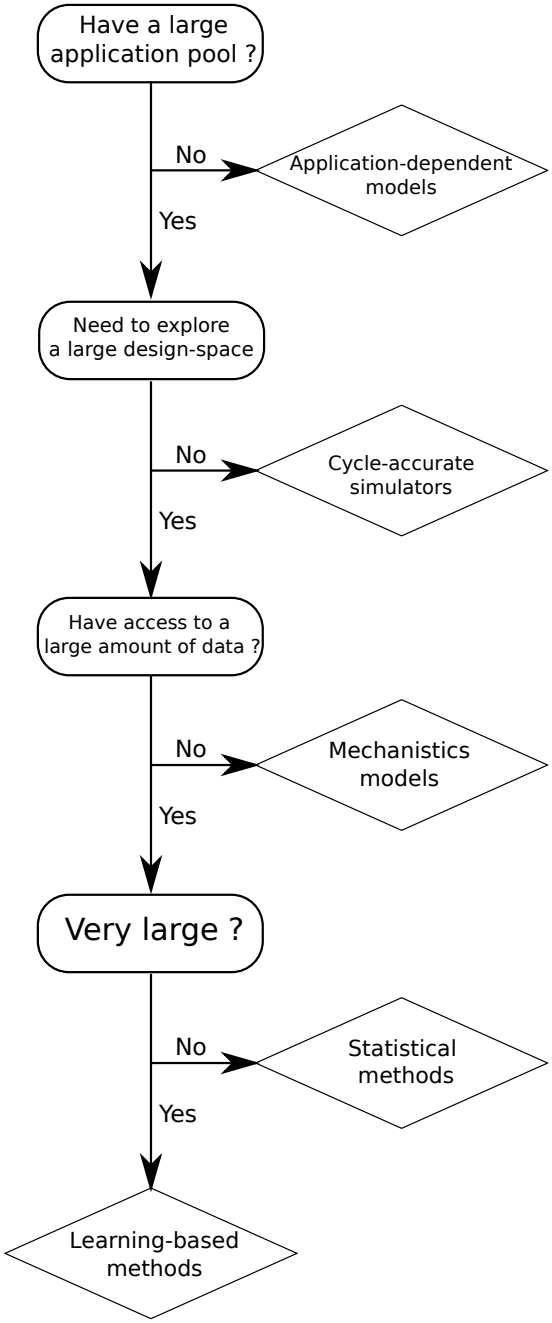


Figure 4.3: Directions to choose a prediction approach for design-space exploration

# Part II

# Contributions

# Chapter 5

# Setup of the performance projection methodology through codesign environment definition

This work aims to propose a performance prediction approach to enable codesign initiatives on future Arm processors. The performance of HPC applications depends on many factors. With so many factors from different fields intervening in the performance of applications, it is essential to define the aspects where each actor of a codesign environment can give feedback and make design choices. Moreover, as a mechanistic approach is a reliable solution to our needs, this further increases the need to define our environment.

## 5.1 Codesign environment definition

As the expertise domains of the actors of the HPC environment vary, it is essential to assess each aspect that can impact performance. Following this need, we have chosen to define the aspects of different impacts on performance in 3 aspects:

- **Application**: Source code, Numerical schemes, data structures, ...
- **Software stack**: Compilers, libraries, runtimes, ...
- **Hardware**: Memory hierarchy, use of SIMD/FMA, Instruction-Level Parallelism (ILP), ...

The interaction of all these aspects defines the performance of an HPC application (see Figure 5.1). Moreover, the actors intervening in the design and usage of HPC supercomputers can all be linked to one of these aspects: researchers

and users develop an application that runs on hardware designed by constructors thanks to a software stack developed by software engineers. Therefore, each of these actors has a different background and expertise fields.

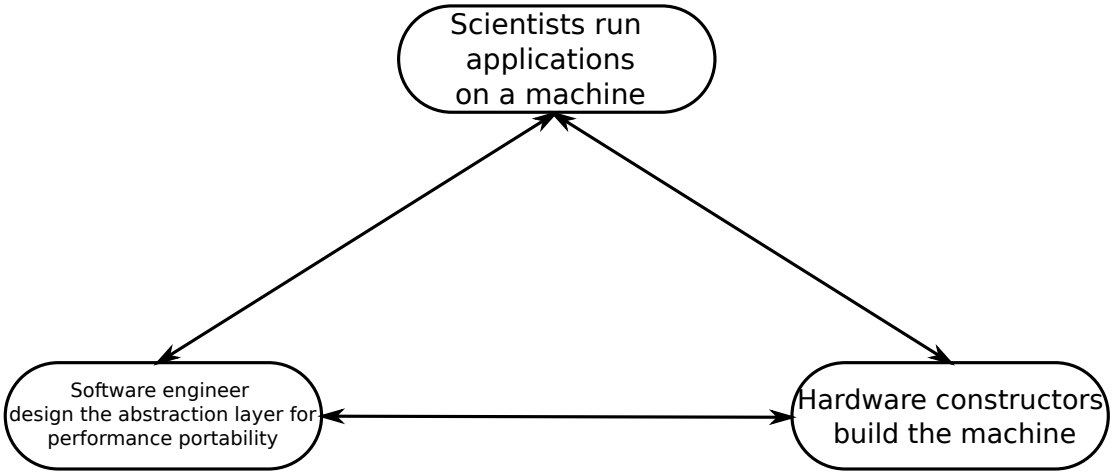


Figure 5.1: Schema of the triplet of aspects interacting between each other to deliver a performance on an available machine. Information about their interaction is available through profiling in this example.

In the case of performance optimization on a current machine, working on these three aspects of performance is often a tedious task that is simplified by profiling tools. However, in a codesign environment, the hardware characteristics are not yet defined as the future hardware is not yet accessible (see Figure 5.2). Hence, the impact of the interactions between the application and software stack with the hardware on performance is not measurable with profiling tools.

One solution could be to design a prototype for each set of hardware parameters of interest, as it would be the most accurate solution and allow the use of profiling tools to understand the interaction that intervenes in the target triplet. However, there are more time and cost-efficient solutions for the early design stages. Hence, the possibility of exploring the design-space thanks to performance prediction is a key component to enable codesign.

## 5.2 Characteristics of our performance prediction approach

With the previous definition of our codesign environment splitting the parameters impact the performance of HPC applications on machines in 3 groups, we need to

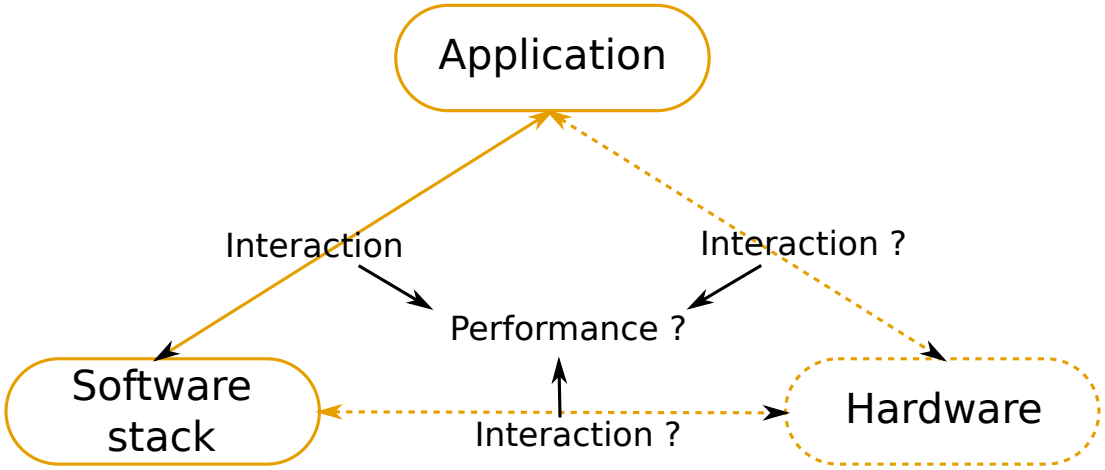


Figure 5.2: Schema of the triplet of aspects interacting between each other to deliver a performance on a future machine. In this case, information on the interaction between hardware and the two other aspects is not easily available, leading to an unknown performance.

rely on a performance prediction model with the following characteristics to open up design-space exploration and codesign :

- **Fast** to rapidly explore the design-space and the different values of the parameters of interest with representative applications;
- **Generic** to be able to cover a good margin of the HPC workload through many different applications;
- **Insightful** on the impact of design choices of each aspect on performance and their interaction.

Because of the many parameters to explore in all aspects with many different applications, relying on an application-dependent approach or cycle-accurate simulation does not seem like the most appropriate solution. Furthermore, with the small pool of Arm HPC or servers CPU available, relying on "black-box" approaches such as learning-based or statistical methods would be limited. The small amount of data available would hinder their accuracy and capacity to explore the design-space. Moreover, in our environment, relying on an approach that allows us to understand the impact on the performance of the interaction between all three aspects is essential. In this case, relying on mechanistic performance modeling is the best approach according to our definition of the codesign environment.

## 5.3 Choice of a performance projection approach

In our vision of codesign, we have split the parameters that impact applications' performance into three groups and chosen to rely on a mechanistic approach for performance prediction. As such, we need to be able to study and explore the parameters of all of these aspects independently. Moreover, we need to isolate how a parameter change impacts the performance.

In this case, relying on a hardware-independent model such as Jongerius et al. [67, 66] would limit the possibility of studying the impact of software stack targeting a specific architecture. It is further motivated by the fact that the Arm HPC environment is still recent, and the software stack choices have an important impact on the performance of applications. One example is the significant impact on the performance of the compiler choice when targeting the Fujitsu A64FX processor [49]. Furthermore, the prediction approach of Van Den Steen et al. [45, 95] extending the Interval Model of Eyerman et al. [52] is too focused on hardware exploration to open the possibility of application and software parameters exploration.

As the complexity of the interaction between machines and applications increases, mathematical performance estimation is becoming more and more difficult in these models. More and more phenomena need to be estimated to compute an accurate prediction, such as bandwidth limitations or load balancing.

In order to alleviate this need to model every phenomenon, we have chosen to rely on a performance projection approach (see Figure 5.3) using a source application/software stack/hardware triplet to estimate the behavior of a target triplet.

## 5.4 The performance projection workflow

The idea of such an approach is to measure performance on a source triplet, analyze the impact of the different aspects interactions, and project the performance on a target triplet according to the differences between this source triplet and the target triplet. Then, the objective is to model how the differences between source and target triplet characteristics impact the performance. With this idea of performance projection, we split its workflow into three steps:

- **Characterization**: Obtain metrics that represent the impact of all three aspects of performance on both source and target triplet. As the target machine may be inaccessible, metrics representing its hardware aspect can be either extrapolated, computed, or given by the constructor. However, metrics modeling applications and software stack impact can be obtained through cross-compilation, emulation, or simulation, depending on the precision needed.

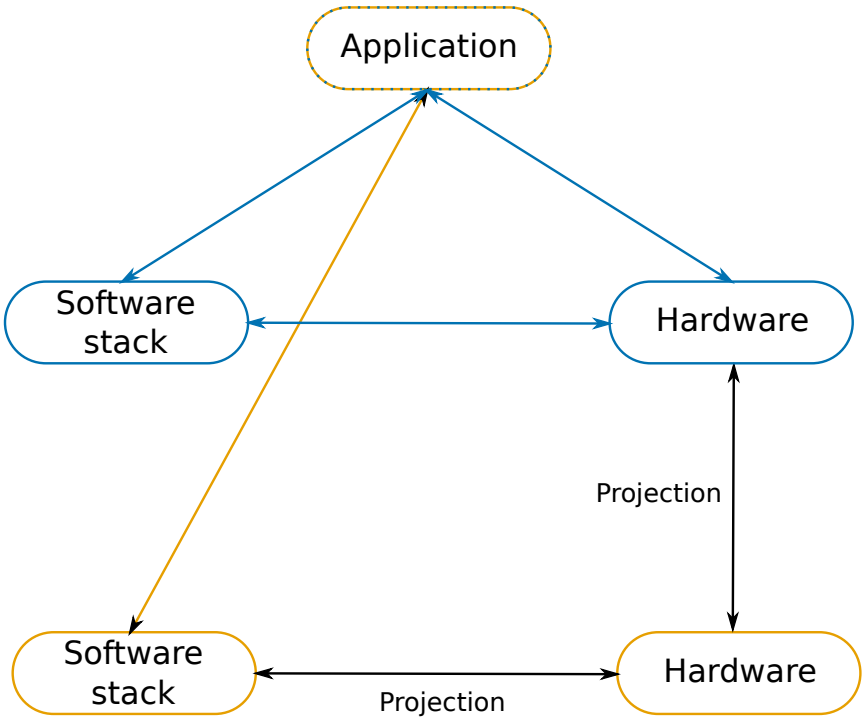


Figure 5.3: Projection approach between an accessible source and a future target triplet.

- **Source triplet analysis**: Use the source triplet metrics and access to the machine to get performance and analyze the impact of the interactions between aspect that leads to this performance.
- **Target triplet projection**: Model the evolution of performance on the target triplet according to the evolution of the metrics on all aspects and the measured impact on the source triplet performance.

The detailed workflow is described in Figure 6.1. The **Characterization** consists of two parts: the source hardware, application, and software stack characterization and the target hardware, application, and software stack analysis. The goal of the first part is to measure the impact of the parameters of all three aspects of the source triplet. We consider the source application and the source software stack together as we cannot neglect the impact of this software stack in our projection. This characterization can be done through static or online analysis. We also characterize how they interact with the source hardware. Then, the hardware characterization objective is to gather representative metrics through benchmarking or precise characteristics description. Thanks to execution on the accessible

source machine, we use these gathered metrics in a performance analysis model for **Source triplet performance analysis** to measure the impact of the parameters of each aspect on the application's performance running on the source machine.

The results of this analysis are then used with the second part, the target triplet characterization, to project the performance. The objective of the target triplet characterization is to gather the same metrics as the source triplet characterization. Consequently, the application and software stack characterization also rely on static or online analysis but would require emulating a non-native set of instructions. As it is inaccessible, the target hardware characterization relies on extrapolation or data given by the constructors. Finally, the last step is to project the performance of the target application with its target software stack on a hypothetical or inaccessible target hardware thanks to a **Projection model**. This model uses the differences between metrics obtained with the source and target characterization but also their measured impact on the application's performance. In this performance projection workflow, its execution time overhead and the prediction accuracy depend on the metrics needed in the analysis and projection model, the way to obtain them, but also on the distance in the design-space between the source and target triplet.

In our model (in Chapter 6) and its multicore extension (presented in Chapter 7), we look at how the maximum performance boundary changes according to the differences between source and target triplet and make projections according to this evolution of the limitations while expecting a similar behavior on non-modelized interaction. It means that we also project the impact on the performance of the non-modeled phenomena from the source triplet to the target one. In this case, the exploration of these parameters should consider this model's characteristics and restrict the parameter exploration to the parameters modelized by the projection approach.

Once the performance projection is defined and accurate enough, the comparison between the projected performance of each value of the parameters of interest leads to a discussion on the choice and feasibility of the most optimal parameters in the interest of each actor.

## 5.5 Conclusion

In this chapter, we have defined the codesign environment in which our performance prediction approach is used. It showed that relying on a mechanistic for fast exploration with the possibility to assess the impact of the software stack on performance seems like a good solution for prediction. We have chosen to use a projection methodology from a source application/Software stack/Hardware triplet to a target triplet of interest. Finally, using the approach to compare the evolution of

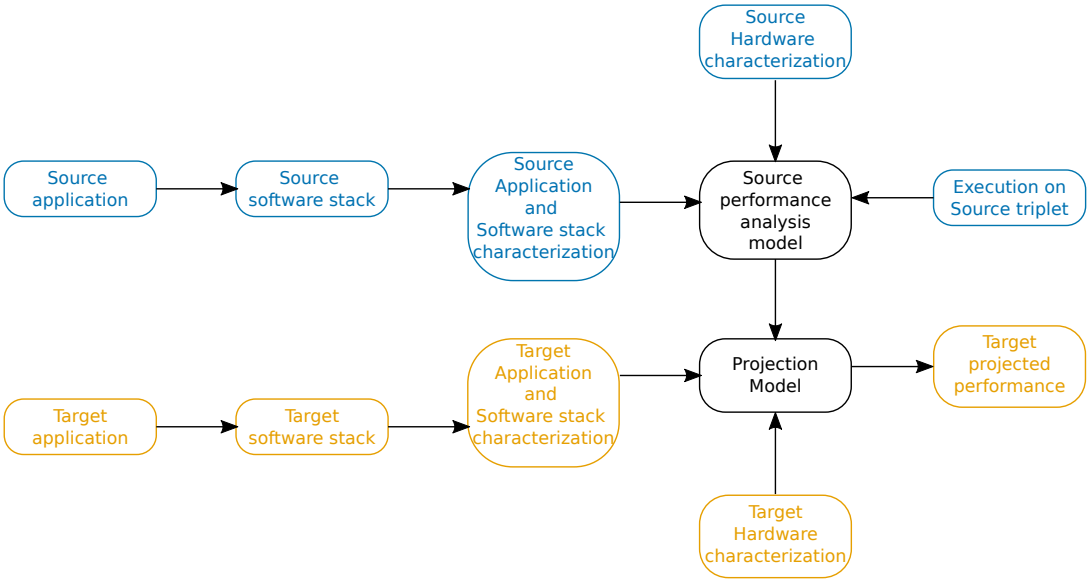


Figure 5.4: Detailed overview of our performance projection workflow.

the predicted performance with the evolution of the targeted hardware, software, or application parameter is a natural way to assess the most optimal design choices between parameter values. The next step is to define the performance analysis method and the projection approach focused on our parameters of interest.

We have chosen to focus our codesign environment and exploration on the performance of applications. This focus has driven our choices and studies presented in the following chapters. However, this may not be the central question for every actor involved in HPC, as power consumption of applications is also an important matter. With the introduction of Green500 [7] in June 2013, there is no denying that the power consumption of today's applications and machines is not one of the main focuses of the HPC today and in the future. Nevertheless, the solutions to these problems are not orthogonal as an HPC machine designed for its user's application workload also translates to better machine usage efficiency and fewer power losses during application execution. Moreover, optimizing a machine for its users' application needs is at the core of our codesign idea.

# Chapter 6

# Exploration of hardware parameters impact on HPC applications single-core performance

In the previous chapter, we have defined a codesign environment to use our projection model. We have also proposed a three-stage workflow adapted to our needs. This chapter aims to implement this workflow to explore hardware parameters around single-core performance. This exploration is motivated by the fact that, even if today's applications are multicore, the single-core performance of HPC applications is still vital. We often use Amdahl's law [24] to describe multicore scaling limitations of applications by sequential execution time. One way to enhance parallel applications' scalability is to reduce their sequential execution time. Hence, the single core performance of an application is still vital to help scale it on a full node. High single-core performance relies on efficient hardware usage as there is no communication.

To perform this single-core hardware parameters exploration, we first must define the projection approach between a source and a target application /hardware /software triplet. We have chosen to use a projection approach relying on a roofline representation of performance.

## 6.1 Single-core projection model

The projection workflow's core idea is to analyze the performance of a source triplet and use it to obtain a projected performance on the target triplet.

In performance analysis, the Roofline model, initially theorized by S. Williams et al. [97], is a well-known representation of an application's performance according to the hardware limitations. There are two crucial parts to such an analysis: the

roofline, representing the hardware's limitations in GFLOPS, and the Operational Intensity (OI), defining the application's behavior in FLOP/Byte. We also add a software characterization that impacts both of these values. Finally, we use this roofline analysis to perform projection after characterizing the target triplet.

Hence, the first step of our workflow is the characterization of both triplets. Figure 6.1 presents an overview of the differents step of the single-core roofline projection workflow. It is the same workflow as we have defined in the previous chapter. Hence, the first step is the Characterization. And we split this characterization stage into two parts: **Hardware Characterization** and **Application and Software stack Characterization**.

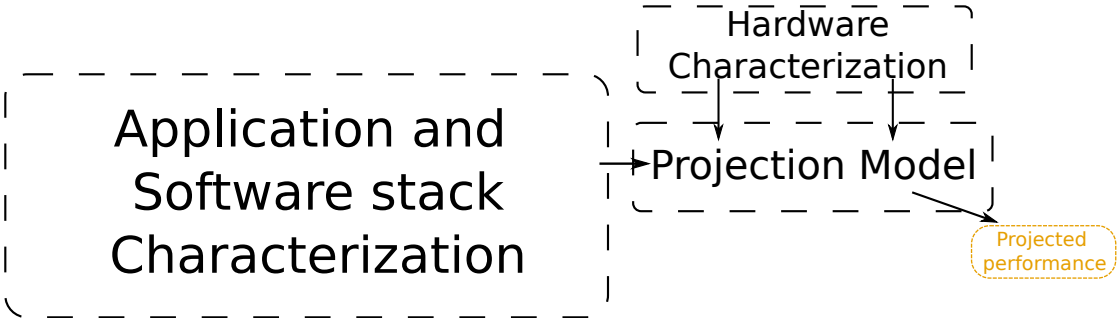


Figure 6.1: Overview of the single-core roofline projection workflow. Target triplet is in dotted yellow.

### 6.1.1 Hardware Characterization

In the roofline model, the roofline represents hardware limitations. Consequently, this roofline should represent an attainable limit in line with what we want to study. In our roofline, we have chosen to represent these limitations through benchmarking with STREAM Triad [80] and High-Performance Linpack [85] (Figure 6.2). They are well-known HPC benchmarks used to characterize bandwidth (STREAM) and peak compute performance (HPL) of machines [1].

Hence, the first study is to obtain these hardware limitations imposed by the peak memory bandwidth for every memory level and the core peak sustainable performance by running the Stream and HPL benchmarks. In the case of a hypothetical or an inaccessible target machine, we consider these values to be given by the constructor, as they are generic benchmarks, or we can extrapolate them.

Once we have obtained these values, we obtain, for a memory level, the roofline described by the following equation (6.1) with $BW_{STREAM}$ [80] the level data band-

[1] HPL is the benchmark used to rank the most powerful machines in the Top 500

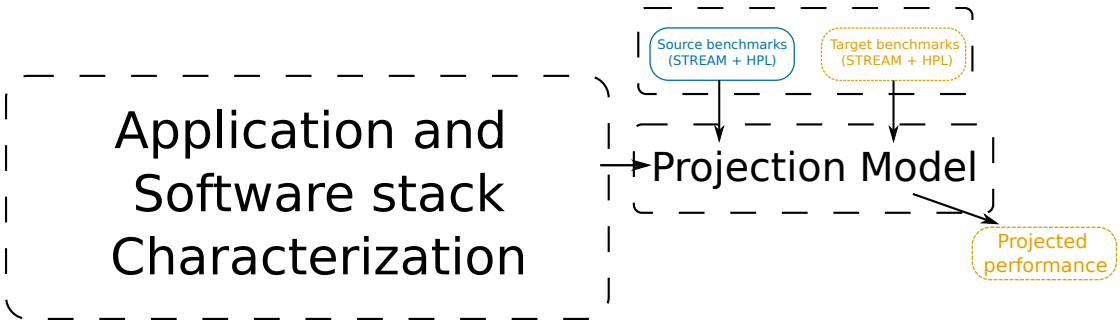


Figure 6.2: Hardware characterization through STREAM and HPL benchmarking. Source machine is in plain blue and target machine is in dotted yellow.

width measured by Stream Triad and $\text{Perf}_{\text{HPL}}$ the peak sustainable performance measured with HPL. Figure 6.3 presents the obtained roofline.

$$\text{roofline}(\text{OI}) = min\left[\text{BW}_{\text{STREAM}} \times \text{OI}, \text{Perf}_{\text{HPL}}\right] \tag{6.1}$$

However, using only HPL performance as a computing speed limit is unrealistic as our applications often do not have the floating-point instruction mix to reach that performance peak regarding SIMD and FMA usage. Consequently, we ponder the compute part of the roofline in the software impact characterization. We obtain this ponderation thanks to the metrics obtained during application and software characterization.

### 6.1.2 Application and software characterization

There are two parts to the application and software characterization: HPL ponderation and Operational Intensity (OI) description. Figure 6.4 presents the workflow of this analysis.

Because we want to assess the impact of hardware parameters, the source code of both source and target application stays the same. Hence, we generate two binaries (source and target) through their dedicated software stack targeting their respective hardware. Then, we analyze these generated binaries to obtain three sets of metrics thanks to instruction and cache analysis:

- The total of **Floating-point operations processed and Bytes moved** during the execution to compute the application's OI;
- The total **hit ratio of each memory level** to generate different OIs for each memory level;

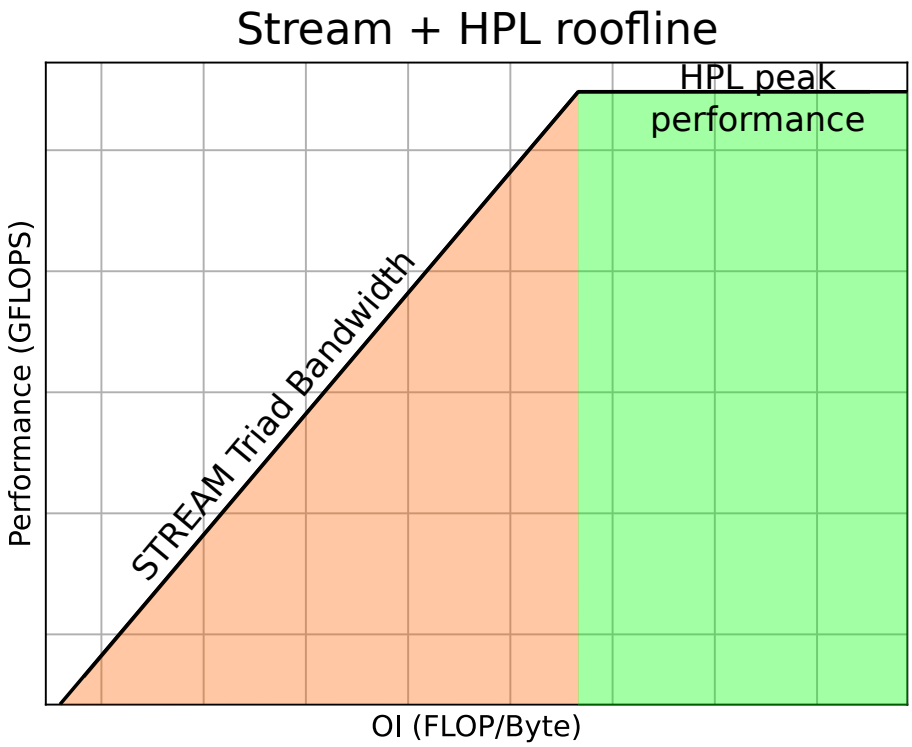


Figure 6.3: Obtained roofline by running STREAM and HPL benchmark.

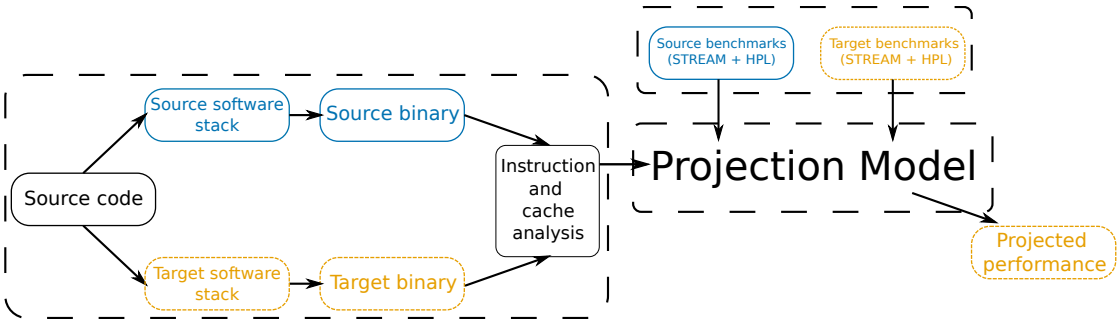


Figure 6.4: Software and application analysis. Source triplet is in plain blue and target triplet is in dotted yellow.

- The total of **floating-point instructions** to characterize the SIMD and FMA usage of the instruction mix.

With the last metric, we have chosen to weigh the HPL peak sustainable perfor-

mance of a single core following the equation (6.2). It is similar to the ponderation used by Marques et al. in their extension of the Cache-Aware Roofline Model [78].

In this equation, we compare the application floating-point operations per instruction to the maximum attainable on the machine, represented by FMA instructions on full vectors. With such a ponderation, the compute part of the roofline now represents the maximum sustainable performance for our application instruction mix and not the maximum attainable only by applications with a similar instruction mix to the HPL.

$$\text{Perf}_{\text{HPL}_{\text{ponderated}}} = \frac{\text{Perf}_{\text{HPL}}}{2 \times \frac{\text{vector size}}{\text{datasize}}} \times \frac{N_{\text{floating point operations}}}{N_{\text{floating point instructions}}} \tag{6.2}$$

Now that we have a hardware limitation ponderated by the software and application analysis, we need to place the application on the x-axis of the roofline chart according to its OI. We want to consider the possible impact of every memory level, as we cannot assert which memory level limits the performance for our projection purpose. Hence, we have considered many OIs, one for each memory level. Then, we define the OI of a memory level with the Bytes that come from this memory level and higher[2].

Hence, in a two cache-level machine, we obtain these three OIs using the equations (6.3) with $B_i$ the total of bytes accessed in the memory level $i$.

$$\begin{aligned} \text{OI}_{L1} &= \frac{N_{\text{floating point operations}}}{B_{L1} + B_{L2} + B_{\text{Main Memory}}} \\ \text{OI}_{L2} &= \frac{N_{\text{floating point operations}}}{B_{L2} + B_{\text{Main Memory}}} \\ \text{OI}_{Main\ Memory} &= \frac{N_{\text{floating point operations}}}{B_{\text{Main Memory}}} \end{aligned} \tag{6.3}$$

As a note, the OI from the L1 memory level is the same OI defined in the Cache-Aware Roofline Model of Ilic et al. [61], and the OI of the main memory is the one used in the Original Roofline Model of Williams et al. [97].

Thanks to this characterization, we now have rooflines representing the limits of both triplets and the position of the application on the x-axis. The last step is to use the roofline performance analysis on the source triplet to project performance on the target triplet.

[2]This is different to the OIs defined in the Hierarchical Roofline Model (HRM) defined in Intel Advisor [63] as they only consider the Bytes of the studied memory level

### 6.1.3 Performance Projection

We perform the projection in 2 phases: performance analysis on the source triplet and projection on the target triplet according to its OIs and rooflines (see Figure 6.5).

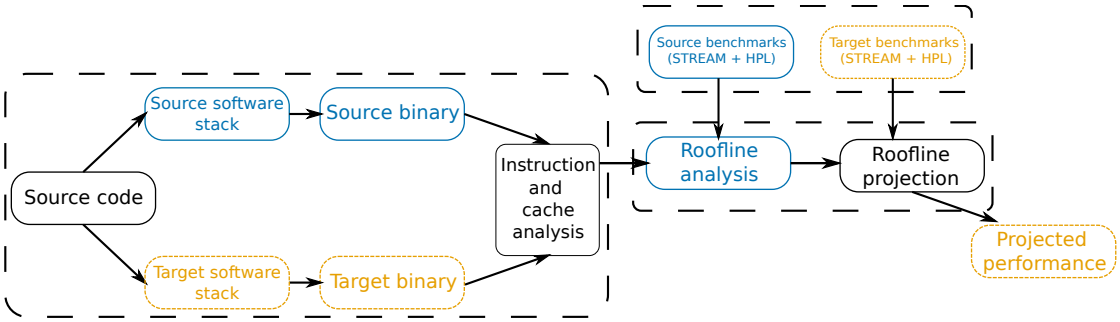


Figure 6.5: Projection model in two stages: Roofline analysis and Projection. Source triplet is in plain blue and target triplet is in dotted yellow.

The projection uses the same idea as Kwack et al. for roofline projection [74]: it considers the ratio between the performance ($\text{Perf}_{\text{source}}$) (obtained with an execution) and a source machine roofline at the $\text{OI}_{\text{source}}$ ($\text{roofline}_{\text{source}}(\text{OI}_{\text{source}})$), and projects this ratio on the target machine using the new OI and the new roofline ($\text{roofline}_{\text{target}}(\text{OI}_{\text{target}})$). Equation (6.4) defines this projection. Thus, depending on the OI value, the memory-level bandwidth or core peak performance limits the application's performance.

$$\text{Perf}_{\text{target}} = \frac{\text{Perf}_{\text{source}}}{\text{roofline}_{\text{source}}(\text{OI}_{\text{source}})} \times \text{roofline}_{\text{target}}(\text{OI}_{\text{target}}) \tag{6.4}$$

#### Projection Interval

In eq. (6.4), the source and target OIs and rooflines are not specified. However, we consider several OIs depending on the memory level to cover the whole memory hierarchy (from the L1 cache to the memory bank of the other socket (OS) for a NUMA node). Moreover, we project performance according to the roofline of each memory level impacting the considered OI. For example, in a two-cache machine, we use $OI_{L2}$ for two projections: one according to the L2 roofline and another with the DRAM roofline. Hence, depending on the OI of a memory level and the considered roofline, the application is limited by either the considered memory-level bandwidth or the ponderated peak performance. In the former case, this analysis results in multiple projected performances (one for each OI/roofline), forming a projection interval (e.g., by applying (6.4) on each OI). In the latter case, the

multiple projections result in the same value as the peak performance does not depend on the data access pattern and is the same for every OI.

As a conclusion, the Figure 6.6 sums up the following steps of our projection approach between two-cache machines:

- **1**: Roofline characterization thanks to STREAM on every memory level and HPL values
- **2**: HPL ponderation with the instruction mix. In this example, even if the HPL of target machine is higher, the instruction mix of the target binary does not use SIMD and FMA to ureach this level.
- **3**: Source analysis of a single OI/roofline on the source machine with the measured source OI and performance. We obtain the Ratio source of efficiency from this analysis.
- **4**: Projection of the obtained Ratio source on the OI and roofline of the target triplet.
- **5**: We repeat the projection for every rooflines and OI concerned. In this example, we project six points because the $OI_{L1}$ is limited by three rooflines (L1, L2 and DRAM), the $OI_{L2}$ is limited by two rooflines (L2 and DRAM), and the $OI_{DRAM}$ is limited by one roofline (DRAM).
- **6**: We gather all the points in a projection interval.

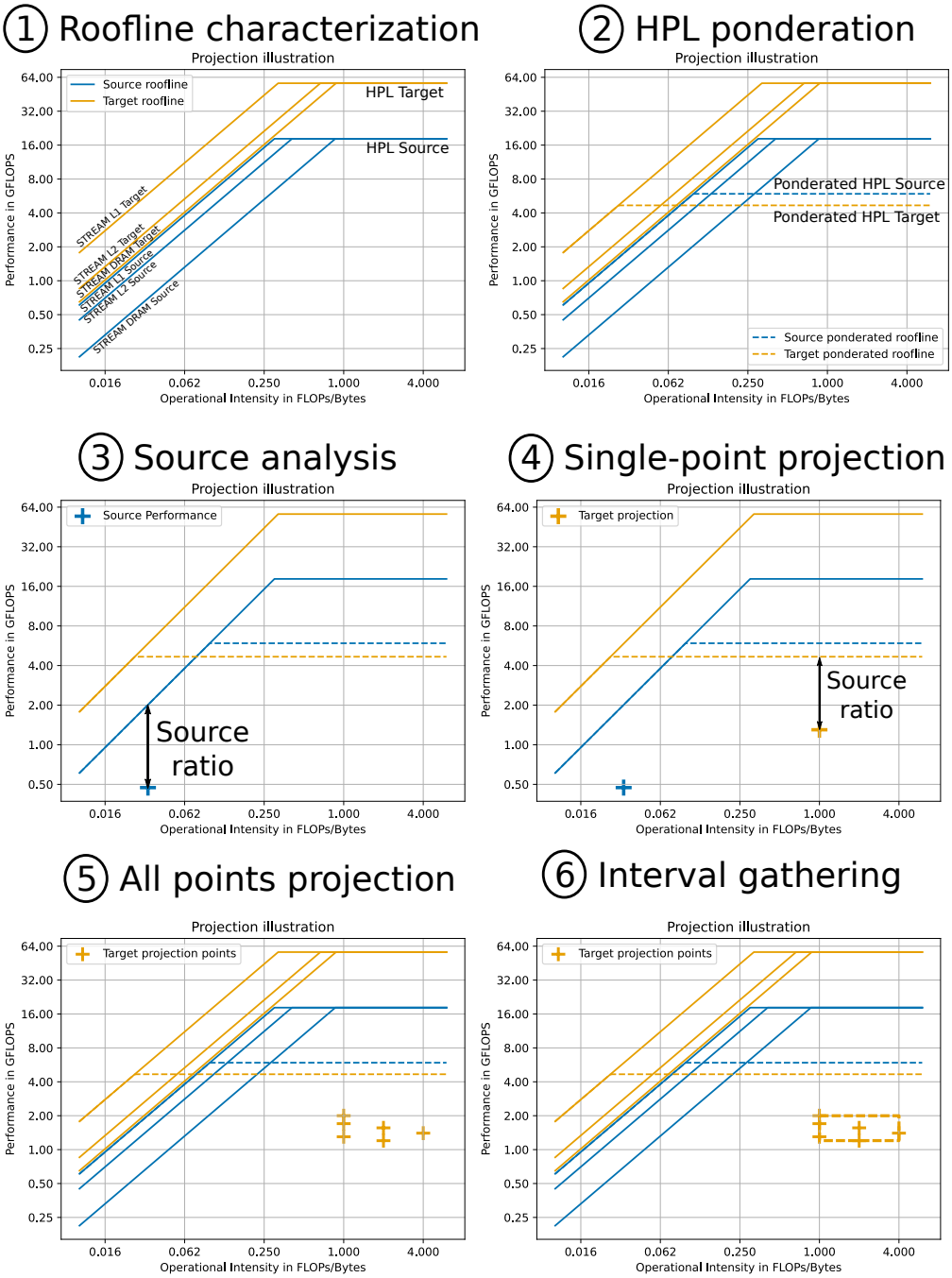


Figure 6.6: Illustration of the performance projection approach with the different steps of the projection. All the figures present the OI (FLOP/Bytes) on the x-axis and the Performance (GFLOPS) on the y-axis.

### 6.1.4 Implementation

As seen in the workflow, there are different analysis steps. All of them may be performed in parallel as they require different runs. Hardware characterization is obtained by running Stream and HPL on our source machine and extrapolating it on the target machine if it is not publicly available.

For the software and application analysis of the binaries, we need to gather two kinds of metrics:

1. **Instruction mix**: The number of floating-point instructions, the total number of accessed bytes, and the number of flops. We rely on the dynamic code instrumentation with DynamoRIO [32] and ArmIE for SVE emulation [1] to iterate over the SVE vector lengths.

2. **Cache analysis**: The total hit ratio of every memory level. In this case, we rely on the hardware counters of the source machine.

DynamoRIO is a dynamic binary instrumentation framework [33]. It is an open-source tool that allows users to instrument a binary code at execution. DynamoRIO is able to observe and manipulate every application instruction prior to its execution with the use of code caching, linking, and trace building. The control flow of this tool is described in Figure 6.7 with a context switch to split the binary code cache from the DynamoRIO code. The application code is copied into two caches: one basic-block cache and a trace cache to keep the application control flow. While in the code cache, the application code is able to be modified and analyzed thanks to an instrumentation client using DynamoRIO API.

In our workflow, we use DynamoRIO and ArmIE, the Arm SVE Emulation tool based on DynamRIO, to run SVE instructions on a machine without native SVE execution (see Figure 6.8) and count the total number of floating-point operations and Bytes moved by the application at execution by instrumenting every memory (load and store) and floating-point instructions.

The strong point of this implementation is the possibility to measure metrics at execution in order to measure according to an actual execution flow of the application, contrary to a static analysis. It also opens the possibility to analyze the impact of runtime libraries.

Now that we have a performance projection approach and its implementation in the Arm HPC environment, we need to choose an experimental environment of different source core architectures to validate the approach and explore the impact of some chosen hardware parameters on single-core performance.

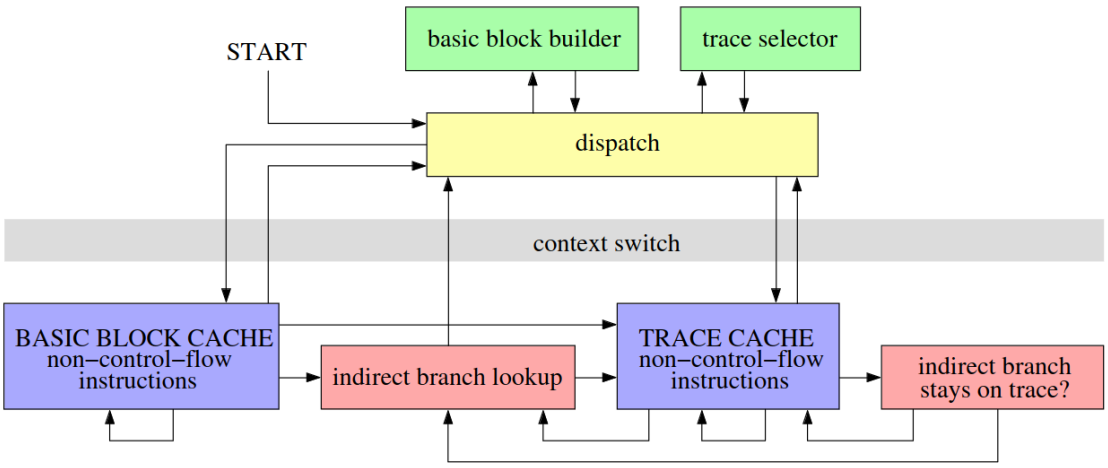


Figure 6.7: DynamoRIO flow chart. The application code is stored in the two caches separated from the DynamoRIO code by a context switch. Source: [33]

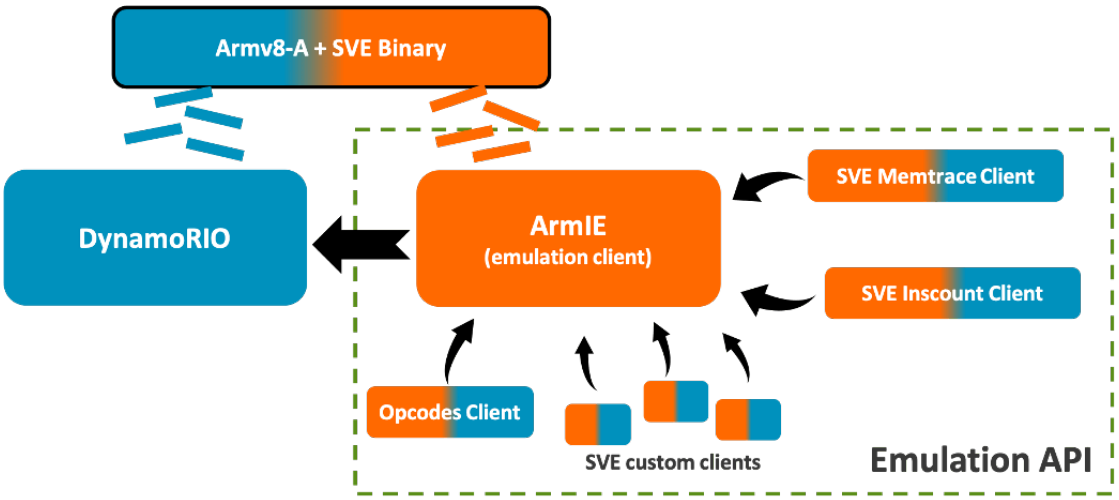


Figure 6.8: ArmIE flow chart. It is an emulation client based on DynamoRIO for SVE instructions (in orange). When it is not in emulation mode (blue instructions), it behaves like a normal DynamoRIO client. Source:[2]

### 6.1.5 Experimental environment

Table 6.1 presents all the core architectures used for validation and hardware exploration in this chapter. The benchmark valeues are obtained by running the benchmarks alone on a single-core of the CPU.

We have chosen to use three different Arm cores to experiment with our ap-

| Core | TX2 | N1 | A64FX |
|---|---|---|---|
| Performance (GFLOPS) | 17.53 | 18.22 | 56.71 |
| MM bandwidth (GB/s) | 25.43 | 21.14 | 65.52 |
| L1 bandwidth (GB/s) | 36.17 | 60.86 | 178.2 |
| L2 bandwidth (GB/s) | 29.62 | 45.14 | 85.49 |
| ISA | NEON | NEON | SVE |
| Vector size | 128 bits | 128 bits | 512 bits |
| Memory Type | DDR4 | DDR4 | HBM2 |
| Compiler | g++ 10.3.0 | g++ 10.3.0 | g++ 10.3.0<br>FCC 4.6.3<br>(clang mode) |
| Flags | -O3 -ffast-math<br>-mcpu=thunderx2t99 | -O3 -ffast-math<br>-mcpu=neoverse-n1 | -O3 -ffast-math<br>mcpu=a64fx |

Table 6.1: Single-core characteristics of our 3 source cores architecture.

proach: a single-core of Marvell ThunderX2 (TX2), Arm Neoverse N1 (N1), and Fujitsu A64FX (A64FX). Table 6.1 summarizes their characteristics and the results of HPL and STREAM single-core benchmarks running alone on a full node we obtained. These three architectures cover different parts of the Arm HPC environment, from the server market (N1 and TX2 processors) to the HPC focus (Fujitsu A64FX). With HBM2 and longer SIMD vectors than N1 and TX2, the performance of an A64FX core is much higher when running STREAM and HPL benchmarks.

## 6.2 Model validation

The following experiments aim to validate the model on three applications (LULESH, Quicksilver, and MiniFE) using three architectures (N1, TX2, and A64FX) by ensuring that the target performance is in the predicted interval obtained by our workflow when using the same software stack (Gnu Compiler Suite 10.3). We analyze each application after initialization and before finalization.

### 6.2.1 Neoverse N1 ↔ Marvell ThunderX2 projection

In this experiment, we project performance between a single core of our closest architectures: Marvell ThunderX2 and Neoverse N1. Then, we compare the projected performance interval with the performance obtained when running the application on the target machine. Results on all three applications are presented in a Roofline chart in Figures 6.9 to 6.14.

#### LULESH

Figures 6.9 and 6.10 present the projection of LULESH from one machine to the other. The maximum sustainable performance weighted by the floating-point instruction mix (corresponding to the dotted rooflines) is higher on TX2 than N1 despite having a lower maximum performance on HPL. The OIs of the L1 memory level are similar on both machines. Situated in the TX2 memory-bound region, the differences between the bandwidth and the projections in this region create an interval not modified by the projections from the OIs of L2 and main memory. This interval is higher when projecting from TX2 because of the difference in L1 and L2 cache bandwidth. Because performances on both machines are nearly equal, we are closer to the TX2 roofline. Hence, we obtain a better ratio, which is then translated into a higher prediction interval. The interval we predict in both figures includes the actual performance measured on the target machine, validating our approach in this application.

#### MiniFE

Contrary to Lulesh, MiniFE exploits vectorization more efficiently on the two architectures. Its better vectorization rate leads to a higher maximum performance of its instruction mix (see Figures 6.11 and 6.12). Moreover, the OI of L1 is in the memory-bound region of all rooflines on both machines. Once again, the interval we predict, only affected by the OI of L1, does not change whether we project from N1 or TX2. However, the N1 performance is higher (1.87 GFLOPS) than the TX2 performance (1.04 GFLOPS). Despite this difference in performance, our interval includes the measured performance. Because we are in the memory-bound region of the L1 and L2 cache levels, we can suppose that the better performance of MiniFE on N1 may result from the higher bandwidth of these levels.

#### Quicksilver

Quicksilver is our application with the lowest OI and measured performance on both machines (Figures 6.13 and 6.14). This low performance results from this

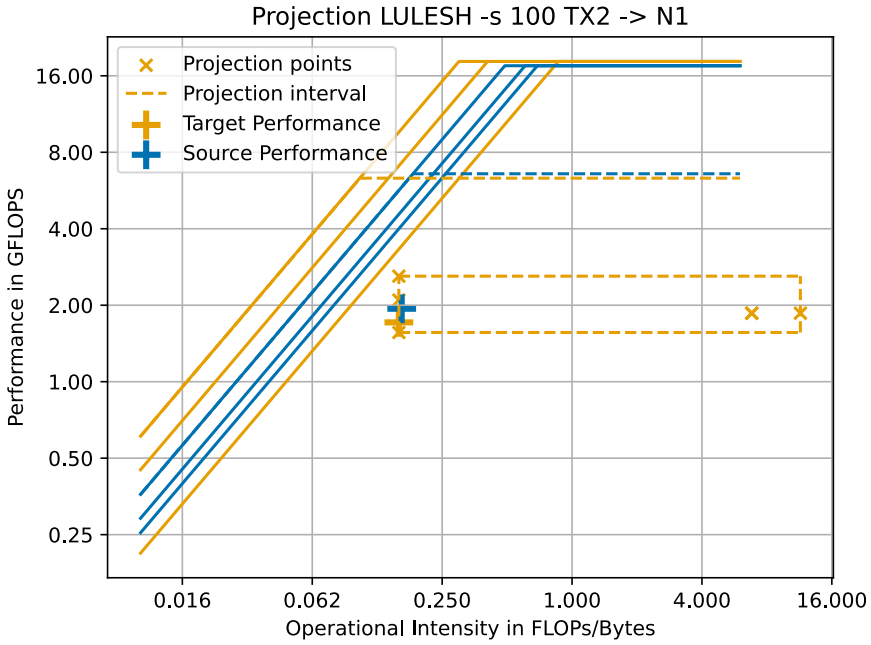


Figure 6.9: LULESH -s 100 TX2 → N1 projection results.

application's poor vectorization rate and FMA usage. It is visible in the maximum performance attainable by the instruction mix of both binaries. All the OI deducted from the L1 are in the memory-bound region of all rooflines, while the OIs derived from other memory levels are in the compute-bound regions. The prediction interval is obtained because of the OI from L1, which includes the measured performances. We observe higher performance on N1 (0.5 GFLOPS) than TX2 (0.4 GFLOPS). This difference in performance may be due to the difference in cache bandwidth, giving an advantage to the N1 core.

Hence, the measured performance between a core of Neoverse N1 architecture and a core of ThunderX2 hit the projected interval for all three applications with different behaviors. As referenced in Table 6.1, these are the cores with the closest Stream and HPL values as they both use DDR4 and NEON SIMD. However, it is also interesting to look at the projection between these cores and the A64FX core as they differ in the benchmarks' values.

### 6.2.2 Neoverse N1 and Marvell ThunderX2 → Fujitsu A64FX projection

Figure 6.15 presents the projection results from N1 and TX2 cores to an A64FX core with the same input and software stack as the previous validation experiments.

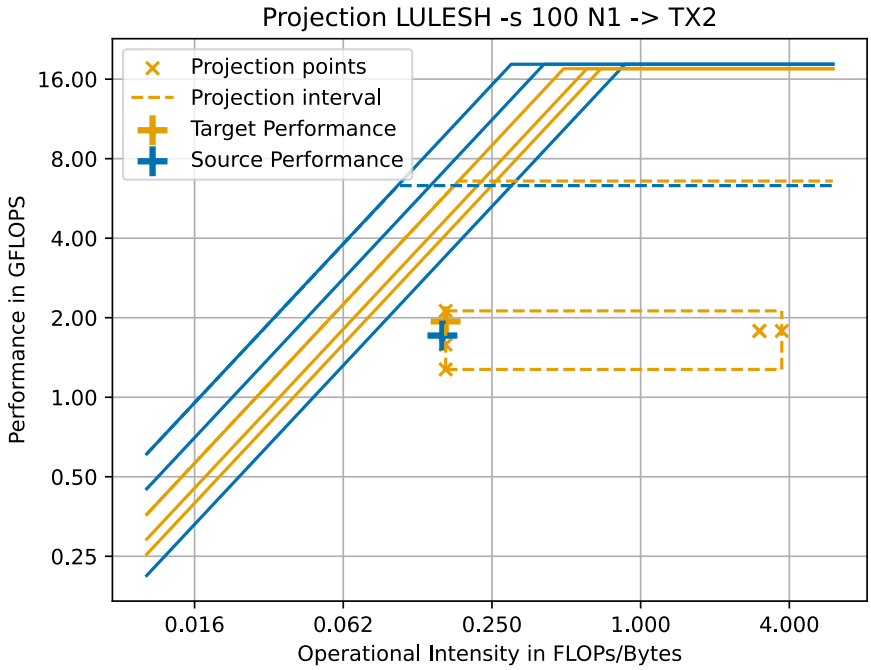


Figure 6.10: LULESH -s 100 N1 → TX2 projection results.

In this case, we over-predict the performance of all applications. The architectural efficiency of N1 and TX2 does not translate to the one observed on A64FX. As it is lower on A64FX, this explains the over-prediction.

One explanation of this low architectural efficiency is that we rely on GCC 10.3 and only on the *-mcpu* flag for architecture-specific optimization. In the case of A64FX, because of its limited Out-Of-Order resources and the long latency of floating-point instructions, more compiler optimizations such as software pipelining and loop fissioning are required for the performance of applications not to be limited by its micro-architecture [87]. This impact of the differences in Out-Of-Order resources between our cores is not considered in our projection. Hence, as we wanted to keep a similar software stack optimization between all cores, these differences in the need for software optimizations to attain a similar architectural efficiency led to this over-prediction.

In a mirrored behavior, the projection from A64FX (see Figure 6.16) to N1 and TX2 cores under-predict the performance of all applications.

To conclude these validation efforts, when we apply our model on the most similar machine in our machine pool, the prediction interval we obtain always includes the measured performance of our application. Nevertheless, when projecting performance from N1 and TX2 cores to an A64FX core, we severely over-predict the

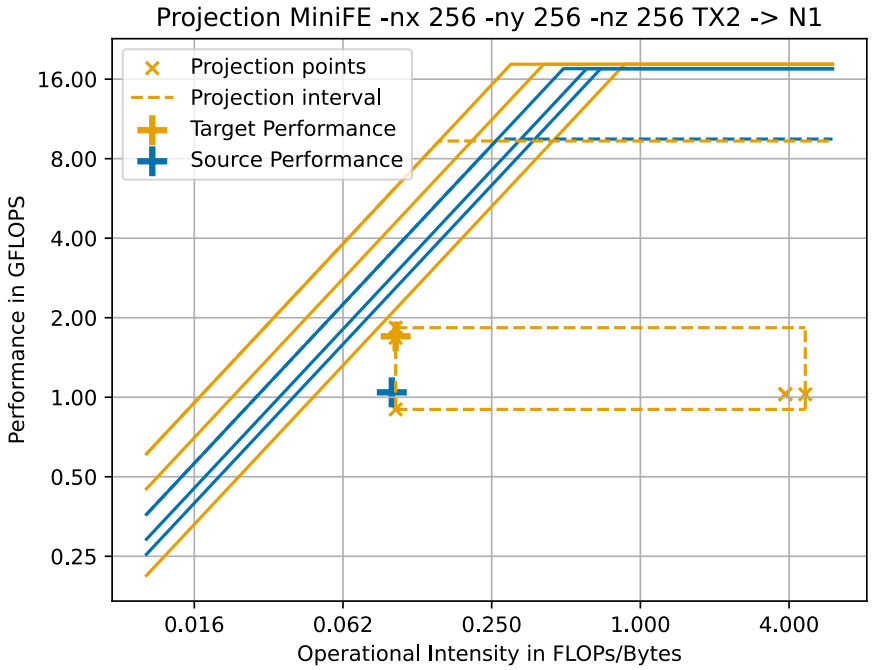


Figure 6.11: MiniFE -nx 256 -ny 256 -nz 256 TX2 → N1 projection results.

performance as the projection of architectural efficiency is too optimistic on all three applications. Hence, in the case of significant micro-architecture differences, such as differences in Out-Of-Order resources, the isoefficiency hypothesis of the projection is too coarse and misses out on micro-architecture's substantial impact on performance. However, as there are no significant micro-architectural changes between source and target machines in the following exploration experiments, the results obtained by the projection are valid enough to open discussion.

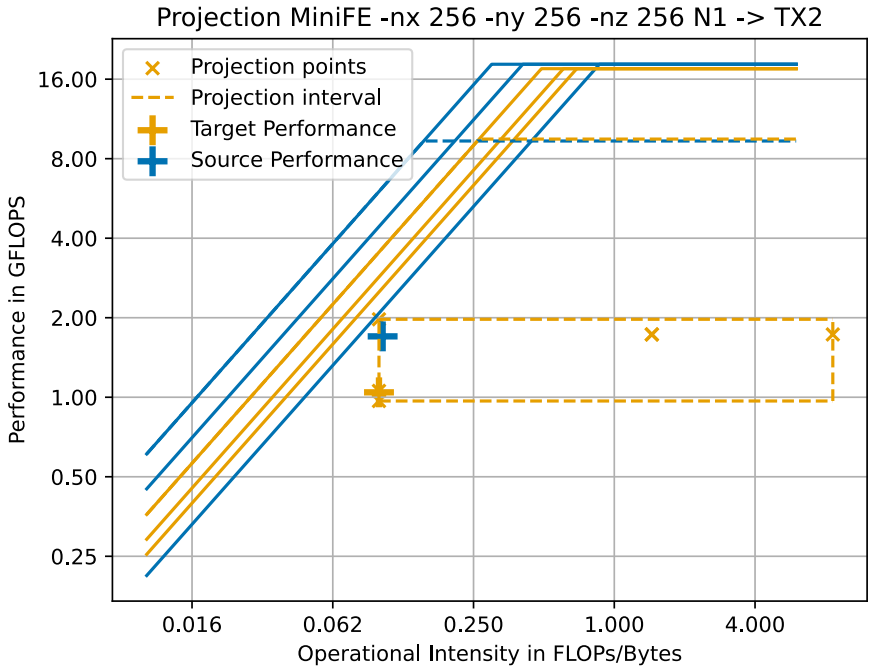


Figure 6.12: MiniFE -nx 256 -ny 256 -nz 256 N1 → TX2 projection results.

## 6.3 Hardware parameters exploration on single-core performance

In these exploration experiments, we use our approach to explore different hardware parameters, but we also study the impact of a software choice on hardware exploration. We have chosen to explore some of the different vector sizes SVE allows on all three machines. Hence, we compare the performance projection from a NEON machine (N1, TX2) to a hypothetical one with SVE with a vector size of 128, 256, 512, 1024, and 2048 bits. Another parameter we explore is the introduction of HBM2 for both DDR4 machines. Then, we combine these parameters to compare hypothetical SVE512 + HBM2 machines with A64FX. Finally, we observe the differences compiler change creates in exploring different vector sizes on A64FX.

### 6.3.1 Exploration on SVE vector sizes

One of the challenges in designing future Arm cores is the size imposed by the hardware on SVE vectors and the impact this choice has on the performance of the applications. We translate the impact of this parameter in two aspects of

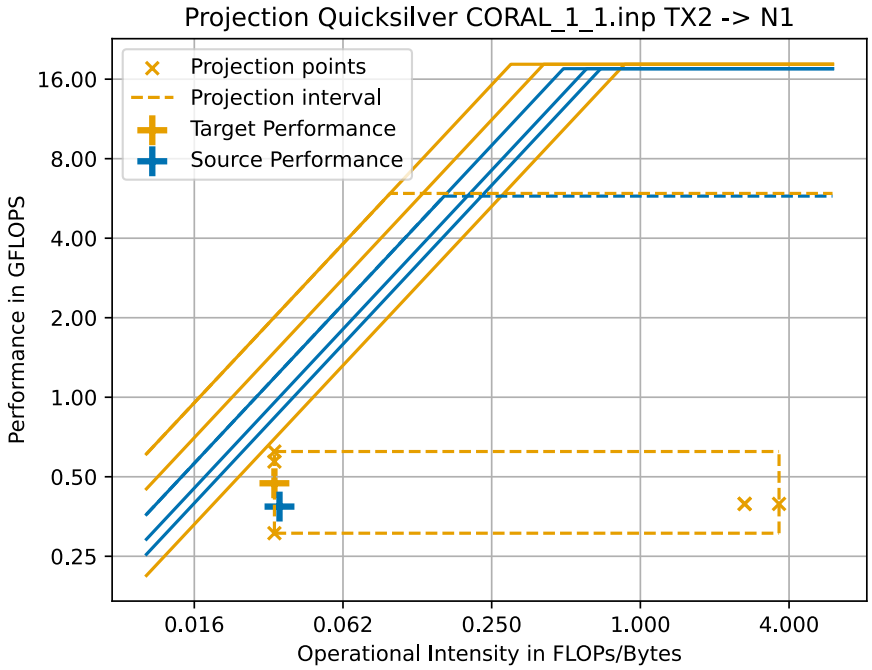


Figure 6.13: Quicksilver with CORAL Problem 1 input TX2 → N1 projection results.

our triplet: the Hardware and the Software Stack. We characterize the hardware impact with our model by extrapolating how such a change impacts the HPL performance imposed by hardware and the software stack. We have chosen a simple linear extrapolation between native and projected HPL values. Furthermore, we also look into the impact of the vector size change on the software stack aspect by compiling with an SVE vector-length agnostic flag and changing vector size at execution with emulation through ArmIE.

Figure 6.17 shows that the impact of the vector size on LULESH depends on the source machine. When targeting A64FX and TX2 architecture, the binary's predicted performance benefits more from the increase in vector size than when targeting N1. This is because GCC does not vectorize LULESH as much when targeting N1. Despite having a similar source performance on native N1 and TX2, this difference in vectorization predicts lower performance on N1 than TX2 with longer SVE vectors. It is also important to note that SVE512 is the native performance on A64FX, which is not a projection. In the case of LULESH, we do not observe a performance interval on this machine, as the higher bandwidth from the HBM2 allows for every OI of each vector size to be in the compute-bound region.

When doing this exercise on MiniFE (Figure 6.18), we observe a similar behav-

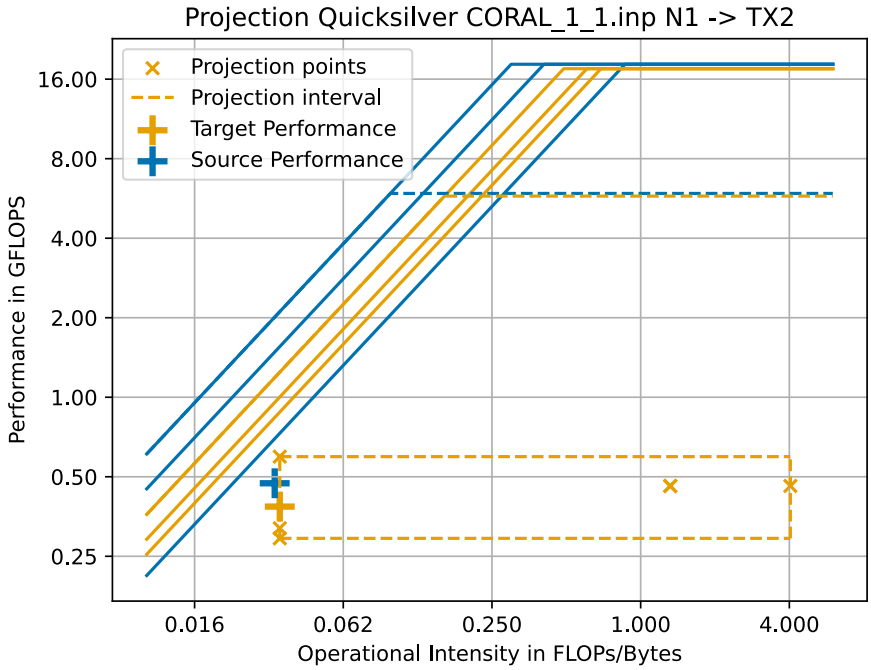


Figure 6.14: Quicksilver with CORAL Problem 1 input N1 → TX2 projection results.

ior on all machines. A change in vector size impacts all the predicted performances of our architectures. However, this impact is different for all our architectures. When comparing TX2 and N1, the predicted interval upper bound of TX2 gains more performance at each step to reach a maximum of 10.2 GFLOPS.

The behavior of MiniFE when exploring vector size is opposed to Quicksilver (Figure 6.19). This application does not benefit from the change of vector size on any architectures. On all architectures, GCC cannot vectorize the application, meaning they do not benefit from this change in vector size. Hence, there are better solutions than increasing the vector size if we want to gain performance on Monte-Carlo applications represented by this proxy app.

### 6.3.2 Exploration on the introduction of HBM2 on DDR4 machines

Another characterization we make with our approach is to analyze the introduction of the HBM2 memory of A64FX on N1 and TX2, creating a hypothetical machine with the same characteristics as our source machine but only the main memory bandwidth (Hardware aspect) change in our model.

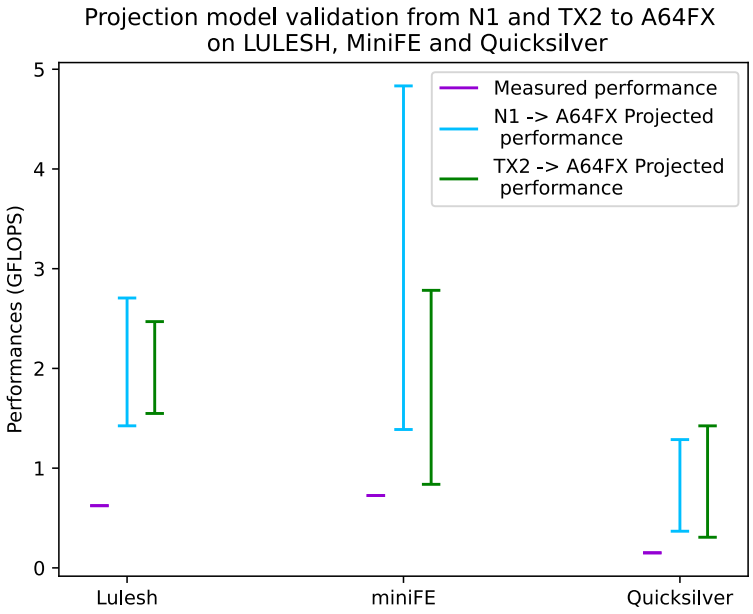


Figure 6.15: Model validation from N1 and TX2 to A64FX on LULESH, MiniFE and Quicksilver

At first glance at Figure 6.22, the memory bandwidth increase does lead to an increase in projected performance on Quicksilver in opposition to the impact of the SVE vector length study. Moreover, looking at Figures 6.20 to 6.22, the N1 core is the one that benefits the most from this change of main memory bandwidth on all applications. It leads to a higher predicted upper bound on all applications on N1 despite LULESH having less performance with DDR4 on this machine. This is likely due to the N1 core having higher cache bandwidth and lower memory bandwidth than TX2. The memory bandwidth gain is higher for the N1 core, leading to more performance gain for these applications. We also see that the lower bound of our predicted interval does not change on both applications compared to DDR4. It is due to the cache bandwidth of our hypothetical machine needing to be adapted to this main memory bandwidth increase.

The conclusion of this study on the introduction of HBM2 on the DDR4 cores is that it would lead to a performance gain for every application, contrary to the SVE vector length increase. However, a limitation of this study is that our model cannot characterize the latency aspect of our applications, which may be an issue with the introduction of HBM2 because of its latency access being higher than DRAM [98].

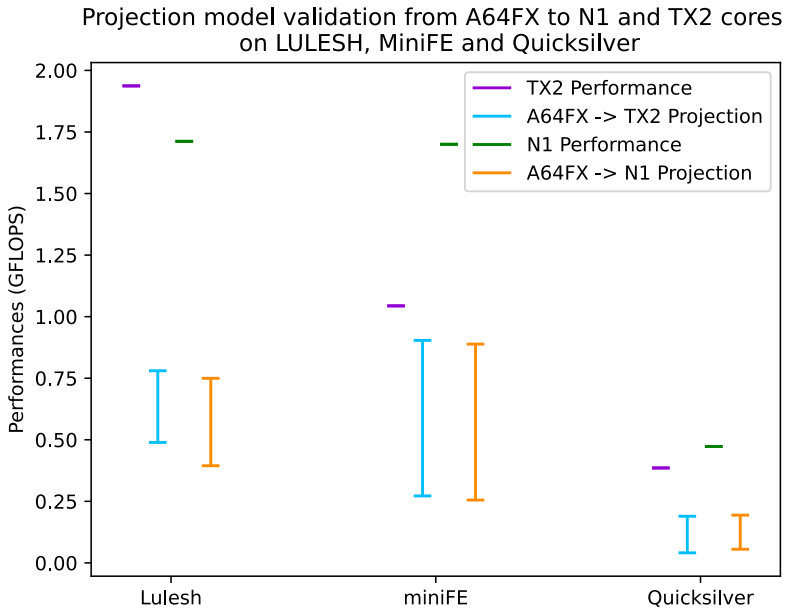


Figure 6.16: Model validation from N1 and TX2 to A64FX on LULESH, MiniFE and Quicksilver

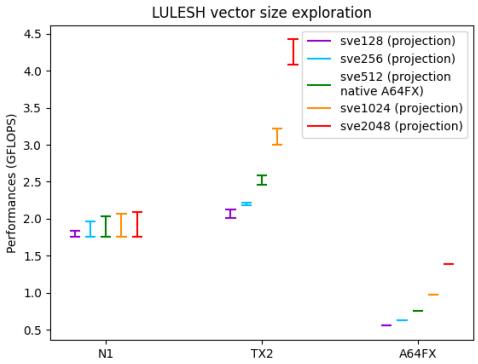


Figure 6.17: Exploration of different SVE vector sizes on LULESH

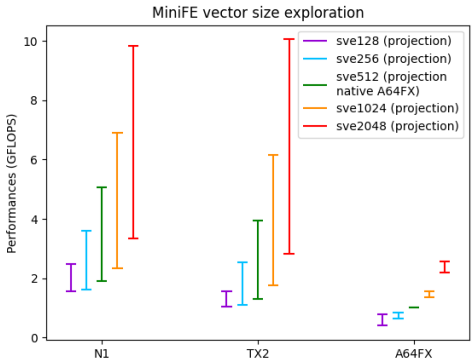


Figure 6.18: Exploration of different SVE vector sizes on MiniFE

### 6.3.3 Comparison of projections from N1 and TX2 with SVE 512 and HBM2 to A64FX

In the previous studies, we have been looking at the impact of a change in computing speed (SVE vector size) and a change in memory bandwidth (HBM2 intro-

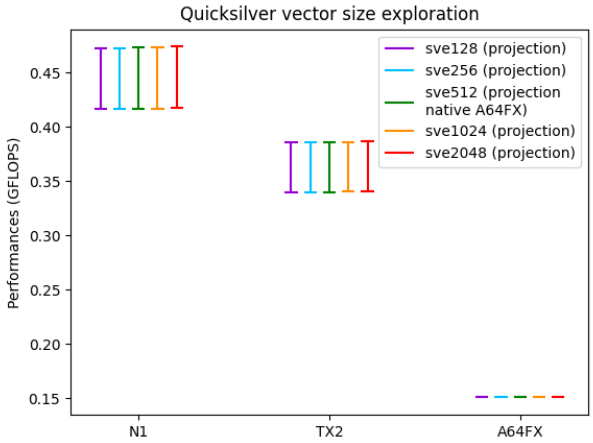


Figure 6.19: Exploration of different SVE vector sizes on Quicksilver

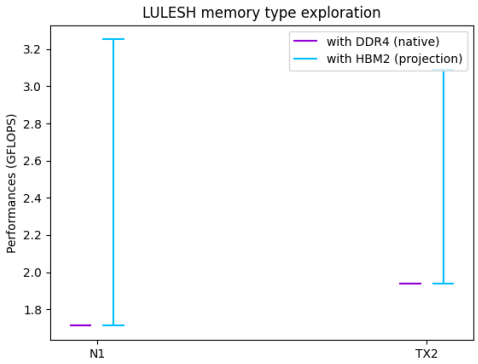


Figure 6.20: Exploration of introduction of HBM2 on LULESH

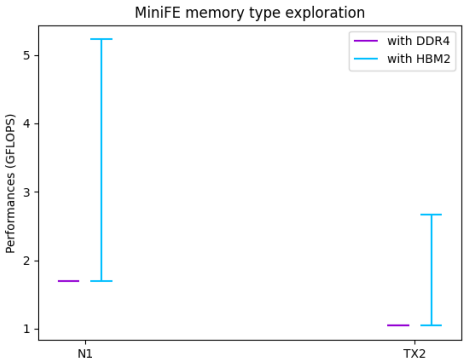


Figure 6.21: Exploration of introduction of HBM2 on MiniFE

duction). Combining and comparing them with a core that uses SVE 512 vectors coupled with HBM2 bandwidth would be interesting: the Fujitsu A64FX. We choose to use GCC 10.3 on all three machines for this comparison presented in Figures 6.23 and 6.24. We can observe the change introduced by HBM2 to the interval predicted only with SVE512.

Similarly, using HBM2 impacts the N1 core the most on LULESH, even with 512-bit SVE vectors. Even if both machines can gain more performance, this leads to similar predicted performance between N1 and TX2 despite LULESH not benefitting from vectorization on N1. On MiniFE, the predicted performance is

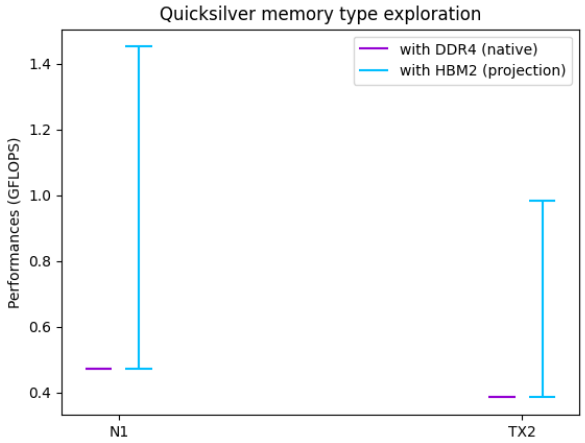


Figure 6.22: Exploration of introduction of HBM2 on Quicksilver

not as impacted on both applications. We only observe the predicted upper bound being higher by 0.6 GFLOPS on N1 and no change on TX2. This analysis shows it can be more impactful on performance to increase vector size than increasing the main memory bandwidth for MiniFE.

When we compare the projection on both applications to the performance on the A64FX machine, we predict performance to be higher on N1 and TX2 architecture. The introduction of HBM2 and SVE512 on these machines changed their single-core roofline to be on par with A64FX, and GCC is more efficient when targeting N1 and TX2 architecture than A64FX, causing this higher predicted performance [87].

### 6.3.4 Vector sizes exploration on A64FX with different software stacks

We have seen that GCC is not recommended to obtain performance on a single core of A64FX, and we want to compare it with the use of the Fujitsu Compiler (FCC). Figure 6.25 presents this comparison when changing the vector sizes on LULESH with GCC and FCC compilers. We do not have an interval with both software stacks, meaning the OIs of both binaries are in the compute-bound region of A64FX. However, we observe a different evolution of the predicted value when increasing the SVE vector size. GCC binary gains more performance when increasing vector size when compared to the FCC binary because it has more vector usage. Despite this difference in vectorization, we observe higher performance on

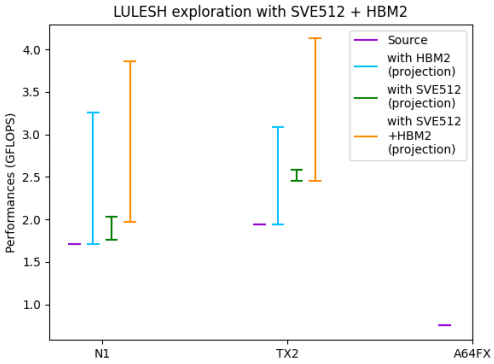


Figure 6.23: Exploration of introduction of HBM2 and SVE512 on LULESH

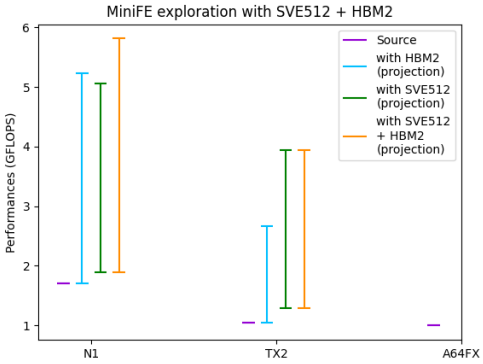


Figure 6.24: Exploration of introduction of HBM2 and SVE512 on MiniFE

FCC with SVE vectors from 128 bits to 512 bits, with the last being the native vector size. FCC is more careful than GCC when vectorizing the application because of its insight into the micro-architecture impact of the A64FX. In contrast, GCC vectorizes a loop without considering it as much. Thus, we have better usage of an A64FX when compiling with the Fujitsu Compiler than with GCC.

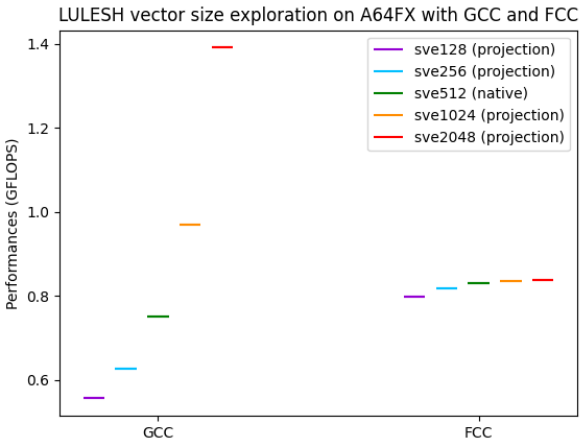


Figure 6.25: Vector sizes exploration with GCC and FCC on A64FX on LULESH

## 6.4 Conclusion

To conclude this chapter, we needed to define a performance projection approach to enable exploring hardware parameters' impact on single-core performance with our workflow. For this objective, we implemented a projection using a single-core roofline representation of performance. This approach also uses application and software characterization to ponder the compute part of the roofline used in projection.

With a pool of 3 Arm cores architectures and three applications, we have concluded that this projection approach allows for accurate performance projection in case of close micro-architectural projection but is too coarse for projection with a significant impact of the micro-architecture differences on performance (such as between Neoverse N1 and Fujitsu A64FX).

Following these validation efforts, we have used this approach to explore various performance parameters such as SVE vector length, HBM2 introduction, the combination of both changes and the impact of the compiler on SVE vector size exploration on three applications with different behaviors. These experiments underline the complexity of interaction between application, hardware, and software stacks, leading to many different optimization patterns adapted to each situation. Even if the single-core performance of applications is vital, nowadays, applications run on a full computing node. This execution on a full node adds complexity and changes the execution behavior of applications. Hence, it would be interesting to extend the projection model to study applications' performance up to a full node and have a better representation of the actual application's execution. This extension is the object of the following chapter.

In the case of single-core performance, the projection made by our model is too coarse to account for the impact of the differences in micro-architecture, especially the instruction pipeline and reordering buffer, on the source and target triplet performance. In the case of constructors, the core design is directed with the use of cycle-accurate simulators [72] as the functioning of the micro-architecture is often too complex to be accurately extracted from a single instruction mix analysis. However, we cannot rely on a full cycle-accurate simulator in our case as the overhead is too important to enable fast design-space exploration.

One way to model the impact of these differences would be to use a representative metric of the application and software interaction with the micro-architecture and ponder the roofline with this metric. This work of Cabezas et al. [35] with a DAG-based (Directly Acyclic Graph) performance analysis and ponderation of rooflines could be implemented to ponder the projection roofline thanks to micro-architectural constraints. However, their work is not thread-compliant and would need further research to be extended to multicore analysis.

# Chapter 7

# Exploration of software and application parameters impact on HPC applications single-node performance

Following the results of the single-core projection approach, the need to extend the approach to multicore application is natural, as it better represents the execution of today's HPC applications. This chapter aims to extend the previous single-core roofline projection approach to multithreaded applications up to a full node. Then, this approach is used, after validation between Graviton2 and Graviton3, to explore parameters around applications and software stack aspects targeting an architecture supposedly close to the European Processor Initiative (EPI). The precise characteristics of this processor are private at the time of the writing.

## 7.1 Roofline projection model extension to multicore

As the previous projection approach is only single-core, the core idea of its extension to multicore is to perform this single-core projection for each application's thread running concurrently on a full computing node. The effects threads have on the performance of each other through cache performance and bandwidth limitation are modelized thanks to a more refined bandwidth analysis of each thread. The analysis workflow is the same as the single core iteration with the addition of a last step where we aggregate each thread's projected performance to determine the multithreaded application's total performance. Consequently, the first step

is the characterization stage, with the characterization of the hardware and its ponderation through application and software stack analysis.

### 7.1.1 Hardware Characterization

First, we define the hardware roofline for each thread running concurrently on a fixed number of cores of the computing node. The roofline for a single core ($\text{roofline}_{\text{ref}}^{\text{core}}$) is not obtained in the same environment as a single core execution. It is obtained by running these benchmarks using the studied number of cores and dividing it by this number of cores ($\text{N}_{\text{core}}$). It usually leads to a lower HPL value than a single core execution, as HPL does not have a linear scaling on our reference nodes. This is presented in eq. (7.1).

$$\text{roofline}_{\text{ref}}^{\text{core}}(\text{OI}) = min\left[\frac{\text{BW}_{\text{STREAM}}^{\text{node}}}{\text{N}_{\text{core}}} \times \text{OI}, \frac{\text{Perf}_{\text{HPL}}^{\text{node}}}{\text{N}_{\text{core}}}\right] \tag{7.1}$$

However, this equation does not accurately represent the differences brought by the applications and software stack through instruction mix and cache usage. Hence, we have chosen to ponder the roofline's compute and memory part, contrary to the single-core approach that only ponders the compute roofline.

### 7.1.2 Roofline ponderation

In the multicore approach, we compute the maximum peak performance of the instruction mix but also the effective bandwidth through cache hits.

Contrary to the single-core approach where we only ponder the peak HPL performance with eq. (6.2), we ponder the memory part of the roofline. We need to refine the bandwidth characterization as the performance of multithreaded applications is even more sensitive to this bandwidth. Moreover, this characterization allows us to reduce the projection interval size as we now only make one projection per memory level.

This bandwidth ponderation is done thanks to cache analysis using the hit ratio of this level and higher levels. Furthermore, as in the Latency-Aware Roofline Model of Denoyelle et al. [46], we consider different sockets in a NUMA topology as another memory level. It is translated to different bandwidths when a core accesses data in the main memory: one when accessing data on its Current Socket (CS) and another when accessing data on the Other Sockets (OS).

Consequently, the memory part of the roofline of this memory level is obtained by averaging its bandwidth and that of higher levels, weighted by their hit ratio. Hence, in eq. (7.2), we show the ponderation of the bandwidth of a core where $\text{BW}^{\text{reference}}$ and $\alpha$ are the reference bandwidth obtained with STREAM and the total hit ratio of that memory level.

$$\mathrm{BW}_{\mathrm{L1}}^{\mathrm{core}} = \frac{\sum_{i=L1}^{i=OS} \alpha_i}{\frac{\alpha_{\mathrm{L1}}}{\mathrm{BW}_{\mathrm{L1}}^{\mathrm{reference}}} + ... + \frac{\alpha_{\mathrm{CS}}}{\mathrm{BW}_{\mathrm{CS}}^{\mathrm{reference}}} + \frac{\alpha_{\mathrm{OS}}}{\mathrm{BW}_{\mathrm{OS}}^{\mathrm{reference}}}} \tag{7.2}$$

With these ponderations, we obtain a ponderated roofline for each memory level. It represents the hardware limitations and the limitations caused by the interactions between software, applications, and hardware. The memory ponderation allows for a more representative description of the bandwidth resulting from the complex interactions between threads and memory topology through memory levels hit ratio.

### 7.1.3 OI characterization

Now that we have defined more refined limits for each thread, we define its OIs. Similarly to the single-core approach, we define one OI per memory level of a two-level cache machine with eq. (7.3) with Bytes moved in L1 ($B_{L1}$) and L2 ($B_{L2}$) in addition to the splitting of the bytes coming from the main memory in Current Socket ($B_{\mathrm{CS}}$) and Other Socket ($B_{\mathrm{OS}}$). If the machine does not have different NUMA nodes, we do not differentiate between bytes in the Current Socket and the one moved in the Other Socket.

$$\begin{aligned}
\mathrm{OI}_{L1} &= \frac{N_{\text{floating point operations}}}{B_{L1} + B_{L2} + B_{\mathrm{CS}} + B_{\mathrm{OS}}} \\
\mathrm{OI}_{L2} &= \frac{N_{\text{floating point operations}}}{B_{L2} + B_{\mathrm{CS}} + B_{\mathrm{OS}}} \\
\mathrm{OI}_{CS} &= \frac{N_{\text{floating point operations}}}{B_{\mathrm{CS}} + B_{\mathrm{OS}}} \\
\mathrm{OI}_{OS} &= \frac{N_{\text{floating point operations}}}{B_{\mathrm{OS}}}
\end{aligned} \tag{7.3}$$

Furthermore, this per-thread OI analysis allows us to characterize the difference in the workload of each thread. Hence, this allows us to deal with the differences in load balancing as each workload is analyzed separately. Moreover, we also characterize the impact they can have on each other through bandwidth alteration. However, one limitation of this analysis is that the number of threads between source and target triplets has to be equal.

### 7.1.4 Performance projection

We have defined OIs and rooflines for each thread. The last step is to analyze the performance of the source triplet and make the projection on the target triplet. It has two stages: a **thread projection** and the **multicore summation**.

### Per-thread projection

The per-thread projection is the same as the single-core approach. We use the architectural efficiency observed for each thread of the source triplet and project it on the target triplet for each OI/roofline couple representing a memory level. The projection for a memory level is defined in eq. (7.4).

$$\text{Perf}^{\text{proj}}_{\text{target}} = \frac{\text{Perf}_{\text{source}} \times \text{roofline}_{\text{target}}(\text{OI}^{\text{target}})}{\text{roofline}_{\text{source}}(\text{OI}^{\text{source}})} \tag{7.4}$$

The multiple OIs defined in eq. (7.3) can result in a projection interval for each thread similar to the single core projection.

### Multicore summation

Next, we sum all these performance intervals ($\text{Perf}^{\text{proj}}_{\text{thread}}$) to obtain the application's performance for all threads running concurrently on the same node ($\text{Perf}^{\text{proj}}_{node}$). It is important to note that the performance gain from increasing the thread count is non-linear and a critical aspect of our approach is its consideration of the effects (such as cache behavior, bandwidth, and load balancing) that induce non-linear scaling. This justification supports the validity of eq. (7.5) as a reliable approximation.

$$\text{Perf}^{\text{proj}}_{node} = \left[ \sum_{i \in threads} min(\text{Perf}^{\text{proj}}_{\text{thread}_i}); \sum_{i \in threads} max(\text{Perf}^{\text{proj}}_{\text{thread}_i}) \right] \tag{7.5}$$

Hence, we have extended the single-core approach to multicore analysis by multiplicating the core analysis to each application thread running on a core. Because of the effect of multicore execution on bandwidth with shared caches and main memory bandwidth limitation, the effective bandwidth characterization of the roofline needed to be refined. After having obtained a projection of the performance of each thread, we sum all the intervals obtained into a single performance interval representing the application performance projection.

## 7.1.5 Implementation

As the needed metrics are the same as before, our implementation did not change. We use online analysis through DynamoRIO and ArmIE for instruction mix analysis, Stream and HPL for hardware characterization, and hardware counters (if available) or simulation for cache metrics analysis.

| Machine | Graviton 2 | Graviton 3 | EPI-like |
|---|---|---|---|
| Performance (GFLOPS) | 970 | 1380 | 1380 |
| Micro-architecture | Neoverse N1 | Neoverse V1 | Neoverse V1 |
| Vector ISA | NEON | SVE 256 | SVE 256 |
| Memory | DDR4 | DDR5 | HBM2 |
| Bandwidth | 180 GB/s | 313 GB/s | 850 GB/s |
| L3 size | 32 MiB | 32 MiB | 128 MiB |

Table 7.1: Characteristics of the Arm architectures used in model validation and design-space exploration.

### 7.1.6 Experimental Environment

The previous single-core approach has difficulties with projection between machines with distant micro-architectures. As we expected this extension to have similar issues, we have restricted the pool of machines to the Arm Neoverse micro-architecture roadmap [5] for validation and exploration.

This choice of focusing on this series is motivated by future HPC hardware using this roadmap with future processors such as the European Processor Initiative (EPI) represented by Sipearl's Rhea 1 (based on Neoverse V1) [90] or the future NVIDIA Grace (based on Neoverse V2) [14]. At the time of the writing, the two generations of AWS Graviton processors are representative of the micro-architecture evolution and the interaction between current applications and the Neoverse series.

To prepare for the arrival of the EPI, we have chosen to model the future hardware machine (EPI-like) with some of the characteristics publicly available of this V1-based processor: the use of both DDR5 and HBM2e and a large System-Level Cache [90]. Because we do not have access to an EPI CPU, the STREAM value for our "EPI-like" source machine is a projection of the HBM2 bandwidth obtained on the only other available Arm HBM node: the Fujitsu A64FX node. Table 7.1 gathers the characteristics and the results of the Arm architectures used in validation and exploration.

## 7.2 Model extension validation

The objective of this validation effort is to see if the results of our approach between two similar computing nodes of the Neoverse roadmap are accurate enough to open for discussion. In this case, we use two Neoverse machines: an AWS Graviton 2

and an AWS Graviton 3 node. We conduct these experiments using GCC 11.2 with *-O3 -ffast-math* flags and the correct *-march* flag to generate NEON for both micro-architectures as we do not make projections with changes of ISA in experiments. These experiments use two applications: LAMMPS DIFFUSE and LULESH and the OpenMP version of NAS Parallel benchmark suite [77].

### 7.2.1 Graviton 2 ↔ Graviton 3 projection

Figures 7.1 and 7.2 presents the obtained interval (in orange, the lower bound of the interval, and green on the upper bound) and compare it with the performance of the source machine (in blue) and target machine (in yellow) for every application.

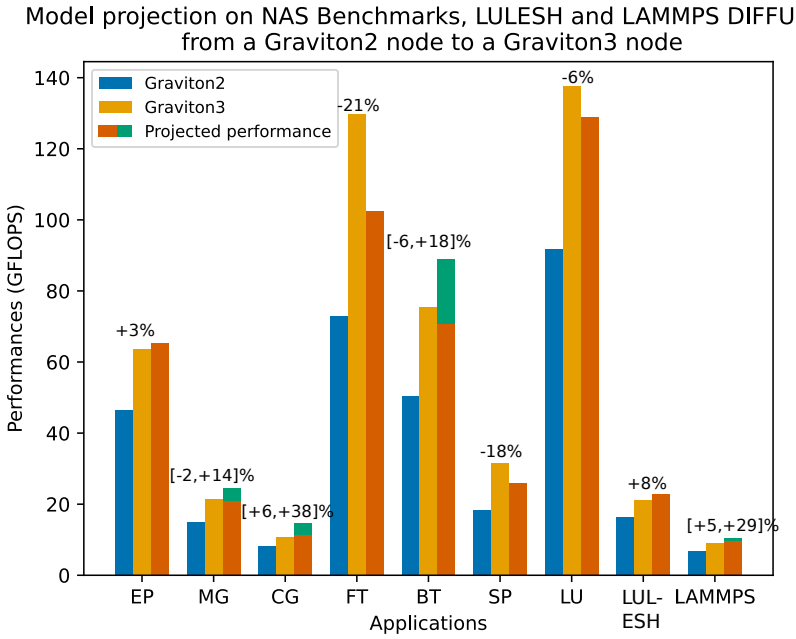


Figure 7.1: Projection intervals compared to actual performance on a Graviton2 and Graviton3 node. Graviton2 and Graviton3 performances are in blue and yellow, with projected performance intervals in green and orange. The number corresponds to the difference between observed and projected performance.

In Figure 7.1, we observe a performance gain on every benchmark when projecting from Graviton2 to Graviton3. Qualitatively, our projection workflow captures the performance gain in all cases.

Quantitatively, on Graviton3, the target performance of MG and BT falls within the projected interval of our method. However, the lower intervals for

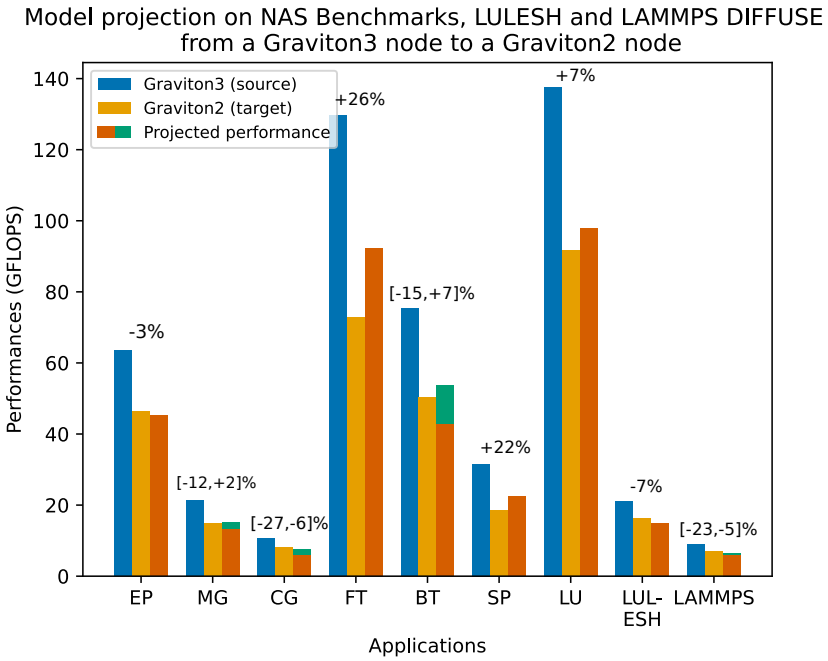


Figure 7.2: Projection intervals compared to actual performance on a Graviton3 and Graviton2 node.

CG and LAMMPS are over-predicted by 6% and 5%, respectively. All other projections do not yield an interval, as there is no difference between the projections of their OI: all kernels are compute-bound on both the source and target machines. Still, the results for the projections closely align with the observed performance of EP (+3%), LU (-6%), and LULESH (+8%).

For FT and SP NAS benchmarks, while our projection correctly predicts a performance gain, we find that the actual performance is underestimated by around 20%. To investigate this discrepancy, we conducted an analysis using Linux *perf* on both machines and observed more stalls due to the front-end on Graviton 2 compared to Graviton 3 (e.g., 14% against less than 1% of total cycles for FT) for both of these benchmarks. Here, the impact of the CPU pipeline and the micro-architecture efficiency of the source machine is projected onto the target machine. Consequently, when the impact of these mechanisms on the application's performance is more significant on one architecture than the other, it may not be fully captured by the model, leading to some observed inaccuracies in the projection.

For the projection from Graviton 3 to Graviton 2 (Figure 7.2), we observe that our workflow correctly predicts the performance loss for every application. Quantitatively, we notice a mirrored behavior: when transitioning from Graviton2

to Graviton3, we over-predict the performance, as seen in the cases of EP kernel (+3%) or LULESH (+8%). When transitioning from Graviton 3 to Graviton 2, we under-predict by similar values (respectively -3% and -7%), as expected. This mirrored behavior is observed for every application tested. Hence, the applications with the highest errors are still FT and SP (for the same reason as above).

### 7.2.2 Comparison with straightforward roofline projection

In Figure 7.3, we compare our approach with the straightforward projection of [74] for the LULESH case. In this figure, we depict the obtained roofline used by both methods: without ponderation (only the raw STREAM and HPL values) in plain line and with our ponderation in dotted line. Our ponderated roofline used in conjunction with the $OI_L1$ has a bandwidtth close to the measured STREAM L1 bandwidth because LULESH has a high hit ratio at this level (95%). In this case, we are in the compute-bound zone for the ponderated roofline. Moreover, we observe no difference between OIs, and according to a CARM analysis, the DRAM Bandwidth limits the performance as it is the closest to the observed performance. With the "straightforward" projection based on the bandwidth gain between Graviton2 and Graviton3, we obtain a projected performance of 24.82 GFLOPS, which is 17.5% higher than the actual observed performance of 21.13 GFLOPS. However, when we use the pondered roofline approach, the projected performance is 22.81 GLOPS, which is only 8.8% higher than the actual observed performance. Consequently, the roofline ponderation of our method is twice as much more accurate than a "straightforward" projection.

To conclude these validation efforts, the projection between two close machines of the Neoverse roadmap is accurate enough to open for discussion. However, as expected, our roofline projection approach accuracy is limited by the differences of the impact of the Out-Of-Order ressources between source and target machines. In the following experiments of Sections 7.3 and 7.4.1, as both the source and target triplet of interest rely on a Neoverse V1 architecture, the differences in micro-architecture behavior is minimal and should not impact the accuracy of the projection.

## 7.3 Application parameters exploration

The objective of this experiment is to observe how a change in the source code would impact the performance of our EPI-like machine. We have used the LAMMPS DIFFUSE benchmark to accomplish this experiment, available in this GitHub repository [15]. This benchmark allows the use of different numerical schemes to compute the same particle diffusion problem. There are two inputs avail-

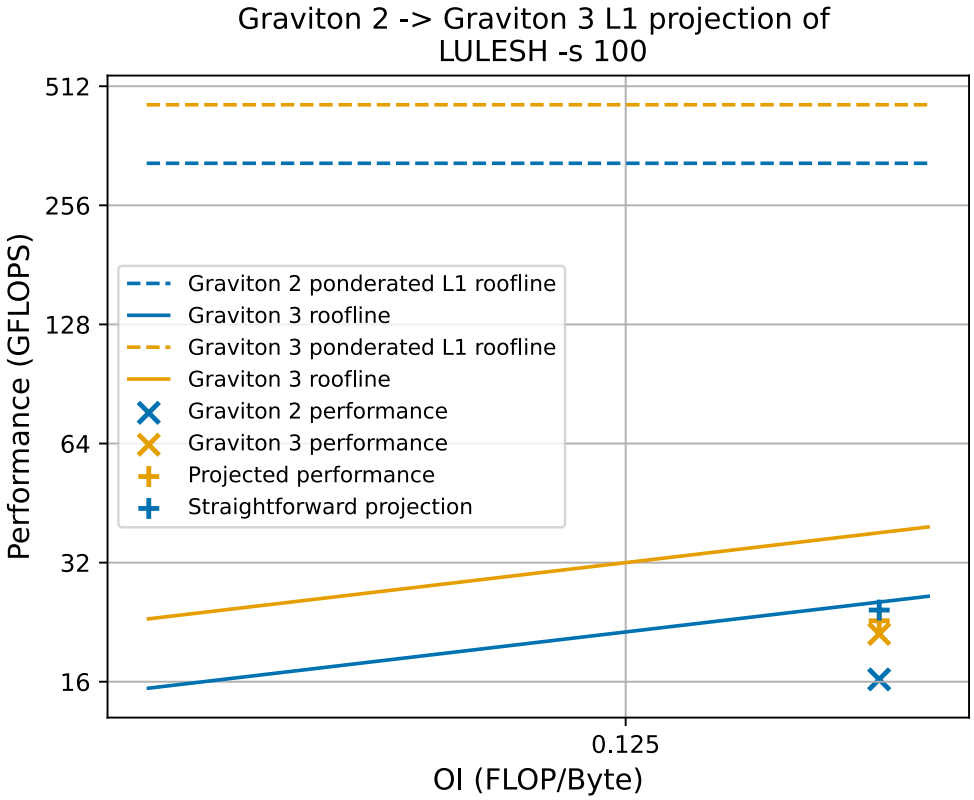


Figure 7.3: Obtained Cache-Aware Roofline model of LULESH on Graviton2 and Graviton3 with the addition of L1 pondered roofline and projection. Rooflines of L1, L2 and L3 memory levels have been hidden for clarity purposes

able: velocity-auto-correlation function (VACF) and mean-squared displacement (MSD).

The results of these two projections can be seen in the following Figure 7.4. A first observation is that the VACF method attains a lower performance (9.63 GFLOPS) than MSD (11.43 GFLOPS) on the source machine, which translates to the projected performance on the EPI-like node.

Moreover, there is no performance gain in the projection from Graviton 3 towards the EPI-like node. In this study, the behavior change between both methods stays in the compute-bound region for both machines. Hence, as the differences between our source and target hardware only impact the memory, we do not project any performance change. In order to gain performance on an application represented by both methods of this benchmark, the focus should be on the peak compute performance of the application.

Another aspect that can impact the performance we want to test on another

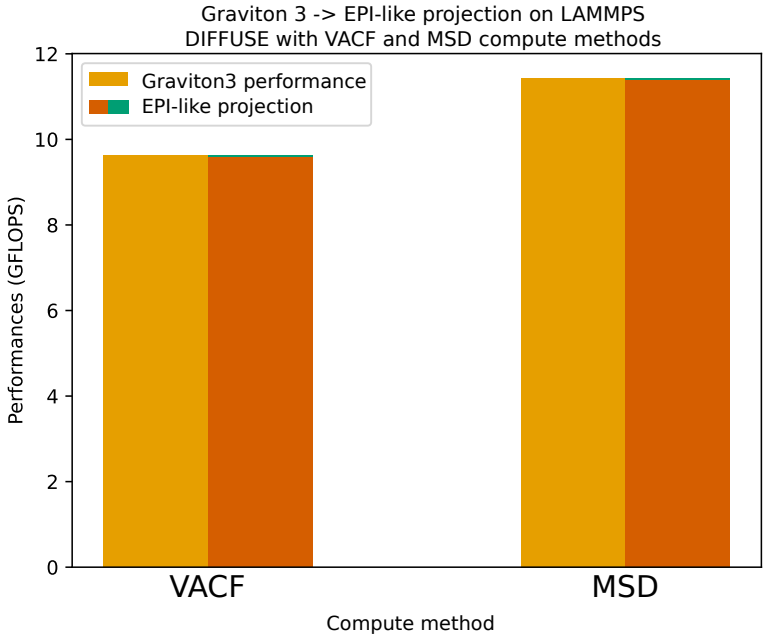


Figure 7.4: Projections of two compute methods of LAMMPS DIFFUSE small benchmark from Graviton 3 to our EPI-like machine

application is the optimizations made by the compiler when generating different ISAs, such as SVE and NEON.

## 7.4 Software parameters exploration on target node architecture

With an ISA as recent as SVE, looking into the impact of the compiler's choice when generating SVE is interesting for applications and software stacks to prepare for a future architecture like the European Processor Initiative (EPI).

### 7.4.1 Compilers and ISA exploration

This experiment aims to look into which ISA and compilers could have the best usage of the faster memory bandwidth and bigger L3 cache of the EPI compared to a Graviton 3 node. Hence, in this experiment, we compare two compilers of the Arm environment, g++ (11.3) and armclang (22.1). Each compiler generates two binaries: one using NEON and one using SVE.

Results of the comparison between projected performance on the EPI-like machine and source performance of a Graviton 3 node are visible in Figure 7.5 with the difference between the measured $OI_L1$ and its corresponding roofline's $OI_Ridge$, representing the inflexion point between compute-bound and memory-bound region. At first glance, despite having higher performance on a Graviton3 node, there would be no performance gain with any ISA using armclang as a compiler as their performance is already similar on the source machine. In this case, the differences between NEON and SVE vectorization are minimal as the vectorization rate and use of vector units are similar. Moreover, using 256-bit vectors instead of 128 bits does not also lead to a better performance on the source triplet. The projected performance is the same because the lowest OI observed in our model (the OI of the L1 level) is already in the compute-bound region for a Graviton 3 node ($OI_{L1} > OI_{Ridge}$). In this case, an increase in memory bandwidth would not lead to a performance gain because the source bandwidth is already fast enough to sustain the FPUs with the instruction mix of LULESH obtained with armclang.

On the contrary, the generation of SVE with GCC does bring a behavior change to our source triplet. The performance of the SVE binary compiled with g++ is lower on Graviton 3 than with NEON (20.07 GFLOPS for NEON VS 19.56 for SVE). When generating SVE, the GNU compiler makes optimization choices that result in an observed $OI_{L1}$ divided by 2 (0.15 against 0.07 FLOP/Bytes). This much lower OI may explain the lower performance of SVE on our source machine.

However, this phenomenon is also the cause of the performance gain when projecting performance toward the EPI-like node. As we see, the observed OI of the L1 memory level is under the OI of the ridge point of its roofline (the limit between the Compute and memory bound zones) on the source machine. Hence, the performance of this binary is the only one affected by this memory bandwidth increase brought by HBM2, explaining the maximum performance gain of 50%.

To conclude this experiment, even if armclang is the compiler that brings the most performance on a Graviton 3 node with either SVE or NEON ISA, the behavior change brought by the Gnu compiler when introducing SVE causes the performance of the binary to be the only one affected by the change between Graviton3 and our EPI-like machine.

If the performance of this binary on the source machine would be much lower because of the OI change, it would not necessarily be a total performance gain on the EPI-like. However, in our case, this OI reduction with performance conservation on the source machine allows for the GCC SVE binary to possibly attain the highest performance on an EPI-like machine.

As this behavior is not intuitive, we want to reproduce this on an architecture that would mimic this possible bandwidth increase. This reproduction is presented in the next Section.

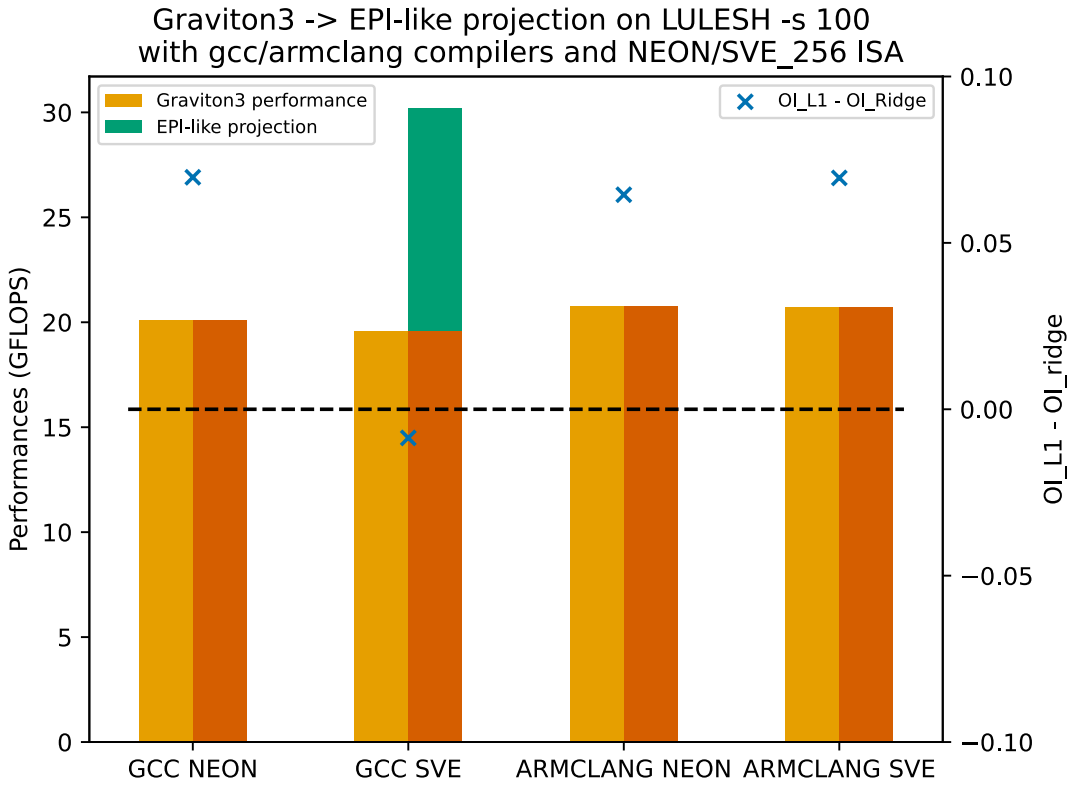


Figure 7.5: Observed performance, projections, and OIs of LULESH with different compilers and ISA on two node architectures: Graviton3 and EPI-like. The $OI_{L1} - OI_{Ridge}$ value is represented with blue crosses.

## 7.5 Behavior reproduction with synthetic kernels

Graviton3 and EPI-like architectures exhibit minor micro-architecture differences, mainly in main-memory bandwidth and L3 cache size. To emulate this difference, we use the Intel Knights Landing (KNL) architecture with Quadrant Flat configuration, allowing us to choose data allocation on either DRAM (bigger but slower memory) or MCDRAM (smaller but faster memory). This particularity enables us to reproduce the bandwidth change while maintaining the same values for other architecture parameters. The benchmark values obtained on KNL are presented in Table 7.2.

We designed two synthetic kernels, detailed in Listing 7.1, to replicate the behavior of LULESH with GCC SVE/NEON. This design allowed us to control the experiment finely. Both kernels perform the same number of floating-point operations on a private array and a reduction on a shared array. *Kernel_1* uses a

| HPL | 1300 GFLOPS |
|---|---|
| DRAM STREAM | 83 GB/s |
| DRAM Ridge | 15.66 FLOP/Bytes |
| MCDRAM STREAM | 463 GB/s |
| MCDRAM Ridge | 2.80 FLOP/Bytes |

Table 7.2: Benchmarks values on KNL.

single private array per thread, while *kernel_2* employs two private arrays, with half of the computations per array. As a result, the OI of *kernel_2* is lower than that of *kernel_1*. All the OI and performance metrics are obtained using Intel Advisor Suite. Intel Advisor also allowed us to verify that both kernels are vectorized and can make efficient use of the faster MCDRAM bandwidth.

```
void kernel_1(int N,int cursor,double *a) {
 #pragma omp parallel
 {
  seed = omp_get_thread_num();
  double *a_private = malloc(N * sizeof(double));
  a_private = lots_of_computation(N,seed,cursor,a_private);
  #pragma omp critical
   for (i=0 ; i<N ;i++ )
    a[i] += a_private[i];
 }
}
void kernel_2(int N,int cursor,double *a) {
 #pragma omp parallel
 {
  seed = omp_get_thread_num();
  double *a_private = malloc(N * sizeof(double));
  double *b_private = malloc(N * sizeof(double));
  a_private = half_of_computation(N,seed,cursor,a_private);
  b_private = other_half_of_computation(N,seed,cursor,b_private);
  #pragma omp critical
   for (i=0 ; i<N ;i++ )
    a[i] += a_private[i] + b_private[i];
 }
}
```

Listing 7.1: Source code of the two kernels of the synthetic benchmark used to reproduce the behavior observed with GCC on LULESH. kernel_1 has a higher operational intensity than kernel_2.

By changing the total number of computations, we have different versions of both kernels: one is compute-bound according to DRAM and MCDRAM ridge points (Kernel_CB), the other is in between these ridge points with an OI of 4 FLOP/Bytes for *kernel_1* and 2 FLOP/Bytes for *kernel_2* (Kernel_MB).

| Kernel | DRAM | MCDRAM |
|---|---|---|
| Kernel_CB_1 | 2.20 | 2.20 |
| Kernel_CB_2 | 2.47 | 2.47 |
| Kernel_MB_1 | 1.91 | 2.05 |
| Kernel_MB_2 | 1.90 | 2.19 |

Table 7.3: Kernels performance in GFLOPS on KNL running on DRAM and MCDRAM.

Table 7.3 presents the results obtained with these kernels when allocating memory on DRAM and MCDRAM. As expected from the compute-bound kernels, increasing the bandwidth leads to no performance gain. However, we observe higher performance for *kernel_CB_2* compared to *kernel_CB_1*.

When examining the memory-bound kernels, we observe that they exhibit similar performance on DRAM despite *kernel_MB_2* being more memory-bound with an OI of 2 FLOP/Bytes, compared to *kernel_MB_1*. However, the performance gain on MCDRAM is significantly higher for *kernel_MB_2* than for *kernel_MB_1*. This discrepancy arises because *kernel_MB_2* can use the bandwidth increase more efficiently than *kernel_MB_1*. Indeed, despite having a lower OI, *kernel_MB_2* achieves similar or even better performance than *kernel_MB_1*. The reason lies in splitting the computation into two private arrays, resulting in better machine usage efficiency. Therefore, as the bandwidth is fast enough to sustain this decrease in OI, selecting the second implementation of this kernel leads to improved performance.

In this scenario, the behavior change is introduced by altering the data structures in the source code, while in the compiler and ISA exploration of previous Section, g++ causes this change. It underscores the software stack's significant impact in efficiently utilizing crucial architecture features of future hardware while highlighting how our performance projection model can aid in assessing his potential improvement.

## 7.6 Conclusion

As a conclusion of this chapter, the extension of the previous single-core projection to multicore relies on an analysis of the performance of each thread with bandwidth ponderation according to the impact they have on each other with the memory hierarchy. This insight into the actual effective bandwidth of the application running on a compute node is needed as the multicore scaling of the application is easily hindered by bandwidth scaling.

After validating the projected performance obtained with this approach be-

tween a Graviton2 and a Graviton3 node on various applications such as NAS benchmarks, LULESH, or a small LAMMPS benchmark, we have chosen to focus the study around application and software aspect of performance on a hypothetical machine that is a Graviton 3 with boosted memory bandwidth and L3 size. The characteristics of this machine are supposedly close to an EPI node with the public information available at the time of the writing. This study aims to look into ways to make more efficient use of the memory gain between a Graviton3 and the EPI-like node, thanks to application and software changes.

When testing two different computing methods of the same problem with LAMMPS DIFFUSE, there was no projected performance gain as both methods are already compute-bound on a source machine. However, when exploring the impact of SVE and NEON generation with GCC and armclang compilers, we observed a counterintuitive behavior that involved a decrease in OI while conserving performance in LULESH, leading to a projected performance gain on the target machine. Using a synthetic benchmark, we verified and replicated this behavior in a similar environment offered by the KNL processor. These studies showed the possibilities of using this approach to explore and optimize the performance of applications on future machines thanks to application and software stack optimization.

In our experiments, we have explored two ways of adapting application and software to get performance from a fixed target architecture. However, there are many other possibilities offered by these aspects that could be explored. On the one hand, for application exploration, one interesting study would be to explore the possibilities offered by a HPC development framework like Arcane [55]. It would open more possibilities to explore around the different behavior when relying on various numerical schemes to compute the same problem. On the other hand, for software focused exploration, the possibilities offered by looking around different linear algebria libraries would also be an interesting study. Finally, looking around the use of the hardware topology by different MPI and OpenMP implementations or environment is also an important factor of intra-node performance. However, as of now, our implementation relying on *drcachesim* for cache metrics and ArmIE for SVE metrics analysis in not MPI-compliant and this would require more development.

# Conclusion and Perspectives

HPC supercomputer architecture is becoming more and more complex, hindering the applications' capabilities to attain performance. It also increases the need to update HPC applications according to the machine's evolution. The possibility of designing future machines adapted to the applications' needs through codesign initiatives appears as a solution to facilitate optimization efforts and attain higher performance. Furthermore, the recent Arm HPC environment is attractive for such codesign initiatives, opening up many design liberties and possibilities. However, a performance prediction model is mandatory to drive codesign initiatives.

## Summary of contributions

This manuscript proposes a performance projection workflow adapted to our definition of codesign in the Arm HPC environment. Then, we implement this workflow with a Roofline representation to explore around hardware parameters impact on single-core performance of applications. Finally, we extend this workflow to multi-core execution of applications and study the possible optimizations patterns around software stack and application aspects of performance.

### Performance projection workflow setup

We designed this three-stage workflow (Characterization, Analysis, and Projection) to rapidly explore the design-space around a source application/software stack/hardware triplet by characterizing the impact of each of these aspects on an HPC applications' performance. The last stage is a projection on a target triplet according to the differences between the parameters of each aspect. This presentation of the workflow aims to be generic as the metrics needed, the benchmark models and the analysis model used in projection can be adapted to the specific parameters of interest to explore. The two following contributions present the implementation of this workflow with the use of a Roofline representation for performance analysis and projection.

## Single-core implementation for hardware exploration

Hence, with a single-core projection relying on a Roofline representation of performance ponderated by application and software characterization, we implemented this workflow that can leads to a projected performance interval. Then, we apply it on a pool of three Arm core architectures: Marvell ThunderX2, Fujitsu A64FX, and Neoverse N1. We conduct a first validation study between our source core architectures, leading to valid results between close architectures (Neoverse N1 and Marvell ThunderX2). However, in the case of projection with the A64FX core, the projection leads to inaccurate results as the differences in the Out-Of-Order ressources is not accounted in our projection and the performance on the A64FX are highly impacted by this particularity. Finally, our last single-core contribution is a study on a panel of 3 proxy applications (LULESH, MiniFE, and Quicksilver) around many hardware parameters of interest in the Arm HPC environment like the SVE vector length, the memory type, or their combination. The results of this study underline the complexity of interaction between applications, hardware, and software stack, leading to many different optimization patterns adapted to each situation.

## Multi-core extension for software stack and application exploration

Finally, the last contribution is a natural extension of the single-core projection approach to a multicore environment. This extension uses a finer bandwidth characterization of the workload of each core executing the application concurrently on a compute node. The validation study of this extension is conducted between two processor separated by one generation: the AWS Graviton 2 and Graviton 3. With the widest gap between measured and projected performance of 20% for two NAS kernels, the model's prediction opens the study around optimization patterns around software stack and applications aspects toward a node supposedly close to the European Processor Initiative (EPI). During this study, one optimization pattern, brought by the compiler targeting SVE, is a reduction of the Operational Intensity with a conservation performance on the source machine. Hence, it is the only binary that benefits from the memory boost of the hypothetical EPI-like machine. As we did not have access to this machine at the time, we reproduced and observed this behavior in similar environment offered by the Intel KNL.

# Perspectives

The conclusion of the experiments and the contributions of this thesis further underlines the need to take feedback on the choice of every actor involved in the usage and conception of HPC application, software stack, and hardware. This discussion, allowed by performance prediction, should drive the conception of future machines at every scale. Following this work, I will conduct a first use of our model during a 3-month internship for the Feasibility Study of the Fugaku Next program of the Riken Center of Computational Science (R-CCS). This is the occasion to use the workflow conceptualized during this thesis in an actual HPC codesign environment.

## Get over the fixed core number limitation

One short term perspective would be to extend this implementation to achieve projection between hardware with a different number of cores. The efficiency ratio, projected between our source and target triplet in our multicore model, could be extrapolated thanks to scaling studies and regression on the source triplet. Such a scaling study would also need to consider the impact of topology and thread placement on the scalability of applications. And, in our bandwidth characterization, we consider the NUMA interaction on bandwidth with the splitting of Bytes coming from the Current Socket and the one moved in the Other Sockets. However, as we carried out our multicore experiments on monolithic or non-NUMA CPUs, this characterization was not used in our experiments. Hence, the only metric needed for accurate projection between machines with a different core number and topology would be the scaling study of the ratio on a source machine and its projection towards a machine with a different topology that could impact this ratio.

## Extension to heterogenous nodes

Other long term perspectives is the characterization of heterogenous nodes and multi-nodes execution of applications. The first is an implementation of our workflow to explore the design-space around heterogeneous nodes relying on accelerators such as GPU or FPGA. A CEA internship conducted by Van Lanker L. and supervised by Taboada H. and Brunet E. has already implemented and adapted our roofline projection workflow to GPUs with the first encouraging results. The following Figure 7.6 presents a comparison between the projection results and the measured performance obtained on the projection between an NVIDIA V100 to an NVIDIA H100 GPU on the Single-GPU UVMBench suite. This implementation is currently limited to single-GPU kernels but the objective of a future PhD thesis is to extend this model to multi-GPU and CPU-GPU kernels.


Figure 7.6: N100 → H100 projection results on UVMBench benchmark suite, courtesy of L. Van Lanker

## Extension to multi-node execution

Then, the next step would be to extend the roofline projection presented in this work to multiple nodes. This could be achieved in two ways. The first way consider the network between them as an additional memory level. However, the impact of communications and synchronization would also need to be considered as an intra-node performance change would also lead to a change in the applications' synchronization timings. This is also valid in multithreaded applications but it does not impact the performance of the OpenMP applications we studied. One solution could be to trace communications and synchronizations, project the performance on each section of the application delimited by these actions, and generate a DAG of the execution of this application with the sections as nodes and synchronizations as edges. The study of the critical path of this DAG with the obtained projected performance of each section would allow us to conclude the global application's performance. However, it would require further development efforts for our DynamoRIO implementation to automatically trace synchronizations and

define the execution sections.

The second way of introducing multi-core analysis in codesign exploration would be to use our model to extend the CPU performance analysis of network simulators such as Simgrid [38] or Kronos [9]. As of today, these network simulation tools do not model the computing time when simulating a new machine. They either rely on the performance of the Host machine, for Simgrid, or a fixed number of cycles for Kronos. Introducing our model into their workflow would lead to more accurate computing time estimation but also on the synchronization's timings.

### Focus on power consumption

Another issue in today's HPC is the power consumption of an application's execution on a machine. Our work did not consider this parameter in the exploration and it is not our main focus. However, the objective is to have a machine tailored to the applications and software stack needs. In this case, better machine usage efficiency means more performance but also less power losses due to inefficiencies. Hence, the problems of performance and power consumption are not orthogonal.

Nevertheless, adapting our general methodology to power consumption is straightforward as we only need to shift the focus from performance to power consumption. Implementation-wise, the Roofline Model we used for analysis and projection does not consider power consumption. However, there are roofline representations that focus on the power consumption of application [62, 39]. Hence, relying on a roofline representation focused on power consumption could be a solution to shift the focus of codesign studies toward power consumption.

## Final word

The conception of tomorrow's machines in a codesign environment is one solution to simplify the optimization efforts of HPC applications on these more and more complex architectures. It is also a way to attain higher performance thanks to machines tailored for their future workloads. However, because application developers, software engineers, and hardware constructors have different domains of expertise, this discussion between them is complicated but mandatory to enable codesign initiatives.

We hope our developed methodology and approach will encourage these initiatives and drive them to design future machines and software stacks adapted to the HPC applications' needs. As our current implementation only focuses on CPU node exploration, we have seen in these perspectives that the workflow could be adapted to many architectures and fo cus of exploration. It can also open the way

to codesign not only in the HPC world but also in many other domains such as cloud computing, or data servers!

# Bibliography


[1] Arm, ARM Instruction Emulator. https://developer.arm.com/tools-and-software/server-and-hpc/compile/arm-instruction-emulator.

[2] Arm, Emulating SVE on existing Armv8-A hardware using DynamoRIO and ArmIE. https://community.arm.com/arm-community-blogs/b/high-performance-computing-blog/posts/emulating-sve-on-armv8-using-dynamorio-and-armie.

[3] Arm, Introducing NEON. https://developer.arm.com/documentation/dht0002/a/Introducing-NEON.

[4] Arm Neoverse V1 micro-architecture reference manual. https://developer.arm.com/documentation/101427/latest/.

[5] Arm, Redefining the global computing infrastructure with next-generation Arm Neoverse platforms. https://www.arm.com/company/news/2022/09/redefining-the-global-computing-infrastructure-with-next-generation-arm-neoverse

[6] Arm, the Aarch64 ISA documentation. https://developer.arm.com/documentation/ddi0596/2021-12.

[7] Green 500. https://www.top500.org/lists/green500/.

[8] Intel VTune. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.

[9] Kronos: Hybrid Discrete Event Simulations. https://www.anl.gov/mcs/kronos-hybrid-discrete-event-simulations.

[10] LULESH GitHub repository. https://github.com/LLNL/LULESH.

[11] Mantevo Project, MiniFE summary v2.0. https://asc.llnl.gov/sites/asc/files/2020-06/MiniFE_Summary_v2.0.pdf.

[12] MiniFE GitHub repository. https://github.com/Mantevo/miniFE.

[13] NVIDIA NSight. https://developer.nvidia.com/nsight-systems.

[14] NVIDIA, NVIDIA Unveils Next-Generation GH200 Grace Hopper Superchip Platform for Era of Accelerated Computing and Generative AI. https://nvidianews.nvidia.com/news/gh200-grace-hopper-superchip-with-hbm3e-memory.

[15] Online lammps repository hosted on github.com, https://github.com/lammps/lammps. https://github.com/lammps/lammps.

[16] Quicksilver GitHub repository. https://github.com/LLNL/Quicksilver.

[17] R-CCS, About Fugaku. https://www.r-ccs.riken.jp/en/fugaku/about/.

[18] The CORAL Benchmark suite. https://asc.llnl.gov/coral-2-benchmarks.

[19] Top500, Statistics, Performance Development. https://www.top500.org/statistics/perfdevel/.

[20] Wikipedia, Instruction pipelining. https://en.wikipedia.org/wiki/Instruction_pipelining.

[21] Wikipedia, Von Neumann architecture. https://en.wikipedia.org/wiki/Von_Neumann_architecture.

[22] The design and performance of batched blas on modern high-performance computing systems. *Procedia Computer Science*, 108:495–504, 2017. International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland.

[23] Alexander Aiken, Utpal Banerjee, Arun Kejariwal, and Alexandru Nicolau. Instruction level parallelism. In *Springer US*, 2016.

[24] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery.

[25] Arm. Arm, Arm Performance Libraries. https://developer.arm.com/documentation/ddi0596/2021-12.

[26] Richard Frederick Barrett and Michael Allen Heroux. The mantevo projectmini-applications: Vehicles for co-design. 2013.

[27] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, aug 2011.

[28] Pradip Bose. *Power Wall*, pages 1593–1608. Springer US, Boston, MA, 2011.

[29] Patrick Brantley, Shawn Dawson, Scott McKinley, Matt O'Brien, Doug Peters, Mike Pozulp, Greg Becker, Kathryn Mohror, and Adam Moody. Llnl mercury project trinity open science final report. 4 2016.

[30] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In IEEE, editor, *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa, Italy, February 2010.

[31] François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier, and Raymond Namyst. ForestGOMP: an efficient OpenMP environment for NUMA architectures. *International Journal of Parallel Programming*, 2010.

[32] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '03, 2003.

[33] Derek L. Bruening. Efficient, transparent, and comprehensive runtime code manipulation. 01 2004.

[34] Anastasiia Butko, Rafael Garibotti, Luciano Ost, Vianney Lapotre, Abdoulaye Gamatie, Gilles Sassatelli, and Chris Adeniyi-Jones. A trace-driven approach for fast and accurate simulation of manycore architectures. In *The 20th Asia and South Pacific Design Automation Conference*, pages 707–712, 2015.

[35] Victoria Caparrós Cabezas and Markus Püschel. Extending the roofline model: Bottleneck analysis with microarchitectural constraints. In *2014*

*IEEE International Symposium on Workload Characterization (IISWC)*, pages 222–231, 2014.

[36] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 52:1–52:12, November 2011.

[37] Carlos Carvalho. The gap between processor and memory speeds. 01 2002.

[38] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, June 2014.

[39] Jee Whan Choi, Daniel Bedard, Robert Fowler, and Richard Vuduc. A roofline model of energy. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 661–672. IEEE, 2013.

[40] Bruno da Silva, An Braeken, Erik H. D'Hollander, and Abdellah Touhafi. Performance modeling for fpgas: Extending the roofline model with high-level synthesis tools. *Int. J. Reconfig. Comput.*, 2013, jan 2013.

[41] Anthony Danalis and Heike Jagode. *Performance Application Programming Interface*. Sun, Baruah and Kaeli, 2022-12 2022.

[42] Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. Cpu db: Recording microprocessor history. 2012.

[43] J. Davis, Gihan Mudalige, S. Hammond, Andy Herdman, I. Miller, and Stephen Jarvis. Predictive analysis of a hydrodynamics application on large-scale cmp clusters. *Computer Science - Research and Development*, 26, 2011.

[44] Arnaldo Carvalho de Melo and Red Hat. The new linux perf tools. 2010.

[45] Sander De Pestel, Sam Van den Steen, Shoaib Akram, and Lieven Eeckhout. Rppm: Rapid performance prediction of multithreaded applications on multicore hardware. *IEEE Computer Architecture Letters*, 17(2):183–186, 2018.

[46] Nicolas Denoyelle, Brice Goglin, Aleksandar Ilic, Leonel Sousa, and Emmanuel Jeannot. Modeling large compute nodes with heterogeneous memories with cache-aware roofline model. 01 2017.

[47] Kiril Dichev and Alexey Lastovetsky. Optimization of collective communication for heterogeneous hpc platforms. *High-Performance Computing on Complex Environments*, pages 95–114, 2014.

[48] Nan Ding and Samuel Williams. An instruction roofline model for gpus. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 7–18, 2019.

[49] Jens Domke. A64fx – your compiler you must decide!, 2021.

[50] Ulrich Drepper. What every programmer should know about memory. 2007.

[51] Emma and Davidson. Characterization of branch and data dependencies in programs for evaluating pipeline performance. *IEEE Transactions on Computers*, C-36(7):859–875, 1987.

[52] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Trans. Comput. Syst.*, 27(2), may 2009.

[53] L. Fedeli, A. Huebl, F. Boillod-Cerneux, T. Clark, K. Gott, C. Hillairet, S. Jaure, A. Leblanc, R. Lehe, A. Myers, C. Piechurski, M. Sato, N. Zaim, W. Zhang, J. Vay, and H. Vincenti. Pushing the frontier in the design of laser-based electron accelerators with groundbreaking mesh-refined particle-in-cell simulations on exascale-class supercomputers. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Los Alamitos, CA, USA, nov 2022. IEEE Computer Society.

[54] Brice Goglin and Stéphanie Moreaud. Dodging Non-Uniform I/O Access in Hierarchical Collective Operations for Multicore Clusters. In IEEE, editor, *CASS 2011: The 1st Workshop on Communication Architecture for Scalable Systems, held in conjunction with IPDPS 2011*, page 7p, Anchorage, United States, May 2011.

[55] Gilles Grospellier and Benoit Lelandais. The arcane development framework. In *Proceedings of the 8th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, POOSC '09, New York, NY, USA, 2009. Association for Computing Machinery.

[56] J.L. Hein. *Discrete Mathematics*. Discrete Mathematics and Logic Series. Jones and Bartlett Publishers, 2003.

[57] Wim Heirman, Souradip Sarkar, Trevor E. Carlson, Ibrahim Hur, and Lieven Eeckhout. Power-aware multi-core simulation for early design stage hardware/software co-optimization. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2012.

[58] R. D. Hornung, J. A. Keasler, and M. B. Gokhale. Hydrodynamics challenge problem. 6 2011.

[59] Khaled Ibrahim, Samuel Williams, and Leonid Oliker. Roofline scaling trajectories: A method for parallel application and architectural performance analysis. pages 350–358, 07 2018.

[60] ICPC. ACM International Collegiate Programming Contest. https://icpc.global/.

[61] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. Cache-aware roofline model: Upgrading the loft. *IEEE Computer Architecture Letters*, 2014.

[62] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. Beyond the roofline: Cache-aware power and energy-efficiency modeling for multi-cores. *IEEE Transactions on Computers*, 66(1):52–58, 2017.

[63] Intel. Integrated roofline model with intel advisor.

[64] Engin Ipek, Bronis R. de Supinski, Martin Schulz, and Sally A. McKee. An approach to performance prediction for parallel applications. 2005.

[65] Radhika Jagtap, Stephan Diestelhorst, Andreas Hansson, Matthias Jung, and Norbert When. Exploring system performance using elastic traces: Fast, accurate and portable. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 96–105, 2016.

[66] Rik Jongerius, Andreea Anghel, Gero Dittmann, Giovanni Mariani, Erik Vermij, and Henk Corporaal. Analytic multi-core processor model for fast design-space exploration. *IEEE Transactions on Computers*, 2018.

[67] Rik Jongerius, Giovanni Mariani, Andreea Anghel, Gero Dittmann, Erik Vermij, and Henk Corporaal. Analytic processor model for fast design-space exploration. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*, 2015.

[68] P.J. Joseph, Kapil Vaswani, and M.J. Thazhuthaveetil. Construction and use of linear regression models for processor performance analysis. In *The*

*Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, pages 99–108, 2006.

[69] T.S. Karkhanis and J.E. Smith. A first-order superscalar processor model. In *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, pages 338–349, 2004.

[70] Darren Kerbyson, Henry Alme, Adolfy Hoisie, Fabrizio Petrini, Harvey Wasserman, and Michael Gittings. Predictive performance and scalability modeling of a large-scale application. page 37, 11 2001.

[71] Donald E. Knuth and Francis R. Stevenson. Optimal measurement points for program frequency counts. *BIT*, 13(3):313–322, sep 1973.

[72] Yuetsu Kodama, Tetsuya Odajima, Akira Asato, and Mitsuhisa Sato. Evaluation of the riken post-k processor simulator. 04 2019.

[73] Dirk P. Kroese, Tim Brereton, Thomas Taimre, and Zdravko I. Botev. Why the monte carlo method is so important today. *WIREs Computational Statistics*, 6(6):386–392, 2014.

[74] Jae Hyuk Kwack, Galen Arnold, Celso Mendes, and Gregory Bauer. Roofline analysis with cray performance analysis tools (craypat) and roofline-based performance projections for a future architecture. *Concurrency and Computation: Practice and Experience*, 2018.

[75] Steven H. Langer, Ian Karlin, and Michael M. Marinak. Performance characteristics of hydra – a multi-physics simulation code from llnl. In Michel Daydé, Osni Marques, and Kengo Nakajima, editors, *High Performance Computing for Computational Science – VECPAR 2014*, pages 173–181, Cham, 2015. Springer International Publishing.

[76] Benjamin C. Lee and David M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, page 185–194, New York, NY, USA, 2006. Association for Computing Machinery.

[77] Júnior Löff, Dalvan Griebler, Gabriele Mencagli, Gabriell Araujo, Massimo Torquati, Marco Danelutto, and Luiz Gustavo Fernandes. The nas parallel benchmarks for evaluating c++ parallel programming frameworks on shared-memory architectures. *Future Generation Computer Systems*, 125:743–757, 2021.

[78] Diogo Marques, Aleksandar Ilic, Zakhar A. Matveev, and Leonel Sousa. Application-driven cache-aware roofline model. *Future Generation Computer Systems*, 2020.

[79] John McCalpin. Memory bandwidth and machine balance in high performance computers. *IEEE Technical Committee on Computer Architecture Newsletter*, pages 19–25, 12 1995.

[80] John McCalpin. Memory bandwidth and machine balance in high performance computers. *IEEE Technical Committee on Computer Architecture Newsletter*, 1995.

[81] Sally McKee. Reflections on the memory wall. 04 2004.

[82] Charles R. Noble, Andrew T. Anderson, Nathan R. Barton, Jamie A. Bramwell, Arlie Capps, Michael H. Chang, Jin J. Chou, David M. Dawson, Emily R. Diana, Timothy A. Dunn, Douglas R. Faux, Aaron C. Fisher, Patrick T. Greene, Ines Heinz, Yuliya Kanarska, Saad A. Khairallah, Benjamin T. Liu, Jon D. Margraf, Albert L. Nichols, Robert N. Nourgaliev, Michael A. Puso, James F. Reus, Peter B. Robinson, Alek I. Shestakov, Jerome M. Solberg, Daniel Taller, Paul H. Tsuji, Christopher A. White, and Jeremy L. White. Ale3d: An arbitrary lagrangian-eulerian multi-physics code. 5 2017.

[83] Alejandro Nocua, Florent Bruguier, Gilles Sassatelli, and Abdoulaye Gamatie. Elasticsimmate: A fast and accurate gem5 trace-driven simulator for multicore systems. In *2017 12th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*, pages 1–8, 2017.

[84] D.B. Noonburg and J.P. Shen. Theoretical modeling of superscalar processor performance. In *Proceedings of MICRO-27. The 27th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 52–62, 1994.

[85] Antoine Petitet, R. Whaley, Jack Dongarra, and A. Cleary. Hpl – a portable implementation of the high-performance linpack benchmark for distributed-memory computers. 2008.

[86] David Richards, Patrick Brantley, Shawn Dawson, Scott Mckenley, and Matthew O'Brien. Quicksilver, version 00.

[87] Mitsuhisa Sato, Yutaka Ishikawa, Hirofumi Tomita, Yuetsu Kodama, Tetsuya Odajima, Miwako Tsuji, Hisashi Yashiro, Masaki Aoki, Naoyuki Shida, Ikuo Miyoshi, Kouichi Hirai, Atsushi Furuya, Akira Asato, Kuniki Morita,

and Toshiyuki Shimizu. Co-design for a64fx manycore processor and "fugaku". In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2020.

[88] Sameer Shende and Allen Malony. The tau parallel performance system. *IJHPCA*, 20:287–311, 01 2006.

[89] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. The arm scalable vector extension. *IEEE Micro*, 37, 2017.

[90] Estela Suarez. The european processor initiative: overview and status update.

[91] Jingwei Sun, Guangzhong Sun, Shiyan Zhan, Jiepeng Zhang, and Yong Chen. Automated performance modeling of hpc applications using machine learning. *IEEE Transactions on Computers*, 69(5):749–763, 2020.

[92] Jialiang Tan, Shuyin Jiao, Milind Chabbi, and Xu Liu. What every scientific programmer should know about compiler optimizations? In *Proceedings of the 34th ACM International Conference on Supercomputing*, ICS '20, New York, NY, USA, 2020. Association for Computing Machinery.

[93] Thomas N. Theis and H.-S. Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in Science and Engineering*, 19(2):41–50, 2017.

[94] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton. LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Comp. Phys. Comm.*, 271:108171, 2022.

[95] Sam Van den Steen, Stijn Eyerman, Sander De Pestel, Moncef Mechri, Trevor E. Carlson, David Black-Schaffer, Erik Hagersten, and Lieven Eeckhout. Analytical processor performance and power modeling using micro-architecture independent characteristics. *IEEE Transactions on Computers*, 65(12):3537–3551, 2016.

[96] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. Augem: Automatically generate high performance dense linear algebra kernels on x86 cpus. In

*SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2013.

[97] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 2009.

[98] Han Zhao, Quan Chen, Yuxian Qiu, Ming Wu, Yao Shen, Jingwen Leng, Chao Li, and Minyi Guo. Bandwidth and locality aware task-stealing for manycore architectures with bandwidth-asymmetric memory. *ACM Transactions on Architecture and Code Optimization*, 15:1–26, 12 2018.

[99] Xinnian Zheng, Lizy John, and Andreas Gerstlauer. Lacross: Learning-based analytical cross-platform performance and power prediction. *International Journal of Parallel Programming*, 45, 12 2017.

[100] Xinnian Zheng, Haris Vikalo, Shuang Song, Lizy K. John, and Andreas Gerstlauer. Sampling-based binary-level cross-platform performance estimation. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, 2017.

[101] Thorsten Zirwes, Feichi Zhang, Peter Habisreuther, Anthony Jordan, Denev, Henning Bockhorn, and Dimosthenis Trimis. Optimizing load balancing of reacting flow solvers in openfoam for high performance computing. 2018.