UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

Niloofar CHARMCHI

Compressed Cache Layout Aware Prefetching

Thèse présentée et soutenue à Rennes, le 10 Juillet 2020
Unité de recherche : Inria
Thèse N° :

Rapporteurs avant soutenance :

Karine HEYDEMANN	Maîtresse de conférence, Sorbonne Université
David DEFOUR	Maître de conférence, Université de Perpignan Via Domitia

Composition du Jury :

Attention, en cas d'absence d'un des membres du Jury le jour de la soutenance, la composition du Jury ne comprend que les membres présents

Président :	Prénom Nom	Fonction et établissement d'exercice *(à préciser après la soutenance)*
Examinateurs :	Florent DE DINECHIN	Professeur des universités, INSA Lyon
	Steven DERRIEN	Professeur des universités, Université Rennes 1
Dir. de thèse :	André Seznec	Directeur de recherches, Inria
Co-dir. de thèse :	Caroline Collange	Chargée de recherche, Inria

Invité(s) :

Prénom Nom	Fonction et établissement d'exercice

Acknowledgement

First and foremost, I have to thank my parents for their care, love and support through my life. Thank you both for giving me the confidence and encouraging me to chase my dreams. You two always believed in me and wanted the best for me. Thank you for teaching me that all I need is to learn, be a good person and be happy in my life. None of these would have been possible without your love and support as my adorable mum and dad. Thanks to my brother, Sadegh, for giving me advice and comments during my tough times.

I would like to sincerely thank my advisors, André Seznec and Caroline Collange. Under their guidance, I learned how to develop a research approach. Furthermore, I would like to thank Erven Rohou and Virginie Desroches, PACAP team leader and assistant, for helping me to settle into the team. I would also like to thank my Master advisor, Dr. Reshadinezhad, who made me familiar with the field of computer architecture. I am also grateful to the members of my PhD committee.

I would like to give a special thanks to Caroline and Daniel. I was lucky to have you not only as my colleagues, but also my wonderful friends. Thank you for supporting and helping me during my thesis. You made this journey much easier for me.

To my amazing friends, Leila, Shahla, Arman, Narges, Masi and Fatemeh, thank you for always being there for me, listening and supporting me for years. Either when we were all in our country or now that we live in different countries. I am blessed to have such incredible friends in my life.

My life in France would not be so fun without Imane, Rabab, Nicolas and Daniel. Your friendship makes it a great experience. Thanks for your understanding and encouragement in my good and bad moments.

Table of Contents

Résumé en Français **9**
0.1 La compression de cache et le préchargement matériel . . . 9
0.2 Potentiel du préchargement dans un cache compressé . . . 11
0.3 Le prédicteur de compression . . . 12
0.4 Préchargement adapté au format de cache compressé (CLAP) . . . 13
0.4.1 Neighbor-CLAP . . . 13
0.4.2 Stride-CLAP . . . 14
0.5 CLAP adaptatif . . . 15
0.5.1 CLAP adaptatif global . . . 15
0.5.2 CLAP adaptatif local . . . 16

1 Introduction **17**
1.1 Memory hierarchy . . . 17
1.2 Cache compression . . . 18
1.3 Hardware prefetching . . . 20
1.4 Problem statement . . . 21
1.5 Proposed key idea and contributions . . . 22
1.6 Thesis outline . . . 24

2 Related work **25**
2.1 Cache compression and compaction . . . 25
2.1.1 Compressed cache layouts . . . 26
2.1.2 Cache compression methods . . . 35
2.2 Hardware prefetching . . . 42
2.2.1 Nextline prefetcher . . . 45
2.2.2 Stream prefetcher . . . 45
2.2.3 Stride prefetcher . . . 46
2.2.4 Best-offset prefetcher . . . 47
2.2.5 Adaptive prefetchers . . . 48

2.3 Cache compression and hardware prefetching interaction 49
2.4 Methodology 50
2.4.1 Simulation platform 50
2.4.2 Metrics of interest 52
2.5 Summary 53

3 Compressed cache layout aware prefetching (CLAP) 55
3.1 Analyzing potentials for prefetching 56
3.1.1 Characterization of compressed cache 56
3.1.2 Gradual refill problem of DISH 57
3.1.3 Opportunities for compressed cache layout aware prefetching 58
3.2 Compression predictor 59
3.2.1 Prediction criterion 60
3.2.2 Proposed compression predictor 62
3.3 Neighbor-CLAP 64
3.4 Stride-CLAP 67
3.5 CLAP evaluation 67
3.5.1 Bus utilization 69
3.5.2 Effective cache utilization 69
3.6 CLAP combination with other prefetchers 70
3.6.1 Nextline-based prefetcher with neighbor-CLAP 71
3.6.2 Stride-based prefetcher with neighbor-CLAP 71
3.6.3 Nextline-based prefetcher with stride-CLAP 71
3.7 Summary 74

4 Adaptive CLAP 77
4.1 Global adaptive CLAP 78
4.1.1 Usefulness predictor 80
4.1.2 Prefetcher selector 86
4.1.3 Global adaptive CLAP evaluation 87
4.2 Local adaptive CLAP 96
4.2.1 Local adaptive CLAP architecture 96
4.2.2 Usefulness in local adaptive CLAP 99
4.3 Adaptive CLAPs comparison 101
4.4 Summary 105

5 Conclusion **107**

5.1 Contributions 108

5.2 Perspectives 110

Publications of the thesis **113**

Résumé en Français

L'un des défis de conception majeurs pour les ordinateurs est l'écart de vitesse entre le processeur et la mémoire qui affecte les performances [HP11; Rog+09]. Les performances des processeurs s'améliorent de 60% par an; dans le même temps, le temps d'accès à la mémoire diminue de moins de 10% par an [Pat+97]. Afin de faire face à ce goulot d'étranglement, des techniques telles que la compression et la préchargement ont été introduites [Tre+01; MV15; FW14]. Cependant, les objectifs de ces deux approches sont différents et elles ne collaborent pas dans un système. Le préchargement vise à réduire la latence au prix d'une diminution de la capacité du cache. Au contraire, la compression tente d'augmenter la capacité en stockant plus de données dans la même taille physique; néanmoins, cela entraîne un coût supplémentaire en termes de latence. L'objectif de cette étude est de tirer parti des deux méthodes en coopération.

0.1 La compression de cache et le préchargement matériel

Récemment, la quantité de données produites par diverses applications augmente exponentiellement [MV15]. La compression du cache peut économiser la bande passante vers la mémoire externe et la compression de la mémoire peut réduire les défauts de page [RKP00; Bis+06]. Le cache de dernier niveau (LLC) a la plus grande taille dans la hiérarchie du cache et il stocke une quantité importante de données; par conséquent, il est avantageux d'augmenter la capacité du LLC pour pouvoir contenir plus de données dans la mémoire interne au processeur. La compression du cache peut être une solution prometteuse pour augmenter la capacité du cache et améliorer les performances des processeurs pour réduire les accès vers la mémoire externe [AW04a; Sar+15; Che+09; AS14].

L'idée des algorithmes de compression de cache est d'obtenir l'avantage de caches plus grands, tout en conservant la surface et la puissance de caches plus petits. Les caches compressés utilisent les techniques de compression [Pek+12; PS16] pour réduire la taille des blocs de cache et les méthodes de compactage [SSW16; SW13; SSW14] pour allouer

les blocs compressés ensemble dans une seule entrée du cache.

Yet Another Compressed Cache (YACC) [SSW16] et Skewed Compressed Cache (SCC) [SSW14] font partie des dispositions de cache compressées économes. Leur objectif est de regrouper les blocs contigus d'un super-bloc de mémoire unique dans un emplacement de cache unique. Lorsque tous les blocs du super-bloc sont compressibles, une entrée de cache doit être allouée pour le premier bloc, mais les blocs suivants ne nécessitent aucun espace de cache supplémentaire. L'objectif de cette étude est d'exploiter cette opportunité pour permettre la préchargement sans pollution.

Dictionary Sharing (DISH) [PS16] est une technique de compression qui tire parti de la disposition YACC. Un dictionnaire de compression commun est partagé entre tous les sous-blocs de chaque super-bloc. Dans la suite du résumé, nous supposons que DISH est utilisé comme algorithme de compression.

La compression peut entraîner une augmentation de la capacité du cache; néanmoins, il augmente la latence de la mémoire, la complexité et la surcharge matérielle. Ainsi, il est essentiel de profiter d'un algorithme de compression à faible latence de décompression.

Comme mentionné précédemment, la latence de la mémoire externe est relativement élevée et elle devient un goulot d'étranglement crucial dans les systèmes de mémoire. La préchargement est une approche pour réduire le nombre d'accès à la mémoire et finalement réduire la latence du système. Un mécanisme de préchargement optimal et idéal peut supprimer presque tous les échecs de cache et améliorer les performances dans une large mesure [APD01]. Cette thèse porte sur le préchargement des données; cependant, il y a eu aussi beaucoup d'études sur le préchargement d'instructions [RCA99; Fer+08; FKF11].

le préchargement matériel est une récupération spéculative de lignes de cache qui n'ont pas encore été demandées par le programme. Plusieurs approches ont été proposées pour le préchargement matériel telles que le préchargement par pas (stride) [FPJ92; SSC00], le préchargement de la ligne suivante (Nextline) [Smi82] et le préchargement best-offset [Mic16].

Si les préchargements sont effectués avec précision et suffisamment tôt, ils peuvent masquer les accès à la mémoire externe [**opportunPref**; Sri+07]. Par conséquent, le préchargement matériel est devenu le centre d'attention des chercheurs dans ce domaine. Cependant, il existe certains inconvénients au préchargement matériel. Cela peut augmenter les conflits pour la bande passante mémoire disponible, car en plus des demandes d'accès, les demandes de préchargement vont également à la DRAM, ce qui provoque des conflits de banc de DRAM supplémentaires. De plus, des préchargements inexacts aug-

mentent la consommation d'énergie en raison des accès inutiles à la mémoire. En outre, le préchargement peut entraîner une pollution du cache si le préchargement remplace des données utiles qui seront nécessaires ultérieurement par le programme par des données inutiles.

0.2 Potentiel du préchargement dans un cache compressé

Dans cette section, nous explorons les possibilités de créer une synergie entre le cache compressé et le préchargement.

Nous avons simulé un cache compressé LLC à l'aide de DISH, pour 27 benchmarks. Dans notre configuration, la compression du cache n'apporte que des améliorations modestes, ou peut même causer plus de défauts de cache en raison de la concurrence des blocs consécutifs non compressés pour les entrées dans le même ensemble du cache.

En effet, la capacité potentielle du cache compressé peut être sous-utilisée en raison de facteurs tels que la non-compressibilité ou la non-compactabilité d'un bloc, et le manque de localité spatiale.

— **La non-compactabilité d'un bloc** peut se produire lorsqu'un sous-bloc n'est compressible que par lui-même et qu'il n'est pas co-allouable avec d'autres. Dans ces cas, il n'y a qu'un seul sous-bloc dans le super-bloc; par conséquent, nous ne profitons pas de la compression.

— **Manque de localité spatiale** pourrait entraîner une sous-utilisation dans les caches compressés. Si les blocs voisins d'un super-bloc compressible ne sont pas accessibles, l'espace supplémentaire gagné par la compression reste inutilisé. Cela est dû au fait que chaque fois qu'un sous-bloc est amené dans le cache, le cache alloue un super-bloc pour le sous-bloc demandé. Ce super-bloc a un sous-bloc valide, celui demandé et trois sous-blocs invalides. Si les autres sous-blocs ne sont pas accessibles dans le programme avant que le sous-bloc actuel ne soit expulsé, cet espace supplémentaire sera inutilisé.

Outre la sous-utilisation des caches compressées, l'expulsion en masse et le remplissage graduel, qui est au centre de ce travail, est un autre problème dans les dispositions sectorielles. Comme le remplacement est effectué à la granularité des super-blocs, tous les sous-blocs du super-bloc compressé sont expulsés à la fois. Par conséquent, une seule expulsion de super-bloc peut générer quatre défauts de cache ultérieurs pour remplir le

super-bloc complet.

Comme mentionné précédemment, même en compressant le LLC, nous obtenons une amélioration modeste en termes de MPKI (défauts de cache par millier d'instructions). Ainsi, nous étudions le nombre de défauts de cache qui pourraient être évités en récupérant systématiquement les blocs qui peuvent être co-alloués dans un super-bloc.

Afin d'évaluer le potentiel de préchargement, nous comptons les opportunités de préchargement. Les défauts de cache qui se produisent lors d'un succès de l'accès au super-bloc et d'un défaut de sous-bloc sont des candidats potentiels pour le préchargement. Si ce bloc peut être co-alloué avec ceux qui sont déjà présents dans le super-bloc, il aurait pu être récupéré à l'avance sans frais de capacité et le défaut de cache aurait pu être évité. Par conséquent, nous n'avons pas besoin d'attendre les accès suivants à chaque sous-bloc pour remplir le super-bloc compressé. Cela n'efface aucun bloc du cache.

À l'aide du simulateur gem5, nous avons mesuré la fraction des défauts de cache qui pourraient être évités en tirant parti du préchargement. Les résultats montrent que nous pourrions réduire le nombre de défauts de cache jusqu'à 20%, en préchargeant les blocs compressés avant qu'ils ne soient demandés par le processeur, sans expulser aucune donnée du cache.

Cela suggère qu'il existe un potentiel d'interaction synergique du préchargement et de la compression du cache dans l'architecture du processeur. Ainsi, en préchargement des blocs compressibles en L3, nous recherchons à réduire le nombre de défauts de cache du LLC.

0.3 Le prédicteur de compression

Comme cela a été mentionné, il est possible de créer une synergie entre le préchargement matériel et le cache compressé en préchargeant des sous-blocs co-allouables. Par conséquent, nous devons prédire quels blocs sont compressibles ou co-allouables à l'avance, avant d'accéder à leurs données. Afin d'estimer ces blocs comme candidats au préchargement avant que leur contenu ne soit disponible, nous avons besoin d'un prédicteur.

Comme le prédicteur doit prévoir la compressibilité des blocs qui n'ont pas été utilisés récemment ou qui n'ont jamais été utilisés, une prédiction basée uniquement sur l'adresse du bloc n'est pas pratique. Au lieu de cela, nous prédisons la compressibilité d'un *flux* de lectures mémoire, où un flux correspond à tous les accès effectués par une instruction de chargement donnée dans le programme. Par conséquent, la prédiction est effectuée sur

la base du PC de la demande, et non sur la base de l'adresse. De plus, la prédiction est effectuée dans le LLC.

L'idée de la prédiction basée sur le flux est que tous les emplacements de mémoire accessibles par une instruction de chargement donnée sont susceptibles de partager le même type de données et des caractéristiques similaires. Ainsi, le fait que les blocs précédents auxquels accédait une instruction de chargement étaient compressibles est un bon indicateur de la compressibilité des futurs blocs chargés à partir de la même instruction.

0.4 Préchargement adapté au format de cache compressé (CLAP)

Nous avons montré qu'une partie substantielle des défauts de cache dans les caches compressés se produisent dans des blocs qui pourraient être alloués et compactés dans un super-bloc déjà présent dans le cache. Ces défauts de cache pourraient être évités sans frais en termes de capacité du cache en introduisant ces blocs dans le cache à l'avance. Tirant parti de cette opportunité, nous proposons deux configurations de CLAP, un système unifié qui bénéficie du préchargement et de la compression du cache en coopération.

Dans cette configuration, L1, L2 et la mémoire ne sont pas compressés et LLC est un cache compressé. Par conséquent, il existe un compresseur pour transférer les données de la mémoire vers LLC. En outre, il existe une logique de décompression pour décompresser les données afin d'utiliser les données dans la CPU et de les stocker dans les L1 et L2.

0.4.1 Neighbor-CLAP

Afin d'améliorer la compression du cache et de réduire le problème de remplissage graduel des caches compressés, nous proposons le neighbor-CLAP, un préchargement adapté au format de cache compressé pour la format YACC. Le but est d'éviter les défauts de cache qui se produisent sur un bloc co-allouable sur un super-bloc atteint par un accès. Par conséquent, nous pouvons tirer parti des dispositions de cache compressées au moyen du prédicteur de compression et profiter du préchargement.

Dans un préchargement adapté au format de cache compressé, chaque fois qu'une demande en lecture provoque un échec de super-bloc, le prédicteur de compression est appelé. Le prédicteur de compression proposé prédit si le bloc est compressible. Sur la base de la décision du prédicteur, le préchargement adapté charge à l'avance trois blocs ou

ne procède pas au préchargement. S'il prédit que le bloc est compressible, le cache envoie trois demandes de préchargement des trois sous-blocs voisins dans un super-bloc. Dans le cas où le prédicteur prédit que le bloc n'est pas compressible, le système ne précharge rien.

0.4.2 Stride-CLAP

Neighbour-CLAP ne récupère que les voisins dans le même super-bloc que le bloc de la demande; cependant, nous pouvons aller plus loin et faire la prédiction à travers les super-blocs. Comme mentionné précédemment, le prédicteur de compression ne dépend pas de l'adresse; il prédit si une instruction de chargement donnée dans un programme (PC) accède à un bloc compressible; par conséquent, nous pouvons appliquer notre prédicteur de compression sur les accès à pas constants. Le préchargement par pas calcule le pas sur la base d'un flux d'adresses correspondant à la même instruction de chargement. S'il est prévu qu'un chargement mémoire accède à un bloc compressible (à l'aide du prédicteur de compression), les autres accès de la même instruction sont également susceptibles d'être compressibles.

L'objectif est d'avoir un préchargement adapté à la compression qui s'active si deux critères sont remplis: une confiance élevée dans le pas et un flux d'accès compressibles; nous appelons cette configuration stride-CLAP. stride-CLAP effectue un préchargement par pas à chaque accès si le prédicteur prédit qu'un bloc est compressible; sinon, il n'envoie pas les demandes de préchargement à la mémoire.

Nous avons implémenté un cache compressé sans préchargement, un cache compressé avec neighbour-CLAP et un cache compressé avec stride-CLAP, en utilisant DISH comme algorithme de compression. Les évaluations montrent que le neighbor-CLAP et le stride-CLAP surpassent le cache compressé sans préchargement en termes d'IPC et de MPKI. Ils améliorent les performances de 3% par rapport au cache compressé sans préchargement, en moyenne (jusqu'à 17% pour neighbour-CLAP et jusqu'à 20% pour stride-CLAP, dans *cactusADM*). De plus, le neighbor-CLAP et le stride-CLAP réduisent les défauts de cache LLC des applications de 9%, en moyenne (jusqu'à 34,8% pour le neighbor-CLAP et jusqu'à 35,3% pour le stride-CLAP, dans *soplex*).

0.5 CLAP adaptatif

Le principal problème avec le préchargement est qu'il n'y a pas de politique de préchargement unique qui fonctionne pour toutes les applications. Par conséquent, nous avons besoin d'un mécanisme de sélection de politique de préchargement pour trouver dynamiquement la meilleure politique en fonction de l'application en cours d'exécution. Dans cette section, nous proposons deux systèmes de préchargement adaptatifs adaptés au format de cache compressé (adaptive CLAP): CLAP adaptatif global et CLAP adaptatif local.

Le CLAP adaptatif global sélectionne la meilleure politique de préchargement dans chaque intervalle de temps pour l'ensemble du cache. Il estime laquelle des politiques aurait été la meilleure dans les intervalles précédents et utilise celle-ci pour l'intervalle suivant. Le CLAP adaptatif local choisit une politique de préchargement pour chaque instruction de chargement individuellement. En exploitant le CLAP adaptatif local, nous trouvons la meilleure technique de préchargement pour chaque chargement particulier.

0.5.1 CLAP adaptatif global

CLAP adaptatif global sélectionne dynamiquement la meilleure politique de préchargement parmi deux préchargements adaptés à la compression pour chaque intervalle de temps. Nous proposons un mécanisme de sélection générique qui peut être appliqué à n'importe quelle politique de préchargement arbitraire; pas seulement CLAP.

Le CLAP adaptatif global active une seule des deux politiques de préchargement pour le cache entier pendant un intervalle et bascule périodiquement entre les deux politiques. Nous choisissons de diriger le préchargement par intervalle afin d'éviter de basculer trop fréquemment entre les politiques. Nous estimons l'historique d'utilité des deux politiques de préchargement. Au début de chaque intervalle, celle dont l'utilité estimée est la plus élevée est sélectionnée pour effectuer le préchargement dans l'intervalle suivant.

L'utilité d'une politique de préchargement est définie comme le nombre de préchargements utiles divisé par le nombre total de préchargements. Un préchargement utile se produit lorsqu'un bloc de données préchargé est disponible dans le cache et qu'il y a une demande sur ces données avant qu'elles ne soient expulsées du cache. Si le bloc préchargé est expulsé avant d'être accédé par une demande, le préchargement n'est pas utilisé.

Afin d'estimer l'utilité des politiques de préchargement, nous proposons un prédicteur d'utilité. Nous attribuons un prédicteur d'utilité à chaque politique de préchargement basée sur CLAP. En tirant parti des deux mécanismes de préchargement, le prédicteur

d'utilité et le sélecteur de politique de préchargement, nous proposons une technique de préchargement adapté au format de cache compressé adaptatif global. Le CLAP adaptatif global active une seule des politiques de préchargement pour l'ensemble du cache dans chaque intervalle, sur la base des résultats des prédicteurs d'utilité associés à chaque politique de préchargement, et bascule entre les politiques à la fin des intervalles.

0.5.2 CLAP adaptatif local

La décision du CLAP adaptatif global de diriger le préchargement est basée sur la moyenne des préchargements utiles dans un intervalle précédent. Cette décision globale est prise pour l'ensemble du cache et la politique de préchargement sélectionnée est activée jusqu'à la fin de l'intervalle, puis, le sélecteur de politique de préchargement recalcule l'utilité de chaque politique de préchargement et choisit la politique avec une utilité plus élevée. Cependant, le CLAP adaptatif global ne fonctionne pas bien en raison des différentes caractéristiques des différentes instructions de chargement dans les applications. Par conséquent, il peut être plus avantageux de diriger la politique de préchargement pour chaque accès en lecture individuellement.

Dans cette technique, l'objectif est de sélectionner le préchargement localement, en fonction de chaque accès mémoire individuel. Dans le CLAP adaptatif local, la sélection de politique de préchargement se produit parmi trois mécanismes de préchargement simples qui sont le préchargement par pas, le préchargement de ligne suivante et le préchargement de voisins. Dans cette configuration, nous pouvons profiter de différentes politiques de préchargement basés sur deux facteurs: la confiance dans le pas entre accès et la compressibilité du bloc. Nous implémentons une sélection de politique de préchargement qui fonctionne comme suit: Le CLAP adaptatif local profite à la fois de la confiance de l'estimation du pas et de la compressibilité du bloc pour sélectionner la meilleure politique de préchargement et créer une interaction positive dans le système de préchargement.

Contrairement au sélecteur de politique de préchargement du CLAP adaptatif global, qui oriente le préchargement globalement en ce qui concerne le nombre moyen d'utilité, le CLAP adaptatif local ajuste le préchargement localement en fonction du PC déclenchant l'accès. Dans cette configuration, différents préchargements sont activés dans différentes circonstances.

En comparant le CLAP adaptatif global et le CLAP adaptatif local, les résultats montrent une réduction du nombre de défauts de cache dans le LLC et également de meilleures performances dans la configuration du CLAP adaptatif local.

Chapter 1

Introduction

One of the major challenges for computers is the speed gap between processor and the memory that affects the computer performance [HP11; Rog+09]. The processor performance has been improving 60% per year; meanwhile, the memory access time has been decreasing less than 10% per year [Pat+97]. In order to confront this bottleneck, techniques such as compression and prefetching have been introduced [Tre+01; MV15; FW14]. However, the goals of these two approaches are different and they do not collaborate in a system. Prefetching aims to reduce the latency at the cost of decreased cache capacity. On the contrary, compression attempts to increase the capacity by storing more data in the same physical size; nevertheless, it brings extra cost in terms of latency. The objective of this study is to take advantage of both methods cooperatively.

1.1 Memory hierarchy

A perfect memory is the one that has a high capacity, low cost and high speed. Since these three parameters are in conflict with each other, such a memory is not implementable [MV99]. In order to reduce the speed gap between off-chip memory and processor [WM95], on-chip memory, referred to as cache, is adopted [Smi82]. Caches store part of the working set and deliver those data to the processor quickly when it is needed. Therefore, the memory hierarchy is divided into several levels. The first level that is closer to CPU is the smallest, most expensive per bit and fastest, while the last level cache (LLC) is the biggest, cheapest and slowest. In most current general-purpose systems, the cache hierarchy consists of three levels, but there are some microarchitectures that take advantage of a four-level cache hierarchy [Ham+14]. Cache hierarchy consists of multiple levels depending on the system architecture. When the requested data is found in a cache level, it is a cache hit and the request can be serviced right away; however, if the data is not there, i.e. cache miss, a request is sent to the next cache level (or the main memory) in order to bring the data to the cache.

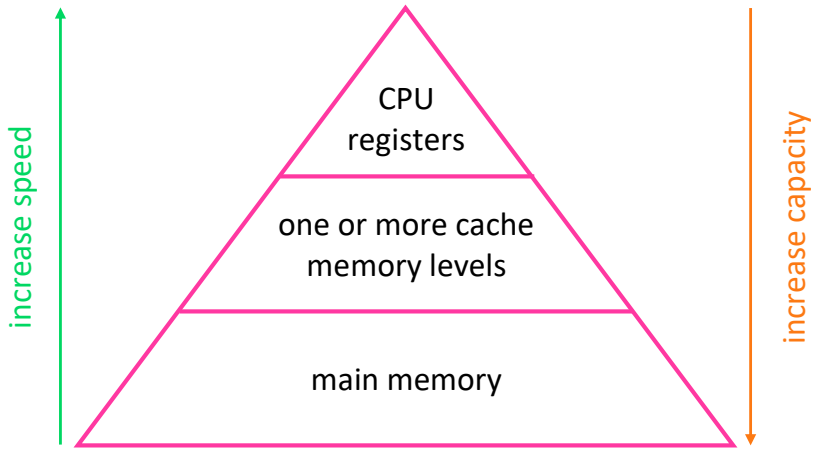


Figure 1.1 – Memory hierarchy

Memory hierarchy takes into account temporal and spatial locality, the two ways that data can be fetched from memory and stored in the cache. Temporal locality means the current fetched data may be accessed again in near future. Spatial locality is based on the fact that the adjacent data that reside next to the fetched data may be accessed soon. A memory hierarchy in a system generally contains main memory, cache memory and CPU registers (Figure 1.1). There are various technologies to build memories [Wu+09]. In this study, the main memory is a Dynamic Random Access Memory (DRAM) that services cache requests and communicates with I/O devices. Cache memory is a fast small memory close to the CPU which is made of Static Random Access Memory (SRAM). Finally, CPU registers are small and fast buffers within the processor.

The on-chip LLC is one of the most critical parts of a computer system. Since off-chip memory latency is high, finding techniques to minimize off-chip memory accesses is of essence. Methods such as cache compression and hardware prefetching can lead to a decrease in number of LLC misses [SSW16; Mic16].

1.2 Cache compression

Recently, the amount of data produced by various applications is increasing exponentially [MV15]. Cache compression can save the off-chip memory traffic and memory compression can decrease page faults [RKP00; Bis+06]. As it was mentioned before, LLC has the largest size in the cache hierarchy and it stores significant amount of data; there-

fore, it is beneficial to increase the LLC capacity to be able to fit more data in on-chip memory. Cache compression, which is the focus of our work, can be a promising solution for increasing cache capacity and performance improvement in processors to reduce off-chip accesses [AW04a; Sar+15; Che+09; AS14].

The key idea of cache compression algorithms is to obtain the benefit of larger caches, while retaining the area and power of smaller caches. Compressed caches use a compression technique to reduce the size of the cache blocks containing redundant data and a compaction method to allocate compressed blocks together into one data entry. The blocks that are compacted and stored in compressed formats in one data entry are called *compactable* or *co-allocatable* blocks. In this thesis, we will use these two terms interchangeably.

Cache compression schemes, such as Frequent Value Compression (FVC) [YG02], Frequent Pattern Compression (FPC) [AW04b], C-Pack [Che+09], Base-Delta-Immediate compression (BDI) [Pek+12] and Dictionary Sharing (DISH) [PS16], show that there is a significant redundancy in data values in the cache. We can use compression algorithms to simulate a larger cache without increasing the physical size of the cache. A compression technique should be effective in saving storage space. In order to allocate compressed blocks in one data entry, we need a compaction method. Some well-known compaction techniques are Decoupled Compressed Cache (DCC) [SW13], Skewed Compressed Cache (SCC) [SSW14] and Yet Another Compressed Cache (YACC) [SSW16].

Compression may result in cache capacity augmentation; nonetheless, it increases memory latency, complexity and hardware overhead. Thus, taking advantage of a compression algorithm with low decompression latency is essential. Furthermore, there are negative interactions between compression and cache replacement policies that causes performance degradation in the system [GAS16].

An overview of a three-level cache hierarchy with compression units is illustrated in Figure 1.2. It is worth mentioning that we can compress all cache levels but we don't aim for compressing L1 due to several reasons. First, L1 cache is too small and by compression we increase the hit latency. Moreover, L1 writes happen on word granularity; while data in L2 and L3 is stored in cache line granularity so in this case implementing compression on L1 becomes more complex.

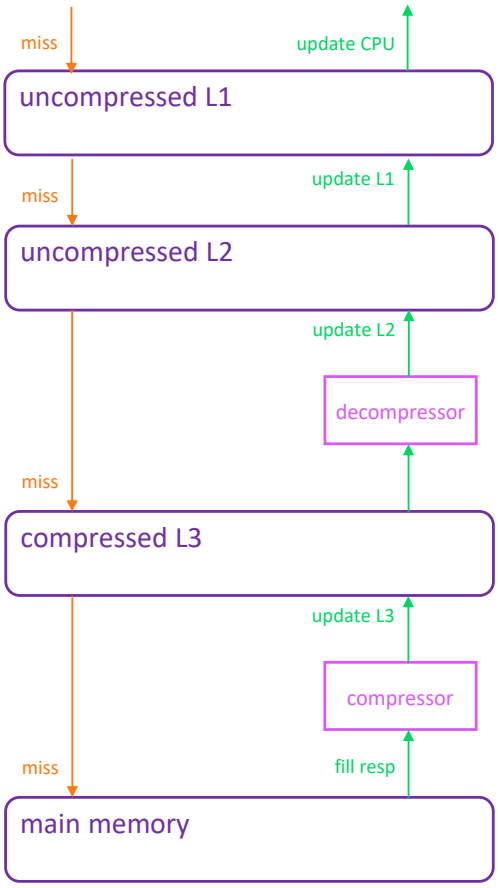


Figure 1.2 – Three-level cache hierarchy with compression in LLC

1.3 Hardware prefetching

As previously mentioned in this chapter, off-chip memory latency is relatively high and it is becoming a crucial bottleneck in memory systems. Prefetching is an approach to reduce the number of memory accesses and ultimately reduce the latency of the system. An optimal and ideal prefetching mechanism can remove almost all the cache misses and enhance the performance to a great extent [APD01]. The focus of this dissertation is on data prefetching; however, there has been lots of studies on instruction prefetching as well [RCA99; Fer+08; FKF11].

Hardware prefetching is a speculative fetching of cache lines that have not been requested yet by the program. Multiple approaches have been proposed for hardware prefetching, such as stride prefetching [FPJ92; SSC00], Nextline prefetching [Smi82] and best-offset prefetching [Mic16].

If the prefetches are performed accurately and early enough, they can hide off-chip memory accesses [Sri+07; ZCS10]. Therefore, hardware prefetchers have become the center of attention among researchers in this field. However, there are some downsides in hardware prefetching. It can increase the contention for the available memory bandwidth, because apart from demand requests, prefetch requests also go to DRAM, which causes additional DRAM bank conflicts. Moreover, inaccurate prefetches increase the energy consumption because of unnecessary memory accesses. Furthermore, prefetching can cause cache pollution if the prefetcher prefetches useless data and replaces useful data in the cache that will be needed later by the program.

1.4 Problem statement

Cache compression and hardware prefetching are techniques to improve system performance and to decrease memory accesses in a program; however, they have different strengths and weaknesses. Cache compression on the one hand increases the capacity of the cache by storing more data in the same amount of physical storage, but on the other hand it adds extra latency to the memory system. Hardware prefetching diminishes latency by bringing data before they are requested by the program; nevertheless, it may cause cache pollution due to bringing useless data to the cache and evicting useful data.

Figure 1.3 illustrates variations in latency and capacity in a cache hierarchy. As it is shown in this figure, The first level cache (L1) has the smallest latency but less capacity; and the last level cache (L3) has the largest capacity but the highest latency. By applying prefetching in LLC, the latency is decreased at the cost of polluting the cache (bringing/evicting more data to/from the cache). Compressing LLC leads to increase the effective capacity of the cache but also the latency, because of the decompression latency, at one and the same time. Hence, the problem is that the goals of cache compression and hardware prefetching are in different directions. In this study we are looking for an approach to take advantage of both techniques cooperatively and to obtain the best performance in the system.

There are some studies on interactions between cache compression and hardware prefetching. Prefetched Blocks Compaction (PBC) [RPM16] can only compress and compact blocks that are prefetched, whereas in this work the goal is to apply compression and compaction techniques on all data in the LLC. Synergistic cache Compression and Prefetching (SCP) [PHM15] reduces the off-chip memory accesses caused by data stream

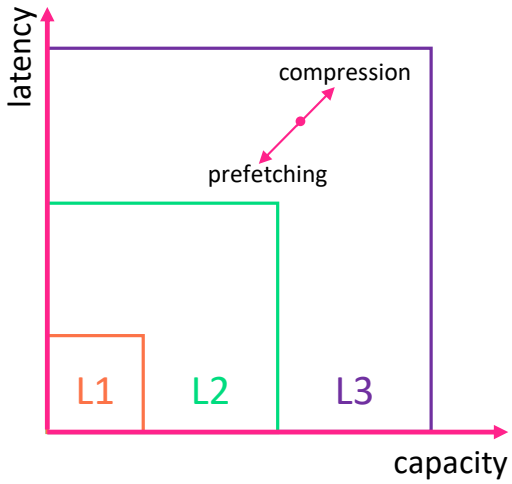


Figure 1.3 – Cache compression and hardware prefetching in LLC

prefetchers [Jou90; BC91] (Section 2.2.2), and to do this, it uses the extra space achieved by having cache compression. SCP utilizes the cache space saved by a compression mechanism to store the metadata required by some prefetchers. SCP does not aim for fitting more data in LLC. However, the focus of this thesis is to propose a prefetcher which is aware of compressed cache layout and fits more useful blocks in the cache.

This dissertation targets a prefetching approach that is integrated with compressed cache layout. In order to achieve this goal, we need to answer following questions. The first question is to find out how much are the prefetching potentials in compressed cache layouts. The second question that this thesis attempts to answer is how to take advantage of the potentials and estimate which data are good candidates for prefetching without degrading the system performance. The next problem answered in this dissertation is how to implement a compressed cache layout aware prefetcher and make a synergy between cache compression and hardware prefetching. Finally, the last challenge is to adapt the prefetcher to different applications and workloads.

1.5 Proposed key idea and contributions

In this thesis the first objective is to find potentials for prefetching without polluting a sector-based compressed cache layout, namely YACC [SSW16]. In YACC, on a demand miss on a compressible block, a complete 64 bytes cache block is allocated; however, the data may be compressible to 16 bytes or 32 bytes. By allocating a compressed block in a

data entry, there will be some space left for the neighbor blocks. Therefore, if the other neighbor blocks are also compressible and compactable with the demanded block, they can be prefetched without inducing any extra cache eviction. To the best of our knowledge, this is the first prefetching technique that takes into account the layout of a compressed cache.

The proposed compressed cache layout aware prefetcher (CLAP) finds a trade-off between the utility of prefetched data and the cache capacity. This positive interaction between hardware prefetching and cache compression results in performance improvement. Therefore, prefetching and compression can take advantage of each other. Prefetching, however, increases the workload's working set size; it can evict useful data and cause misses. In some caches, compression can overcome this issue by increasing effective cache size. Thus, the extra capacity provided by compression can hide the disadvantage of prefetching.

This dissertation makes the following contributions:

— We show that a compressed cache is subject to problems such as non-compatibility of a block, lack of spatial locality and gradual refill. Furthermore, our study shows that a substantial portion of the cache misses in compressed caches occur in blocks that could be allocated and compacted in a physical space that is already available in the cache. These misses could be avoided at no cost in cache capacity by bringing these blocks in the cache in advance. Therefore, we can take benefit of prefetching on top of compression **(Chapter 3)**.

— In order to identify which blocks should be prefetched, we propose a compression predictor. Leveraging the prefetching opportunity, we propose CLAP, a unified system that benefits from prefetching and cache compression cooperatively. We propose two configurations for CLAP: Neighbor-CLAP prefetches the four blocks that are expected to be compressible and co-allocatable in the same data entry. Stride-CLAP prefetches a stream of addresses if the requested load address is accessing a compressible block. We evaluate the benefits of both configurations of CLAP compared to a compressed cache without prefetching. The system that exploits CLAP in LLC has 3% performance improvement and 9% reduction in number of misses, on average. Moreover, we show that our proposed strategy is also applicable to other prefetchers, such as Nextline and stride **(Chapter 3)**.

— We introduce two adaptive CLAPs that decide dynamically among different prefetchers. The global adaptive CLAP uses a global approach for the whole cache and

decides between two prefetchers, depending on useful prefetches in the application. The local adaptive CLAP takes advantage of four different prefetchers and choose the appropriate prefetcher for each individual memory load **(Chapter 4)**.

1.6 Thesis outline

This study mainly focuses on proposing a unified system, compressed cache layout aware prefetching, that takes advantage of both hardware prefetching and cache compression techniques cooperatively. This approach increases visible cache capacity and mitigates latency without polluting the cache, simultaneously.

Chapter 2 provides a background on compressed cache layouts, cache compression and hardware prefetching methods. Furthermore, the experimental framework, the system configuration and the metrics of interest are explained in this chapter.

Chapter 3 discusses the problems in compressed caches and explores the opportunities for a synergistic interaction between prefetching and cache compression. In this chapter we show that there is potential for prefetching and we can reduce the number of misses by prefetching compressible blocks beforehand. In order to figure out which blocks are candidates for prefetching, we propose a compression predictor. We estimate the accuracy of the proposed compression predictor against an oracle compression predictor and an oracle compaction predictor. The chapter continues with a discussion on the hardware overhead of the predictor. Taking advantage of the compression predictor, we propose a compressed cache layout aware prefetching that prefetches neighbor blocks only if the requested block is predicted compressible. In this layout, we are able to prefetch without cache pollution and we can avoid misses that may happen due to spatial locality in the future.

The thesis continues with introducing two adaptive compressed cache layout aware prefetchers in Chapter 4. The difference between the two adaptive CLAPs is the prefetcher selector which decides which prefetcher is suitable for an application. The global adaptive CLAP steers the prefetcher based on the prefetch usefulness in each interval. The local adaptive CLAP navigates the prefetcher locally based on each individual memory access. In this technique, the prefetcher selector selects among four different prefetchers.

Finally, Chapter 5 presents concluding remarks of the dissertation and discusses possible future work.

Chapter 2

Related work

Employing cache compression and hardware prefetching in a system can lead to performance improvements and memory access reduction. In this chapter, the necessity of cache compression algorithms and compressed cache layouts is first discussed. Moreover, state-of-the-art hardware prefetching methods and the interactions between prefetching and compression are presented. Afterwards, we introduce the simulation platform and the main metrics of interest to evaluate the system performance.

This chapter answers the following questions:

— What are the layouts for compressed caches?

— What are hardware prefetching issues?

— How do cache compression and hardware prefetching interact in the state-of-the-art?

2.1 Cache compression and compaction

Caches play a crucial role in memory systems by storing the important data close to the CPU. They decrease the number of costly off-chip main memory accesses and replace them with less expensive on-chip cache accesses [SP11]. Nowadays, in modern high performance processors, increasing the cache capacity and reducing memory accesses is crucial.

Storing data in compressed form in a cache is appealing but one has to tackle two important issues, namely the compression scheme and the compressed cache layout. A cache compression scheme gets a piece of data (e.g. a 64-byte data block) and shrinks it to a smaller size; hence, the data can be represented with fewer bits. Cache compression can be a solution for increasing the cache capacity by compressing data and obtaining benefits of larger caches while using the area of smaller caches [Pek+12; DPS09; Kim+11]. Taking advantage of compression, we can avoid data redundancy in the applications; however, compression schemes suffer from drawbacks such as adding extra latency to the system (Section 2.1.2).

However, solely compressing the blocks is not enough; allocating compressed blocks next to each other in a given data storage is also necessary. Therefore, we need a layout for compressed caches (compaction mechanism) to compact compressed blocks in one data entry. However, not all the data are compressible; hence, a compressed cache must hold both compressed and uncompressed blocks. Thus, in a compressed cache layout, unlike the conventional cache layout, the size of the blocks is not fixed and they have variable physical block sizes.

Although compressed cache layouts fit more blocks in the same data entry, but they add complexity to the tag management, comparing to a conventional uncompressed cache layout. In an uncompressed cache, each tag is correlated to one data entry that contains only one cache block. However, in a compressed cache layout, we may have multiple cache blocks in one data entry; thus, special techniques for handling tag and data are required. There are various approaches for managing compressed cache blocks and compacting them [SSW14; SSW16].

Douglis [Dou93] observed that by taking advantage of compression, the working set of some large applications can be fitted in a small part of the memory; therefore, we can increase cache capacity and improve the performance. In the last few years, lots of work have been done on cache compression [AW04a; Pek+12; PS16; Che+09; Hon+19] and also memory compression [Sha+14; DS11; Pek+13; ES05] that shows the importance of these techniques in performance improvement of the systems. Furthermore, there are multiple techniques proposed for compressing floating-point data [ILS05; RKB06; ADS15]. However, in this thesis, we do not focus on floating-point data specifically. We introduce general purpose compression schemes as state-of-the-art.

2.1.1 Compressed cache layouts

[YZG00] and [LHK99] introduced layouts that compact the compressed cache blocks, that are compressible to half of their size, in a single data entry. These two methods both assume two tags per data entry that means there is one tag correlated to one compressed cache block. The tag-data mapping of these layouts is shown in Figure 2.1. Each tag entry contains a tag, a valid bit (V for valid and I for invalid) and a compressibility bit (C for compressed, U for uncompressed and X for an invalid data). Three different cases may happen in this layout:

— Both blocks are valid and compressible to half of their size or less (blocks A and B).

— Only one of the blocks is valid and compressible (block C).
— The block is valid and uncompressed (block E).

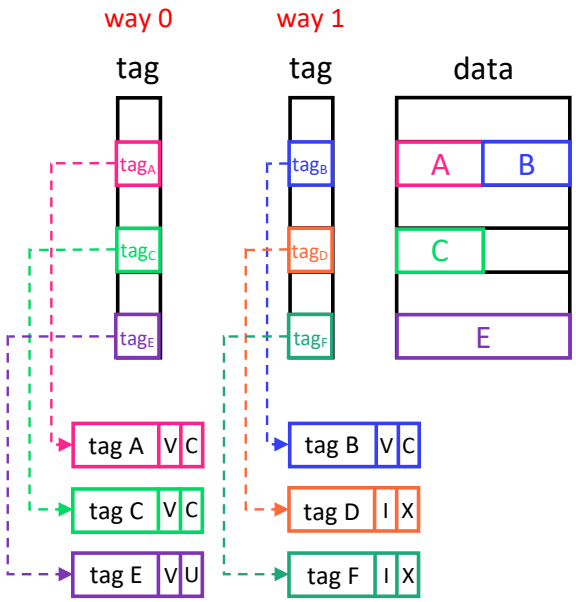


Figure 2.1 – Tag-data mapping in [YZG00] and [LHK99]

Lee et al. [LHK99] introduced a compressed memory system that compresses the blocks selectively, which means the blocks are stored as compressed only if the compression factor is more than 2 (they are compressed to less than half of their size and we can have at least two compressed blocks in a data entry); therefore, they do not compress the blocks that are compressible to more than half of their size because they may not be compactable with any other block. Thus, using this technique, the decompression overhead can be reduced. The same approach was used in [YZG00] by Yang et al.

As it was mentioned, [YZG00] and [LHK99] can not compact the blocks that are compressible to more than half of their size. Later, Alameldeen et al. [AW04a] proposed a variable-segment cache that divides the data entry into multiple 8-byte segments. Therefore, a block can be compressed to a variable number of 8-byte segments. Compressed blocks can be compacted with other blocks if it is possible. This technique stores a compressed block in contiguous segments and unlike previous works, it is able to compact even more than two blocks.

The techniques for cache compaction introduced above [AW04a; YZG00; LHK99] requires extra metadata and tag (storing more than one tag per data entry) which results in

a large overhead. In this section, we present the most advanced compressed cache layouts, such as DCC [SW13], SCC [SSW14], YACC [SSW16] and Touché [Hon+19]. SCC and DCC are based on sectored caches [Lip68] that was introduced in 1968. We recall below the structure of a sectored cache briefly. Taking advantage of sectored cache and associating only one tag per data entry can be useful in compressed caches. In order to maintain a consistent terminology, we refer to cache sectors as *super-blocks* for both sectored caches and compressed cache layouts.

Super-block

Unlike direct mapped caches, in which each memory block maps to a single possible cache location, the n-way set-associative cache is divided into sets, and a block can map to one of the sets. Within a set there are n blocks (n ways); therefore, each memory address maps to a specific set, but it can be allocated to any one of the n blocks in the set. On look-ups, the cache reads blocks from all ways in the selected set and checks the tags for a hit to determine which block was accessed.

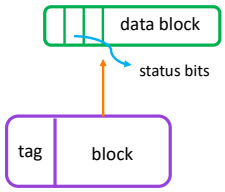
A set-associative cache requires one tag per block. This means that if we have a 4MB cache and each cache line is 64B, we have 16384 entries and we need one tag for each entry. Storing this much tag bits occupies a large capacity of the cache; therefore, reducing the tag overhead is crucial.

A sectored cache [Lip68] is a set-associative cache that is designed using a shared tag among contiguous cache blocks; thus, it is able to diminish the tag overhead. In this layout, the sectored cache is divided into sets of super-blocks and each super-block consists of blocks, a group of contiguous entries that share the same tag. Hence, using sectored cache reduces the tag area comparing to a conventional cache. In a super-block, each block contains the data and also some additional status bits such as valid and dirty bits, as in the conventional uncompressed cache (Figure 2.2).

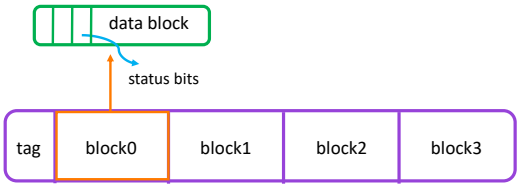
The most recent compressed cache layouts consider the compressed cache as a sectored cache with a variable number of blocks per super-block. In these layouts, the blocks that are residing in super-blocks are called *sub-blocks*.

The objective of super-block-based compressed caches is to pack the contiguous blocks of a super-block in a single cache location, using only one tag entry. When all blocks in the super-block are compressible, a cache entry must be allocated for the first block, but the subsequent blocks do not require any extra cache space.

The address splitting of a conventional cache and a sectored/compressed cache are



(a) A standard cache entry



(b) A sectored cache entry

Figure 2.2 – Sectored cache layout.

shown in Figure 2.3. These two caches have different hash functions and, therefore, they generate different access patterns as follows.

- In a conventional cache layout, the address is divided into offset, set and tag (Figure 2.3a).
- In a compressed cache layout, an additional *block ID* field is interpreted differently depending on the compression status of the super-block (Figure 2.3b):
  - When the super-block is compressed, block ID field indexes the proper sub-block of the super-block.
  - When it is uncompressed, block ID is used for comparison against the cache tags, together with the tag field. Thus, a compressed cache can hold multiple uncompressed super-blocks with the same tag in the same set, with different valid sub-blocks (Figure 2.4).

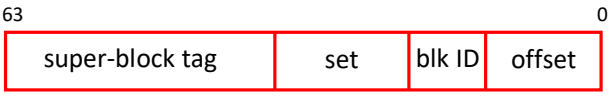
In compressed cache layouts, depending on the number of valid blocks in a super-block, we can have different compression factors (CF). For instance, if on average there are two valid blocks per super-block in the cache, the CF will be 2. In general, the maximum CF is the number of blocks compacted in a super-block.

Decoupled compressed cache (DCC)

Sardashti et al. [SW13] introduced decoupled compressed cache (DCC) that divides the data entry into eight 8-byte chunks. As it is shown in Figure 2.5, in DCC, two tag



(a) Conventional cache address mapping



(b) Sectored/compressed cache address mapping

Figure 2.3 – Conventional cache and compressed cache address mapping

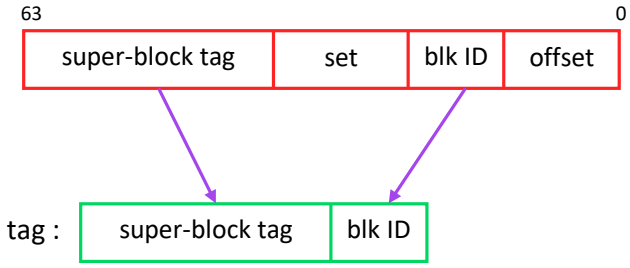


Figure 2.4 – The tag address of an uncompressed super-block in a compressed cache

entries can map to the same cache entry. Each entry in DCC contains a super-block tag, a back pointer entry and a data entry. Each back pointer or data entry is divided into eight 8-byte chunks. A super-block tag consists of coherency state bits (V for valid and I for invalid) and compressiblity bits (C for compressed and U for uncompressed) for each of the four sub-blocks. A back pointer entry is divided into eight elements, each has one bit for tag ID (the way number) and two bits for sub-block number. Therefore, for each element of a data entry, we can correlate an element in a back pointer entry that indicates to which tag and block this chunk corresponds.

DCC solves the issue of storing a block in contiguous chunks in [AW04a]. In DCC, due to the existence of the back pointer array, the chunks can be stored in a not consecutive order; therefore, the allocation in this layout is more flexible than [AW04a]. Nevertheless, the disadvantage of DCC is that it does not have a direct mapping between tag and data due to the back pointer array. This indirect mapping makes this layout more complex and may increase the access time.

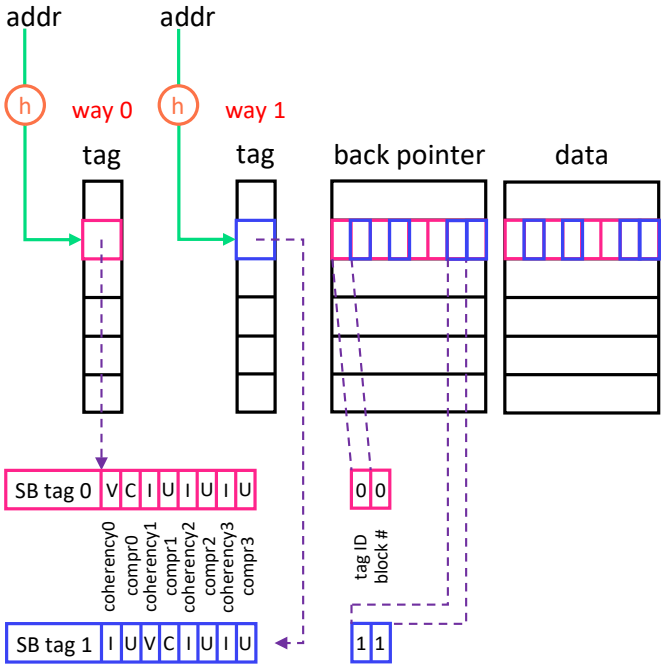


Figure 2.5 – DCC layout

Skewed compressed cache (SCC)

In order to avoid DCC drawback, the indirection level of back pointers, one can make the data location implicit based on the address. By eliminating the back pointer array, we need another approach to keep track of the compressed/uncompressed blocks and their correlated tags. Considering a set-associative cache, knowing which way contains compressed data and which contains uncompressed data is essential. Therefore, a method that calculates the way based on the address and the compression factor is required. Furthermore, we need to know the compression factor of a block from its address and way.

If the way is only correlated to the compression factor, a naive approach in a 2-way set-associative cache associated to way 0 to compressed data and way 1 to uncompressed data. However, in case all the data are compressible, only half of the cache is utilizable. Therefore, we need to take into account the address to compute the way in which the cache block resides.

The third downside of set-associative caches is conflict misses. If all ways of a set are occupied by valid blocks, a new block that is mapped to the same set is a conflict miss. Some solutions in order to decrease the number of conflict misses are to increase the

associativity by increasing the number of ways in the cache or the number of replacement candidates [SK10]. Nevertheless, these techniques increase the power consumption and the latency.

Skewed associative cache [SB93] tackles the aforementioned problems by exploiting different hash functions for different ways. Skewed compressed cache (SCC) [SSW14] is a compressed cache layout presented by Sardashti et al. [SSW14] that takes advantage of the skewed associative cache. The idea of their paper is to reduce conflict misses and also make use of the whole capacity in compressed caches by using different hash functions to map an address to different sets. Implementing various hash functions adds extra circuits to the system.

Figure 2.6 shows a 4-way SCC that can co-allocate up to 8 sub-blocks in a super-block. Furthermore, the tag entry contains a super-block tag and eight valid bits correlated to each sub-block.

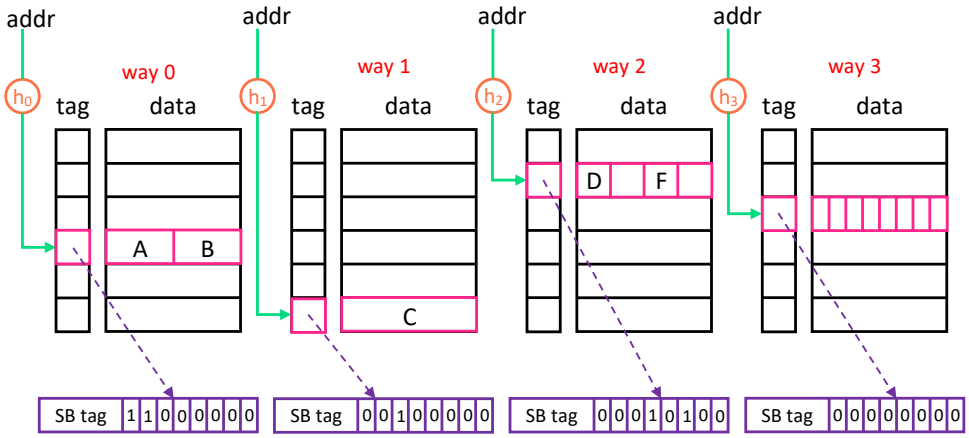


Figure 2.6 – SCC layout

In this configuration, the authors use a term called compression factor and the address to determine the set and the way of a block. Moreover, SCC assumes that a block can be compressed to 8 bytes, 16 bytes and 32 bytes. Therefore, it can allocate 8 sub-blocks in a super-block, which makes the block ID size being three bits in the address splitting of Figure 2.3b.

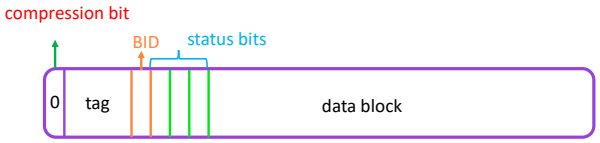
The compressibility of a data entry in SCC is implicit in the way that the block is stored in. In other words, for a given address X, if a block is compressible to 16B, it can only reside in way 0. However, the result of hash functions for different addresses are not the same; therefore, for address Y, if the block is compressible to 16B, it may be allocated

in another way other than way 0.

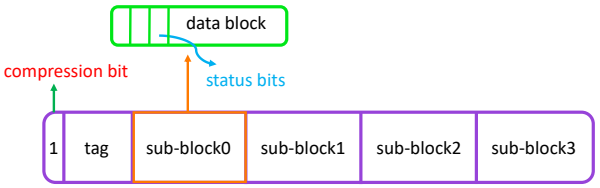
Compared to DCC, SCC reduces the metadata overhead by eliminating the back pointer array; therefore, there is a simple tag-data direct mapping in SCC.

Yet another compressed cache (YACC)

The SCC architecture is complex due to implementing different hash functions. Later, Sardashti et al. [SSW16] proposed yet another compressed cache (YACC) that reduces the complexity of SCC by using only one hash function for all the ways. As it is shown in Figure 2.7, YACC is a super-block-based layout.



(a) An uncompressed cache entry



(b) A compressed cache entry

Figure 2.7 – Super-blocks in YACC can be conventional or compressed depending on the compression bit

A super-block could have two layouts in YACC based on its compression bit. If the super-block is uncompressed, it has the compression bit as zero and contains only one sub-block. On the other hand, if the super-block is compressed, its compression bit is one, and it can hold up to four contiguous sub-blocks. Each sub-block has its own status bits.

In a compressed super-block in the YACC layout, the blocks are implicitly identified by their position inside the super-block. In an uncompressed super-block, we need to know which of the blocks is located in the super-block. To this end, the BID field of the super-block holds the block ID. However, in SCC, the super-block tag carries the valid bits of the cache blocks of the super-block (Figure 2.6).

As it was mentioned previously, in a compressed cache layout, we can have compressed or uncompressed entries. Suppose there are four neighbor blocks (A, B, C and D) aligned

on a super-block boundary, two of them are uncompressed (A and B) and the two others are compressible and co-allocatable (C and D). Figure 2.8 shows the compressed cache layout which maps the four blocks to the same set, identifying them by block ID. In this case, the cache contains:

— A compressed super-block consisting of sub-blocks C and D, that have the same tag and its valid bits are 0011.

— An uncompressed super-block, with the same tag as the aforementioned super-block, that contains only sub-block A, and valid bits of 1000.

— An uncompressed super-block with valid bits of 0100, showing that the super-block consists of only sub-block B.

All of these 3 super-blocks have the same tag and they will be allocated in different ways of the same set (they contain neighbor sub-blocks).

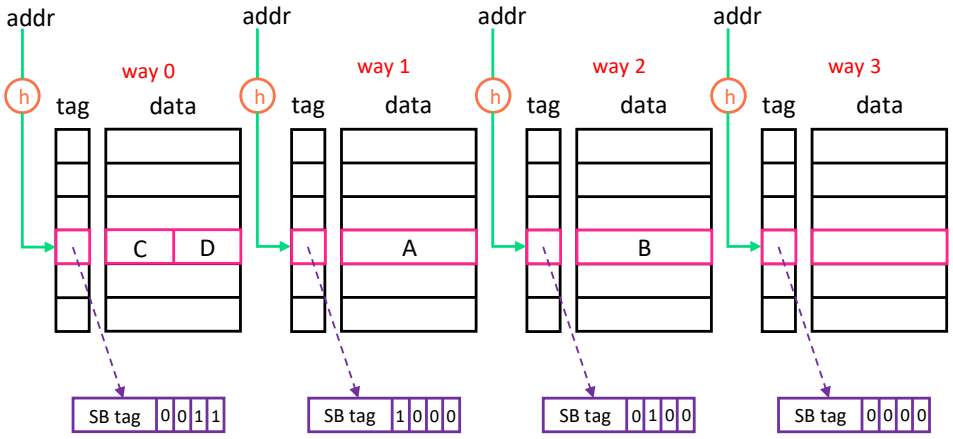


Figure 2.8 – YACC layout

Comparing to DCC, YACC does not have the additional overhead of the back pointer array in order to track the blocks. However, in DCC, a block that is compressed to 24 bytes can be stored in three chunks; whereas, in YACC, because of the fixed size of the compressed blocks (16, 32 or 64 bytes), this block needs a 32-byte storage. Furthermore, YACC has a simpler implementation than SCC, because in YACC all the ways use the same hash function for indexing.

Touché

Super-block-based cache layouts such as YACC and SCC reduce the tag overhead area comparing to the methods that double the size of the tag array [YZG00]. However, in these

layouts the compressed blocks, that are going to be compacted in one super-block, should be neighbors in the memory so that they can share a super-block tag. Therefore, these approaches have placement restrictions and they can only compact contiguous compressed blocks.

Touché, a recently proposed compression layout by Hong et al. [Hon+19], stores arbitrary compressed cache blocks in one data entry without the placement restrictions and tag overheads. The Touché framework consists of three units. The first component is the signature engine that takes the tag address and generates a 9-bit signature for each compressed block. In this technique, a tag entry referring to compressed entries can store up to three 9-bit signatures. Thus, the cache can allocate up to three non-contiguous compressed blocks next to each other in one data entry.

Nevertheless, the shortened signatures may cause false positive tag matches; hence, in order to avoid this drawback, the authors introduced the tag appended data (TADA) mechanism which stores the full tag addresses of the compressed blocks. On a read access, a 9-bit signature is generated for the address. After that, the cache compares the address' signature with the three signatures in the tag entry. If none of the signatures in the tag entry matches the address' signature, it is guaranteed that the block is not there. If there is a signature match, the cache checks the TADA unit in order to verify the full tag address.

The last component in Touché is the super-block marker mechanism that generates a marker to notify that there is a super-block, containing neighbour compressed blocks, in the cache entry.

Touché increases the cache capacity and mitigates the tag area overhead at the cost of complex tag-checking and complex management of the exceptional false positives.

2.1.2 Cache compression methods

Compressed caches use compression algorithms to compress a data block and increase the effective capacity of the cache. However, there are some global challenges in cache compression schemes:

— Compression algorithms work on a very small granularity that is a 64-byte block.
— These techniques take advantage of data redundancy, that depends on the program. Different applications have different redundancies; therefore, applying a single compression algorithm may not compress data in all applications.

— Compressing a cache level increases its capacity; however, because of the compression and decompression circuits, it adds complexity to the system.
— They increase the latency due to the existence of a decompression unit in the critical path, on a cache read.
— Compression schemes should be compatible with compaction methods. For example, the DCC layout is able to allocate blocks that are compressible to any multiples of 8 due to dividing the data entry into 8B chunks; whereas, SCC or YACC can only allocate blocks that are compressible to either 8B, 16B or 32B, or that are not compressible.

This section introduces some data compression techniques that are applicable in the memory hierarchy.

Zero value compression

Ekman et al. [ES05] showed that 30% of all cache blocks and 55% of all bytes in memory contain zeros. Therefore, the memory wastes a part of its capacity storing zero data.

Later, zero-content augmented cache (ZCA) was introduced by Dusser et al. [DPS09]. ZCA has a special cache for memorizing null data, the zero-content cache (ZC), in addition to the conventional cache. In this technique, zero data is implicitly stored by merely adding its tag to the ZC and updating the validity bit.

Islam et al. [IS09] observed that, on average, 18% of the total loads are zero loads which are accessing zero value locations in the memory. Such loads can be executed without accessing the cache hierarchy, leading to a speedup and also a reduction in energy consumption of the system.

Frequent value compression (FVC)

Apart from spatial and temporal locality, there is another notion of locality. It is possible for a previously accessed value to be accessed again in the future; this facet of locality is called value locality [LWS96; AS12].

Frequent value compression (FVC) [YG02] is a compression technique that relies on frequent-value locality, first proposed by Zhang et al. [ZYG00]. FVC takes into account that some values appear very frequently in memory locations; thus, they can be compressed to fewer number of bits. Since frequent values vary from one application to another, employing useful algorithms to find the frequent values is of essence. FVC has a

table to record these frequent values that is used by the whole cache; however, as the table gets larger, the FVC access time increases.

Frequent pattern compression (FPC)

Frequent pattern compression (FPC) is a significance-based compression scheme, proposed by Alameldeen et al. [AW04b], which is based on narrow-value locality. FPC utilizes the fact that cache lines can consist of common word patterns and they can be stored using a small number of bits. FPC compresses data on a word-by-word basis and it replaces the frequent data patterns with specific encoding; therefore, it stores words using 3-bit prefixes and some auxiliary data.

Table 2.1 shows the encoding of patterns used in FPC, which are selected based on their high frequency in different benchmarks. Moreover, examples of data in uncompressed and compressed format, taking advantage of FPC's encoding, are shown in this table. The zero run is a special case, which compressed multiple words under a single pattern; the data in the compressed format is addressed with the 0b000 prefix followed by 3 bits: 000 would show there is a zero run containing one zero word, 001 indicates a zero run containing two zero words, 010 means that there are three zero words in the zero run pattern and so on. For instance, the zero run pattern in this table assumes a cache line consisting of 24 bytes of zeros; hence, the compressed data is shown as 000 as the prefix pattern followed by 010.

Table 2.1 – Frequent pattern encoding with examples

Prefix	Pattern encoded	Uncompressed data	Compressed data
000	zero run	3*(0x00000000)	(0b000)0b010
001	4-bit sign-extended	0xFFFFFFFA	(0b001)0xA
010	1-byte sign-extended	0x0000004C	(0b010)0x4C
011	half-word sign-extended	0xFFFF8C6B	(0b011)0x8C6B
100	half-word padded with zero	0x000023AB	(0b100)0x23AB
101	sign-extended half-words	0x0012FFA6	(0b101)12A6
110	repeated byte	0x4A4A4A4A	(0b110)4A
111	uncompressed	0xA1B2C3D4	(0b111)0xA1B2C3D4

The authors evaluated the method using SPEC CPU2000 [Ben00] and several commercial workloads from Wisconsin Commercial Workload Suite [Ala+03]. The compression factor of FPC is 1.5X, on average, on these benchmark suits. Moreover, the authors claimed that the decompression latency is 5 processor cycles.

Cache packer (C-PACK)

C-pack, proposed by Chen et al. [Che+09], is another significance-based compression technique that treats data blocks in 4-byte words. It takes advantage of both narrow-value locality and dictionary encoding on a word-by-word basis. Just like FPC, C-pack detects the frequent patterns in a data stream and besides, it employs a dictionary for repeated patterns. Thus, for compressing a data, C-pack checks whether each word matches a dictionary element (either completely or partially) or one of the predefined patterns.

Table 2.2 – C-pack pattern encoding. z: zero byte, x: byte doesn't match, m: byte matches, i: index of the dictionary element.

Prefix	Pattern	Output data	Output length (bits)
00	zzzz	(0b00)	2
01	xxxx	(0b01)xxxx	34
10	mmmm	(0b10)i	6
1100	mmxx	(0b1100)ixx	24
1101	zzzx	(0b1101)x	12
1110	mmmx	(0b1110)ix	16

Table 2.2 shows the pattern encodings for C-pack. If a 4-byte word matches the "zzzz" or "zzzx" patterns, it is not added to the dictionary; however, for the rest of the patterns we always update the dictionary, which is a FIFO of at most 16 entries. When compressing, the dictionary is created from scratch while parsing the cache line, then it is discarded. On decompressions it is reconstructed using the compressed data.

Figure 2.9 shows an example of a cache line, corresponding dictionary and the compressed data using C-pack. The first word of the cache line does not match any dictionary element (the dictionary is empty in the initialization); therefore, it is added to the dictionary and it is uncompressible. The same happens to the second word of the cache line. The third word matches the pattern "zzzx"; thus, we do not add it to the dictionary but we can store it as a compressed data with a 1101 pattern as the prefix, followed by the unmatched byte that is AB. The last word of the cache line partially matches the dictionary element "12345678" under the pattern "mmmx"; hence, we use 1110 as the prefix, followed by a pointer to index the dictionary element (0000) and the unmatched byte. Furthermore, this word is added to the dictionary to increase the likelihood of compression of further words in this cache line.

C-pack has a compression factor of 1.6X on multiple workloads from SPEC CPU2000 [Ben00] and Mediabench [LPM97] suites. However, C-pack compresses or decompresses 8

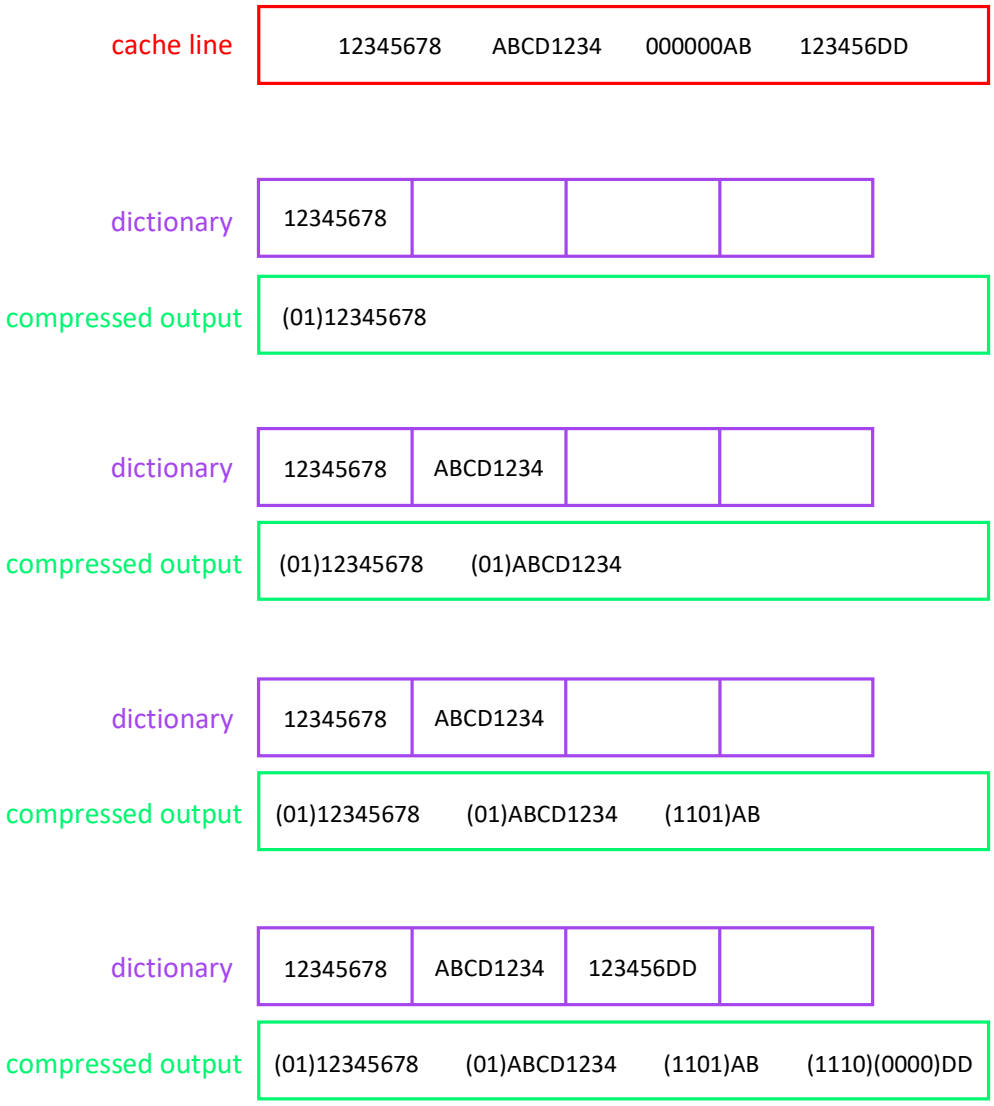


Figure 2.9 – C-pack compression example

bytes per cycle which makes the decompression latency 8 cycles that is relatively high. Recently, Alameldeen et al.[AA18] proposed an opportunistic compression algorithm that exploits some frequent patterns and a dictionary, which limits the C-pack dictionary to the last two seen double words. In this compression technique, by employing frequent patterns and a limited dictionary, the decompression latency is reduced to two cycles which is less than the decompression latency in C-pack and FPC algorithms.

Base-Delta-Immediate compression (BDI)

Pekhimenko et al. [Pek+12] introduced an approach for compressing data that have low dynamic range. Based on their experiments, many cache blocks have a small difference between different values. Therefore, cache blocks can be stored as a base value and an array of deltas. The deltas represent the differences between the data values and the base, which is shared among all the deltas.

Furthermore, the authors showed that it is beneficial to have two bases in this compression technique. The first value in the cache block as the first base and zero value as the second base. Zero is a good choice for narrow values (small values and near to zero) which can be found in a cache block frequently.

BDI outperforms previous compression algorithms in terms of decompression latency. It provides a single-cycle decompression latency. Furthermore, the compression factor in BDI is 1.53X, using SPEC CPU2006 suite [Ben06], which is slightly better than FPC but lower than C-pack.

Dictionary sharing algorithm (DISH)

Dictionary Sharing (DISH), presented by Panda et al. [PS16], is a compression technique that takes advantage of the YACC layout. A common compression dictionary is shared among all sub-blocks of each super-block. If the contiguous blocks are compressible with the same dictionary elements, they can be compacted into one super-block. However, it may happen that although two or more blocks can be compressed independently of each other, they cannot be compacted together with the same dictionary. In that case, they will be distributed into multiple compressed super-blocks.

DISH treats a 64-byte cache block as 16 4-byte chunks. These chunks can be repeated in a cache block; For example, a block may contain 0x22138F07 in multiple chunks. DISH keeps each distinct chunk value in a cache block in an 8-entry dictionary, and replaces each of the 16 chunks with a 3-bit pointer to the corresponding dictionary entry (Figure

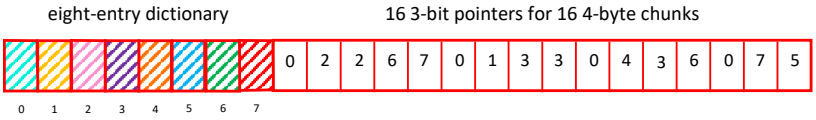
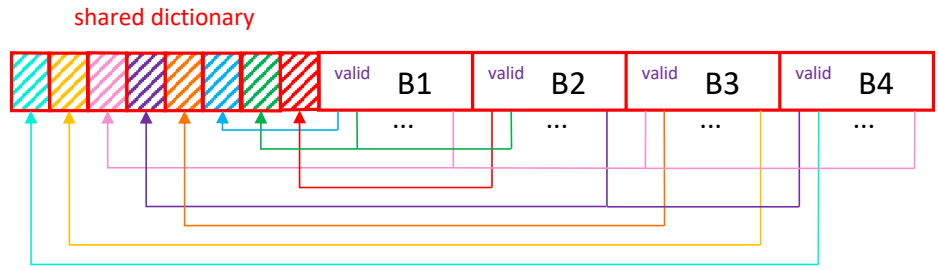


Figure 2.10 – A compressed cache block using DISH

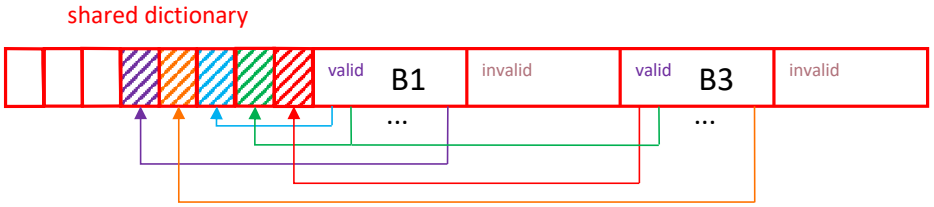
2.10). Each dictionary entry has a valid bit which is set when the entry contains a 4-byte chunk. If DISH does not find any invalid dictionary element to allocate a distinct 4-byte chunk, it considers the cache block as uncompressed. DISH also has another encoding scheme that is similar to the first one. It is possible that a block is compressible with both DISH schemes. In this case, a set-dueling [Qur+07] mechanism is applied to choose between the two schemes.

The key idea of DISH is to share a dictionary not only within a block, but also among multiple sub-blocks by taking advantage of the YACC layout. If the contiguous blocks are compressible with the same dictionary elements, they can be compacted into one super-block (Figure 2.11a). DISH compacts between one to four sub-blocks in a super-block. Suppose that block B1 is compressible with one of the DISH schemes (B1 is a valid sub-block in Figure 2.11a). When there is an access to block B2 (B1's neighbour block), if B2 is compressible with the same compression scheme and the same dictionary elements as B1, these two blocks can be compacted in the same super-block entry. Consequently, the cache entry contains two sets of pointers. The same can happen to other neighbour blocks of B1. DISH can compact up to four blocks in one super-block; hence, it has maximum compression factor of 4. Figure 2.11a shows a fully compressed super-block and the sub-blocks share the same dictionary. In Figure 2.11b there are only two valid sub-blocks that are co-allocatable.

DISH minimizes hardware complexity and decompression latency to 1 cycle. Besides, it exceeds previous works, such as BDI and C-pack, in terms of compression factor, which ranges from 1.7X to 2.3X, on SPEC CPU2006 [Spr07] and CRONO benchmark suites [Ahm+15]. It should be pointed out that DISH uses a compaction technique that allows compacting up to four blocks in a data entry; whereas, the other compression methods employ compaction strategies which co-allocate up to two blocks in a data entry.



(a) A super-block with four compressed sub-blocks



(b) A super-block with two compressed sub-blocks

Figure 2.11 – Co-allocation of multiple sub-blocks in a super-block using DISH

2.2 Hardware prefetching

One of the important techniques to reduce the memory latency, that is a bottleneck in systems, is prefetching the data before it is needed [Mit16]. Prefetching can be carried out by predicting the memory access pattern of the program. If prefetches are accurate and early enough, it reduces or hides the miss latency.

The misses can be classified as capacity, compulsory and conflict:

— In large scale applications, the program working set is larger than the cache size; therefore, the cache can not contain all the blocks needed by the application. The capacity miss happens when the requested data is not in the cache due to the small size of the cache.

— The first request to a cache block is always a miss, which is called compulsory miss, and it happens regardless of the cache design. In this situation, a request is sent to the memory, the block is read from the memory and is updated in the cache.

— Conflict misses occur when different addresses map to the same set and evict blocks that are still useful in the program.

Hardware prefetching is a strategy that can reduce capacity and compulsory cache misses and as a consequence, it diminishes miss rate and miss latency in the memory system. Prefetching is performed in either software or hardware [BC07; KSH14]. In this research, we consider only hardware prefetching.

Prefetching useless data wastes resources such as memory bandwidth, cache or prefetch buffer space and also it causes an increase in energy consumption. Furthermore, sending prefetch requests on top of demand requests may delay demand misses to be serviced. Prefetching too early brings data that might not be used before they get evicted from the storage. Prefetching too late might not hide the whole miss latency. Hence, accurate and timely prediction of addresses to prefetch is crucial in computer systems. Often prefetching can be made more timely by making the prefetcher more aggressive, i.e, try to stay far ahead of the processor's access stream, yet still one has to be conservative enough to keep it from generating early prefetches.

Another concern regarding hardware data prefetching is where to place the prefetched data. If they are stored in cache, it will be a simple design and there is no need for separate buffers; however, it can evict useful demand data and cause cache pollution. Jouppi [Jou90] introduced the idea of *stream buffer* which is a storage of n entries that is separated from L1 cache and it keeps n subsequent addresses. Therefore, next time a miss happens on the cache, the stream buffer will be accessed to see if the block was already prefetched before. The blocks in the stream buffer are not allocated in the cache in order to avoid cache pollution. Nevertheless, this approach requires more complex memory system design decisions, such as:

— Where to place the prefetch buffer?
— When to access the prefetch buffer (parallel or serial with cache accesses)?
— When to move data from the prefetch buffer to the cache?
— How to size the prefetch buffer?
— How to keep the prefetch buffer coherent with the cache?

A three-level cache hierarchy integrated with a hardware prefetcher is shown in Figure 2.12. In this figure, the prefetcher is implemented on LLC; however, prefetching unit can be applied at any level in the memory hierarchy. The prefetcher unit has a prefetch queue to store the prefetch addresses. The prefetch queue is different from the prefetch buffer, a storage outside the cache, that was mentioned previously. Before sending the requests to the prefetch queue, the prefetcher checks the cache to see if the block exists there already. If the block is not in the cache, then the prefetch request is stored in the prefetch queue.

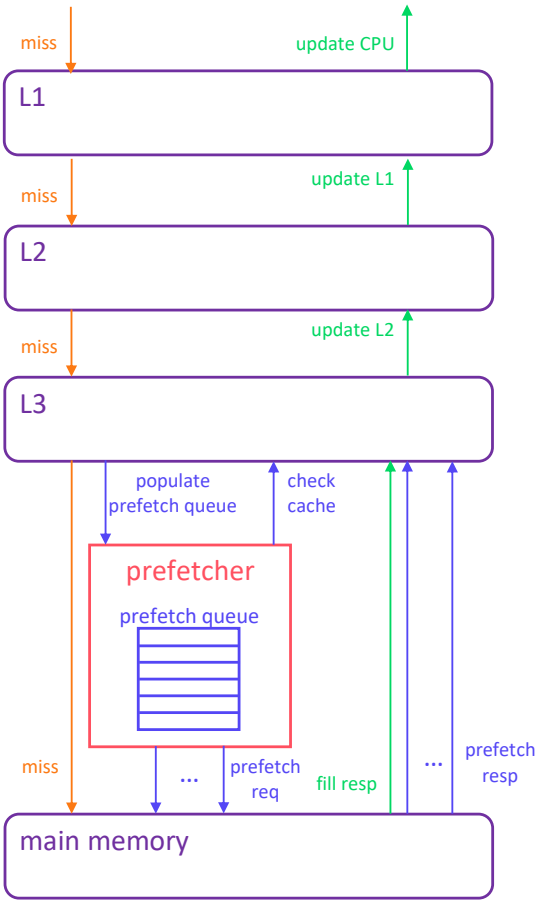


Figure 2.12 – Cache hierarchy diagram with prefetcher unit on the LLC

Whenever there is no demand request from higher cache levels to be serviced, the system services one of the prefetch requests that is stored in the prefetch queue. So it goes to the memory, fetch the data and updates the cache level that requested the data. It is worth noting that the prefetcher can be triggered on every access (miss or hit). Furthermore, prefetching does not perform across the pages, which means if the prefetcher requests a block from another page, it is abandoned.

Prefetchers have various implementation overheads. For instance, there are simple prefetchers, such as Nextline, that do not require any additional complex units; however, their coverage and accuracy is not high enough. On the contrary, there are more accurate, but sophisticated, prefetching techniques that add extra metadata to the system [FF07; LFF01]. In the remaining of this section, some of the prefetching techniques are introduced.

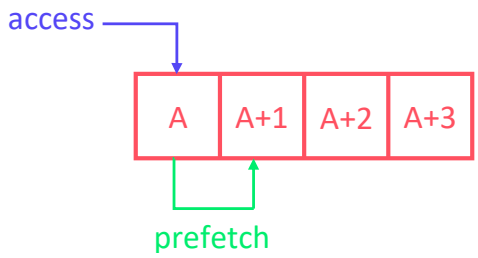


Figure 2.13 – Illustration of a Nextline prefetcher

2.2.1 Nextline prefetcher

Prefetching blocks in consecutive addresses, sequential data prefetching, was introduced by Smith in 1978 [Smi78]. Nextline is the simplest form of a sequential hardware prefetcher. Nextline is an immediate prefetcher, since it performs a prefetch as soon as it is given an input reference address. As it is illustrated in Figure 2.13, if block A is accessed either by a hit or a miss, a prefetch request is sent to the memory for block A+1. Later, if block A+1 is accessed, a prefetch request for block A+2 will be sent to the prefetch queue, and so on.

Nextline prefetcher is simple to implement and it does not need any sophisticated pattern detection. Moreover, it works well for sequential/streaming access patterns. Nonetheless, there are some drawbacks in this approach. Prefetching irregular access patterns is difficult and inaccurate, and Nextline prefetcher can waste bandwidth with irregular patterns. Furthermore, the program may traverse memory from higher to lower addresses and needs to prefetch data that are physically placed in the previous blocks. Another disadvantage of Nextline prefetcher is that it may cause lots of late prefetches due to demanding only for the next block.

2.2.2 Stream prefetcher

Instead of prefetching the next block, if we can prefetch a sequence of addresses, this prefetcher is referred to as a stream prefetcher. In 1990, Jouppi [Jou90] proposed a stream prefetcher that uses stream buffers (Section 2.2) in order to avoid pollution in L1 cache. In a stream prefetching technique, if a sequence of addresses of A, A+1, A+2 is accessed, A+3, A+4, ..., A+n are prefetched; where n is the number of prefetches.

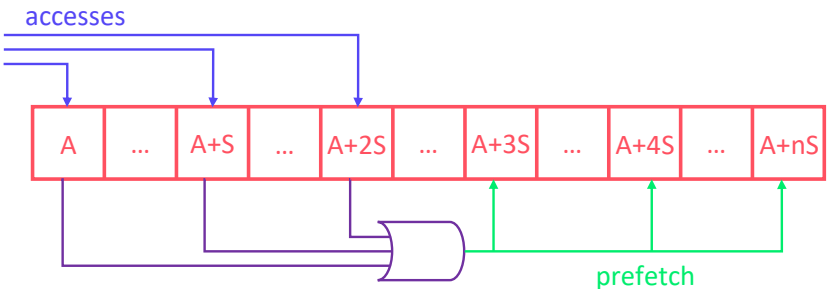


Figure 2.14 – Illustration of a stride prefetcher

2.2.3 Stride prefetcher

In 1991, sequential prefetchers were extended to be able to prefetch strided accesses, by Baer et al. [BC91]. In a stride prefetcher, if we detect a sequence of accessing to address A, A+S and A+2*S, then the stride prefetcher sends prefetch requests for A+3*S, A+4*S, ..., A+n*S; where S is the stride and n is the number of prefetches that should be performed (Figure 2.14).

Stride and stream prefetchers are confirmation-based prefetchers, because they do not send prefetch requests unless they detect a stream of accesses as a confirmation to start prefetching.

Stride prefetcher works based on a table which is indexed by the program counter (PC). The reason of having a PC-based table is that the stride prefetcher tries to keep track of the addresses that are from the same load accesses, and not random accesses. Each request carries its PC through the memory hierarchy. Each table entry contains the last address that was accessed by this load instruction, a stride and a confidence related to the stride. Stride prefetcher records the distance between the memory addresses referenced by a load instruction, i.e., stride of the load, as well as the last address referenced by the load. Stride prefetcher decides to perform prefetches based on the stride confidence of the table entry, which shows how reliable this stride is. The confidence is a saturating counter that is incremented each time we observe the same stride as the last one. If the confidence of the computed stride is less than a threshold, prefetch generation is aborted. In other words, stride prefetcher does not generate prefetches with low stride confidence. It issues a prefetch request only if there is a certain confidence that the prefetch will be useful.

Prefetcher aggressiveness is an important metric in the stride prefetcher. This metric can be defined by two parameters: distance and degree. Prefetch distance is how far

ahead of the demand stream we are prefetching. Prefetch degree determines as how many prefetches per demand access are going to be carried out. As these parameters increase, the aggressiveness of the prefetcher increases.

Stride prefetcher prefetches several blocks ahead; therefore, unlike Nextline prefetcher, the blocks do not arrive late to the cache. However, if stride prefetcher is too aggressive, it consumes energy and bandwidth. Furthermore, it may cause pollution in the cache by bringing too many useless data and hurting the performance.

The stride prefetching can be activated in any of the cache levels in the memory hierarchy. The authors in [BC91], trigger the stride prefetcher on L1 cache. It is noteworthy that the accesses to L1 are word-based, that means the we can access one word of a cache block at each time. This makes the strides, calculated in L1, multiples of a word size.

However, in this thesis, we employ the prefetcher on LLC. Unlike exploiting prefetcher on L1 cache, the accesses to L2 and L3 are block-based, which indicates that the request is accessing the whole cache block, not only one word of the cache block. Therefore, the stride that is calculated on L2 or LLC is a multiple of cache block size. Eventually, the addresses that access L1 and L2/LLC are different due to the word-based and block-based behaviour of the caches. Furthermore, by applying the prefetcher on LLC, we decide to carry the PC with the request from the CPU to LLC.

Another difference in employing the prefetcher in L1 or LLC is the number of accesses. L1 cache receives more accesses; nonetheless, the accesses to LLC are the ones that were misses on L1 and L2. Thus, the prefetcher is triggered less when it is exploited in LLC.

2.2.4 Best-offset prefetcher

Offset prefetcher is a generalization of the Nextline prefetching technique. Moreover, unlike stream/stride prefetcher, an offset prefetcher does not detect streams [Mic16]. When line X is requested by the processor, offset prefetcher prefetches line X+D, where D is the prefetch offset.

As it previously stated in Section 2.2.1, Nextline prefetcher may carry out lots of late prefetches due to prefetching the block right after the accessed block. Timeliness is an essential factor in prefetching and it defines as how early a block is prefetched comparing to when it is actually referenced. Michaud [Mic16] proposed best offset prefetcher (BO prefetcher) that takes into account timeliness and tries to find the best prefetch offset. D is a good offset for line X if line X-D have been accessed just recently, and not too recently. In other words, the time between accessing lines X-D and X must be greater

than the latency of a prefetching request.

The BO prefetcher has a recent request (RR) table, which records the base address of prefetch requests (if the prefetched line is X+D, the base address is X). Besides, this prefetcher has a learning phase to select the proper offset among a predefined offset list. In the offset list, there is a score associated with each offset. This offset list, and the scores are getting updated in every learning phase. During the learning phase, each offset d from the offset list is chosen and X-d is calculated (X is the requested line). If X-d hits the RR table, the score of offset d gets incremented. When all the offsets in the list are tested, the best offset for prefetching is chosen based on the maximum score.

The BO prefetching unit is applied on L2 cache due to inaccurate prefetches that my happen on L1 cache and causing cache pollution. Although the best-offset prefetcher prefetches the data that are going to be arrived on time, it does not always perform optimal and it can not always find the best offset.

2.2.5 Adaptive prefetchers

As prefetching is advantageous in a memory system, it also has some downsides, and it may degrade performance due to cache pollution and memory traffic caused by unnecessary prefetches. Hence, adjusting and triggering hardware prefetcher at the appropriate time during the execution of the program is influential in the system. There are some metrics that estimate prefetcher effectiveness [Sri+07]:

— **Prefetch usefulness** is defined by number of useful prefetches divided by number of total prefetches. If the cache accesses a block that was prefetched, before it gets evicted, this prefetch is counted as a useful one. This means that the memory address prediction of the prefetcher is correct and it is accurate.

— **Prefetch lateness** measures the timeliness of prefetch requests. If there is a prefetch request to the main memory and the data has not been brought back to the cache; and meanwhile, another demand request needs the prefetched data, this prefetch is called late. That is to say, there is an access to a block that is still being prefetched. This metric is calculated as number of late prefetches divided by number of useful prefetches.

— **Cache pollution caused by prefetcher** occurs when a demand miss happens on a block that was evicted by a prefetch request beforehand. Thus, prefetching can cause cache pollution, and as a result performance degradation, by evicting useful data from the cache to make space for the prefetched data.

There are multiple prefetching approaches that consider these metrics to trigger the prefetcher dynamically in the program. Dahlgren et al. [DDS95] proposed an adaptive sequential prefetcher that takes into account the prefetch usefulness. If the prefetches are useful, the prefetch distance increases; otherwise, it is decreased. Wu et al [Wu+11] introduced a prefetching technique that switches the prefetcher on and off dynamically. This approach avoids cache pollution and also keep the data that can not be prefetched easily in the cache.

Feedback directed prefetching (FDP), presented by Srinath et al. [Sri+07], uses the three metrics (usefulness, lateness and pollution) as inputs to adjust the aggressiveness of the prefetcher dynamically, from very conservative to very aggressive with a stream prefetcher [Jou90; BC91] as the baseline. In this configuration, a very conservative stream prefetcher has a prefetch distance of 4 and degree of 1; while the very aggressive stream prefetcher is based on a distance of 64 and degree of 4. FDP selects the best prefetcher dynamically, depending on applications characteristics, based on usefulness, timeliness and cache pollution. However, there are other influential metrics that are concerned for prefetching, such as memory contention and the available bandwidth.

2.3 Cache compression and hardware prefetching interaction

Some related works discussed about taking advantage of hardware prefetching and compressed cache in a system side by side, and not cooperatively.

Raghavendra et al. [RPM16] introduced Prefetched Blocks Compaction (PBC) which splits the cache into two parts; so, one part stores non-compacted blocks (C1) and the other part has compacted blocks (C2). C1 uses address mapping of an uncompressed cache (Figure 2.3a) and C2 accesses the cache blocks using compressed cache address mapping (Figure 2.3b). This work takes advantage of a compaction unit, which keeps the compactable blocks in each entry of a buffer. Each prefetched block goes through the compaction buffer to check if it can be compacted with any entries and be allocated in C2 part of the cache. In case of non-compactability of the prefetched block, a way balancer decides whether it should be allocated in C1 or C2, based on cache space available in both parts.

The goal of Synergistic cache Compression and Prefetching (SCP), proposed by Patel et al. [PHM15], is to improve data streaming prefetchers, by decreasing the off-chip mem-

ory accesses. Some stream prefetchers have high complexities and they have large storage tables and buffers because of saving a part of the accesses as the history for predicting future prefetches [Som+09]. Owing to the large overhead of the tables and buffers, and limited size of the caches, designers need to to implement them in DRAM, which increases the off-chip bandwidth utilization due to extra DRAM accesses. Therefore, SCP employs cache compression to reduce the number of off-chip memory accesses. Using the extra space gained by compression, SCP can implement the storage arrays that the stream prefetchers require. Eventually, the prefetching mechanism is performed entirely in the cache level and it improves the performance of data streaming prefetchers.

2.4 Methodology

2.4.1 Simulation platform

In order to implement compression and prefetching mechanisms and propose our idea, we need to evaluate different configurations. These evaluations and analyses can not be carried out on an actual processor; hence, we employ a simulator to model the system and be able to modify the memory hierarchy.

There are multiple computer architecture simulators with different simulation techniques [AS16], such as Simics [Mag+02], gem5 [Bin+11] and ZSim [SK13]. The implementation framework of our study is gem5, one of the most popular computer architecture simulators. The gem5 simulator is a modular platform for computer-system architecture research, encompassing system-level architecture as well as processor microarchitecture [].

There are two types of execution mode in gem5:

— System-call Emulation (SE) that doesn't need to model devices or the operating system.

— Full System (FS) that executes user level and kernel level instructions and models a complete system including operating system and devices.

Moreover, this simulator models processor cores, memory hierarchy and I/O systems. It takes advantage of the detailed CPU modeling in M5 [Bin+06] and the detailed memory hierarchy from GEMS [Mar+05]. Gem5 also supports multiple CPU models and instruction set architectures (ISA) such as ARM, X86, ALPHA, etc.

In this work, the simulations are performed on full system mode in gem5 running the Linux 3.16 kernel for 64-bit ARM.

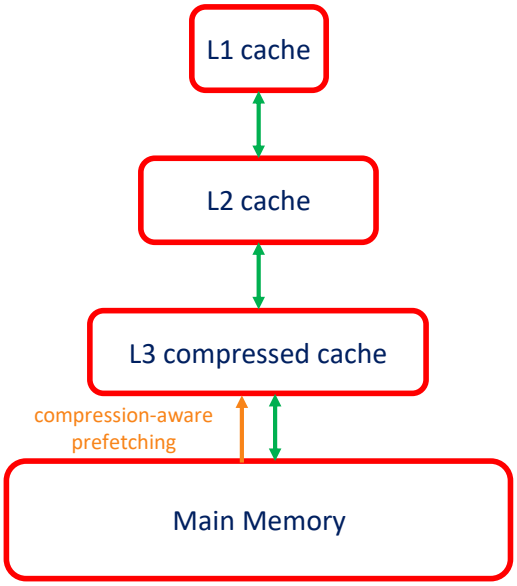


Figure 2.15 – Three-level cache hierarchy with compression-aware prefetching

Table 2.3 – Simulation parameters

Processor	ARMv8, 6-issue out-of-order at 4GHz
L1 cache	32kB, 4-way, 4 cycles
L2 cache	256kB, 8-way, 11 cycles
L3 cache	1MB, 16-way, 34 cycles
Cache line size	64-byte
Replacement policy	LRU
DRAM	DDR4 2400 MHz, bandwidth 12.8GB/s

A system configuration with a three-level cache hierarchy is shown in Figure 2.15. We have implemented the compression and prefetching mechanism in LLC and prefetching compressible blocks is done from memory to LLC. All the evaluations in the following sections are done on LLC.

Table 2.3 shows our configuration parameters in the memory hierarchy. A cache slice of 1MB is considered, corresponding to the L3 cache share of a single core. Caches are non-inclusive, non-exclusive and have a write-back policy. We implemented the DISH algorithm [PS16] as the compression method and YACC [SSW16] as the compressed cache layout for our framework, in LLC.

2.4.2 Metrics of interest

There are several metrics to evaluate cache compression and hardware prefetching. However, finding a trade-off among these metrics is essential.

Some metrics to assess cache compression are as follows:

— Effective cache utilization specifies the number of valid blocks in the cache.
— Compression factor is defined as the average number of valid sub-blocks in a valid super-block.
— Compression and decompression latency of a a compression algorithm.
— Complexity and the overhead of compression and decompression units.

To obtain the best performance in a compressed cache, we require a trade-off among complexity, latency and compression factor. For instance, some techniques exploit a low-overhead algorithm with low compression factor [Pek+12; AW04a]. On the contrary, IBM introduced a memory compression algorithm with a high complexity but resulting in a high compression factor [Tre+01].

In order to evaluate prefetchers effectiveness, we employ following metrics:

— Prefetch usefulness shows how often the prefetches are going to be used before they got evicted from the cache.
— Cache pollution generated by prefetch requests which can be observed by extra misses occur on LLC due to prefetching.

To measure the total performance of the proposed approaches, these metrics are considered:

— Instruction per cycle (IPC) that shows how many instructions the CPU can conduct in each cycle.
— Misses per kilo instruction (MPKI) showing the number of LLC misses per thousand instructions.
— Bytes per kilo instruction (BPKI) indicates the memory traffic in a configuration for measuring bandwidth consumption.

The evaluations of this work are done across 27 workloads. We use SPEC CPU 2006 benchmarks. The statistics for the benchmarks are collected for 100M simulated instructions after a warm-up period of 50M instructions for 15 different execution snapshots.

2.5 Summary

This chapter reviews several compressed cache layouts, such as DCC, SCC and YACC and presents multiple cache compression algorithms, including FVC, FPC, C-pack, BDI and DISH. It then explains how hardware prefetching is beneficial in a memory hierarchy and it can cause performance degradation, simultaneously. Furthermore, the chapter reviews the previous works on the interactions between compression and prefetching in the state-of-the-art. The methodology, implementation framework and the evaluation metrics are presented in the last section of this chapter.

In the next chapter, we will study cache compression characterization, using DISH algorithm and evaluate the potentials for prefetching in a compressed cache. Moreover, we will proposed a compression predictor that predicts which blocks are compressible. If a block is compressible, we may have the chance that its neighbors are compressible too; therefore, prefetching them can avoid the future misses that may happen. Finally, taking advantage of the compression predictor, we will propose a hardware prefetcher that is adopted to compressed cache layouts.

Chapter 3

Compressed cache layout aware prefetching (CLAP)

Yet Another Compressed Cache (YACC) [SSW16] and Skewed Compressed Cache (SCC) [SSW14], which we introduced in the previous chapter, are cost-effective compressed cache layouts. The idea of these layouts is to compact the compressible contiguous blocks that are neighbors in the memory with low area overhead in the cache. In the remainder of this thesis, YACC layout will be used through the study.

This chapter proposes a prefetching technique that takes into account the compressed cache layout. As previously mentioned, prefetching is not always beneficial in a system and it can cause pollution. Prefetching contiguous blocks, which can be compressed and compacted together with the requested block on a miss access, is practical in LLC. Prefetched blocks that share storage with existing blocks do not need to evict a valid existing entry; therefore, it avoids cache pollution.

In order to decide which blocks are candidates for prefetching, we propose a compression predictor. It is necessary to highlight the distinction between compression and compaction. The "compression" predictor only predicts the compressibility of a block, whereas the "compaction" predictor predicts if a block is compressible and also compactable with the neighbor blocks in one super-block. If a block is compactable with other blocks in a super-block, we say the block is compactable or co-allocatable.

Finally, we present a unified system based on the compression predictor and prefetching to avoid polluting the cache.

This chapter addresses the following questions:

— Are compressed caches subject to underutilization?

— How can we create a synergy between compressed cache and prefetching?

— What are the potentials for prefetching in compressed cache?

— Which blocks are candidates for prefetching?

— Is compression or compaction a better indicator for prefetching?

— How can we leverage sector-based compressed cache layouts such as SCC or YACC to create a synergy between compressed cache and prefetching?
— How can we utilize the compression predictor in this layout?

The contributions of this chapter have been published in Compas national conference [CC19] and SBAC-PAD international conference [CCS19].

3.1 Analyzing potentials for prefetching

In this section, we explore opportunities for creating a synergy between compressed cache and prefetching.

3.1.1 Characterization of compressed cache

One of cache compression techniques described in Chapter 2 was DISH algorithm. In order to evaluate a compressed cache, we have implemented the DISH compression algorithm in the LLC within the gem5 [Bin+11]. The parameters shown in Table 2.3 are used for the analysis of our experiments in the thesis.

Figure 3.2 illustrates the reduction in number of LLC misses for the 27 benchmarks. In this figure, we consider an uncompressed cache (conventional cache), a compressed cache (using DISH) and three other configurations:

— 2Xbaseline: a sectored cache with two sub-blocks in each super-block. It has a capacity of 2MB in LLC which corresponds to a hypothetical compressed cache with a compression factor of 2X.
— 4Xbaseline: a sectored cache with four sub-blocks per super-block. The LLC size of this configuration is 4MB corresponding to a compressed cache with 4X as the compression factor.
— Baseline-modified: another configuration of a 1MB conventional cache with the same address mapping and hash functions as the compressed cache. In this configuration, we consider block ID bits of the compressed cache layout as a part of the tag (Figure 3.1). In other words, baseline-modified is a compressed cache with uncompressed data (compression factor of 1X)

In these settings, a 1MB compressed cache can approach a 2MB conventional cache (2Xbaseline), in terms of MPKI, in some of the benchmarks. Cache compression yields only modest improvements or may even cause more misses due to having consecutive

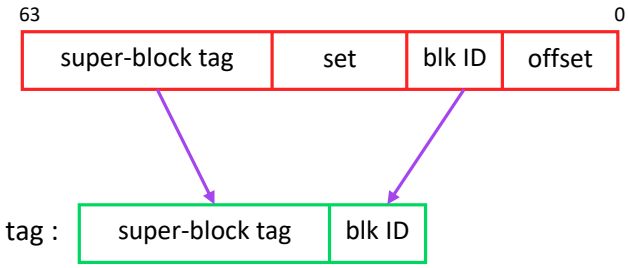


Figure 3.1 – Address mapping in baseline-modified configuration

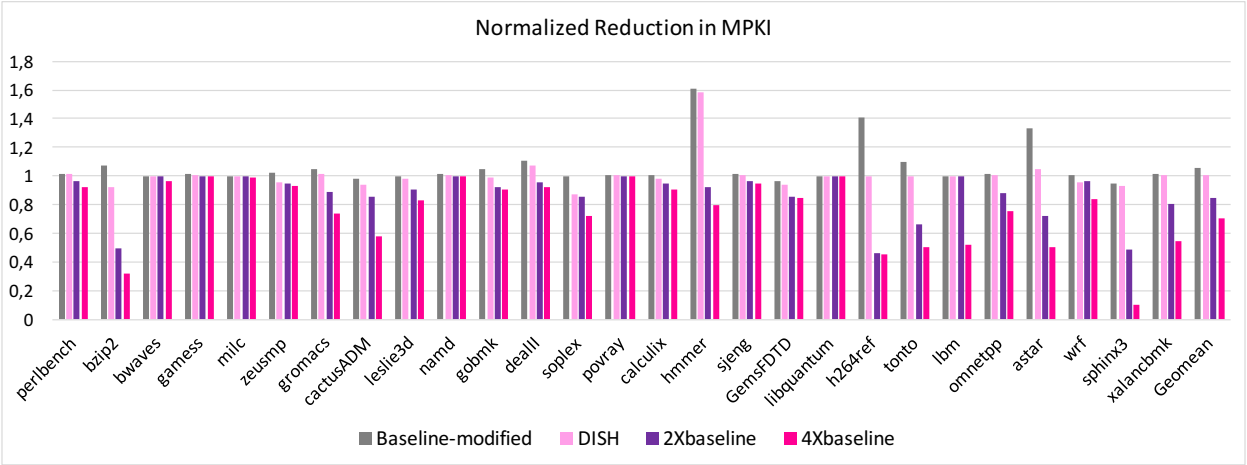


Figure 3.2 – Number of LLC misses normalized to an uncompressed cache, in terms of MPKI

uncompressed blocks compete for entries in the same set. As we will discuss in the next section, compressed caches can be subject to underutilization.

3.1.2 Gradual refill problem of DISH

Indeed, the potential capacity of the compressed cache may be underutilized due to two factors: non-compressibility or non-compactability of a block, lack of spatial locality.

— **Non-compactability of a block** may happen when a sub-block is compressible only by itself and it is not co-allocatable with others. In these cases, there is only one sub-block in the super-block; therefore, we are not taking advantage of compression.

— **Lack of spatial locality** could cause underutilization in compressed caches. If the neighboring blocks within a compressible super-block are not accessed, the extra space gained from compression remains unused. This is due to the fact that

whenever a sub-block is brought to the cache, the cache allocates a super-block for the requested sub-block. This super-block has one valid sub-block, the requested one, and three invalid sub-blocks. If the other sub-blocks are not accessed in the program before the current sub-block is evicted, this extra space will be unused.

Apart from the underutilization of compressed caches, bulk eviction and gradual refill, which is the focus of this work, is another problem in sector-based layouts. As replacement is performed at the granularity of super-blocks, all sub-blocks of the compressed super-block are evicted at once. Therefore, a single super-block eviction may generate four subsequent misses to refill the whole super-block.

3.1.3 Opportunities for compressed cache layout aware prefetching

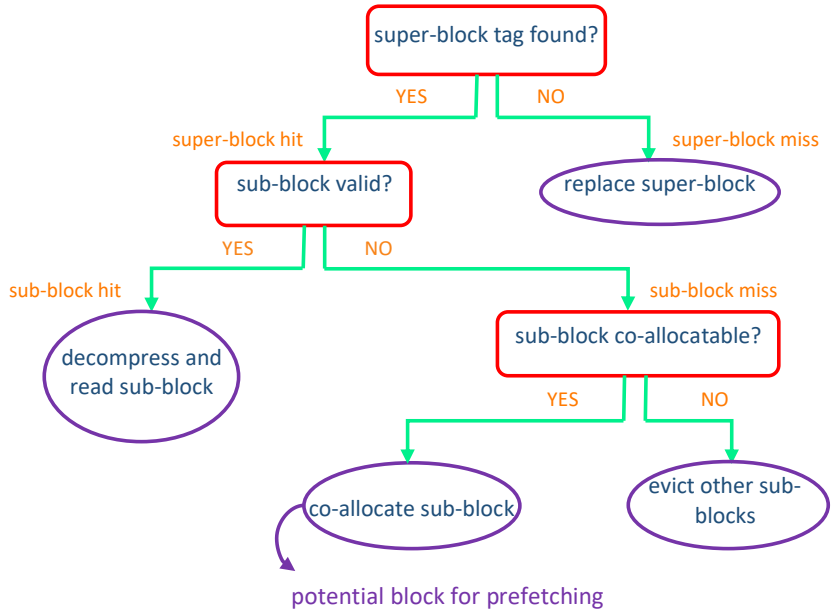


Figure 3.3 – Cache operations on read access

As shown in Figure 3.2, even by compressing the LLC, we get a modest improvement in terms of MPKI. In this section we investigate how many of the misses could be avoided by systematically prefetching the blocks that are co-allocatable in a super-block.

Figure 3.3 shows the decision diagram of compressed cache operations on a read access.

The cache checks all super-blocks of the set for a tag match: If no match is found, the access is a miss and the cache needs to replace the whole super-block; otherwise, it is a super-block hit. In this case, DISH checks the valid bit of the corresponding sub-block, or the BID field for uncompressed super-blocks, for each matching entry. Depending on sub-block validity, a sub-block miss or a sub-block hit could happen. In this work we focus on the co-allocation of a sub-block on a super-block hit and sub-block miss.

In other words, if there is an access to a sub-block (e.g., B2 in Figure 2.11b) that is co-allocatable in the super-block, but not yet present in the cache, we could have prefetched it without any cache pollution. Thus, we could avoid this sub-block miss in the cache. In Figure 2.11b B2 is co-allocatable in the super-block if it can be compressed with the same dictionary elements as the super-block or it can add new entries to the empty slots of the dictionary. We will show that cache miss on a co-allocatable sub-block occurs frequently in cache accesses and prefetching potential blocks is advantageous on all accesses.

In order to evaluate the potential for prefetching, we count the prefetching opportunities. As it is shown in Figure 3.3, the misses that occur on a super-block hit and sub-block miss are potential candidates for prefetching. If such block is co-allocatable with the ones that are already present in the super-block, it could have been prefetched at no expense of capacity and the miss could have been avoided. Hence, we do not need to wait for miss accesses to each sub-block to refill the compressed super-block. This does not evict any block from the cache.

Using gem5, we measured the fraction of misses that could be avoided by taking advantage of prefetching. Figure 3.4 shows the percentage of LLC misses that occur on sub-blocks that are co-allocatable within an existing super-block. We could reduce the number of misses by up to 20%, by prefetching compressed blocks before they are requested by the processor, without evicting any data from the cache.

This suggests that there is a potential for a synergistic interaction of prefetching and cache compression in processor architecture. Thus, by prefetching compressible blocks to L3, we seek the reduction in number of LLC misses.

3.2 Compression predictor

As it was mentioned, there is potential to create a synergy between hardware prefetching and compressed cache by prefetching co-allocatable sub-blocks. Therefore, we need to predict which blocks are compressible or co-allocatable in advance, before accessing their

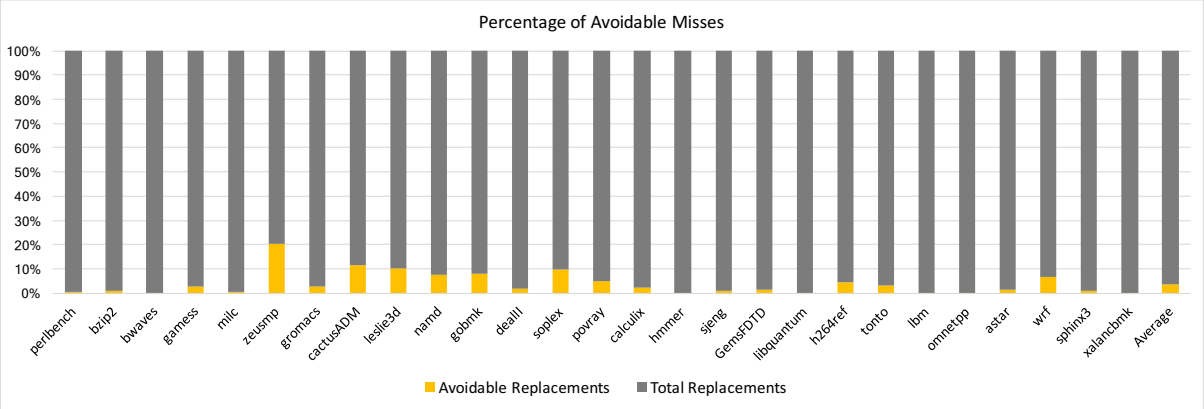


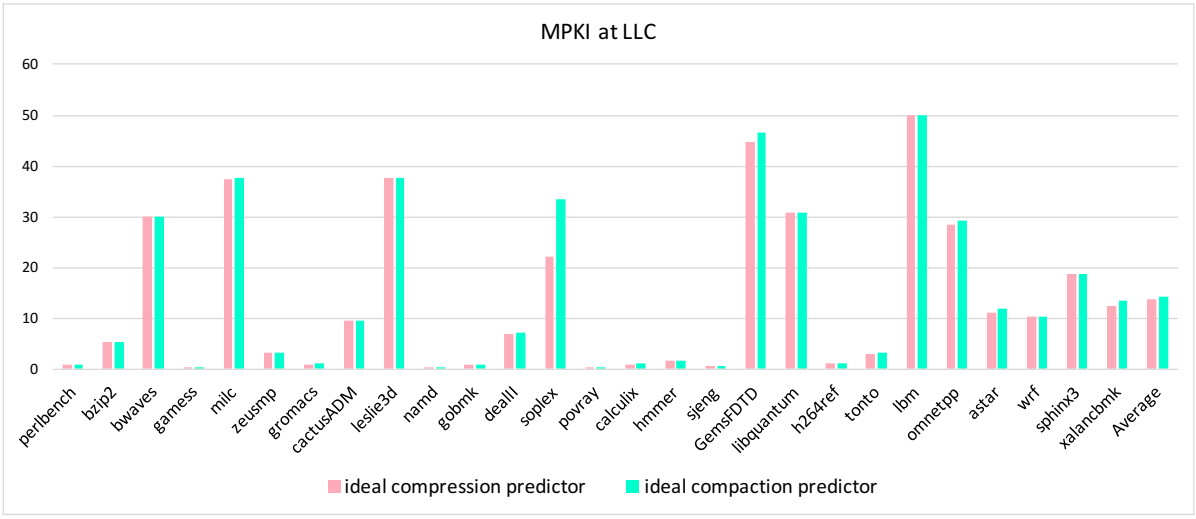
Figure 3.4 – Fraction of avoidable misses in a compressed LLC

data. In order to estimate these blocks as candidates for prefetching before their content is available, we need a predictor. We implemented an oracle compression predictor and an oracle compaction predictor to decide about the prediction metric (compressibility or compactability).

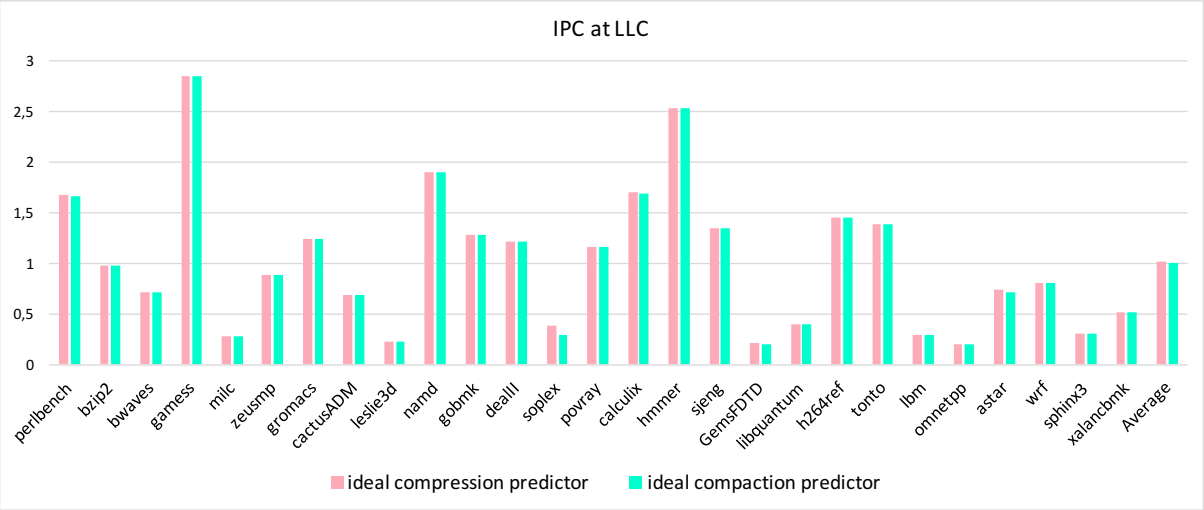
3.2.1 Prediction criterion

The oracle compaction predictor estimates compactability based on the content of individual blocks, and it only considers super-blocks that have 4 valid sub-blocks. The oracle compression predictor considers all compressible blocks regardless of their ability to co-allocate in a super-block. In this section we are evaluating them to figure out which metric (compressibility or compactability) is a better indicator.

Figure 3.5 shows the number of LLC misses and IPC in a compressed cache, with a Nextline prefetcher in LLC, using the oracle compression predictor and the oracle compaction predictor to trigger the prefetch. The results show that an oracle compression predictor is accurate enough to be employed to a compressed cache. Our simulations indicate that prefetching the blocks that are not co-allocatable (but only compressible) does not increase the number of misses, and these blocks are even likely to be useful in the program. Thus, the oracle compression predictor tends to prefetch more blocks than the oracle compaction predictor, and may lead to fewer cache misses as an advantage. As it is shown in Figure 3.5a, benchmarks such as *soplex*, *omnetpp*, *astar* and *xalancbmk* have lower MPKI using the oracle compression predictor rather than the oracle compaction predictor.



(a) Number of LLC misses, in terms of MPKI



(b) Performance, in terms of IPC

Figure 3.5 – Comparison between the ideal compression predictor and the ideal compaction predictor

3.2.2 Proposed compression predictor

The evaluations of prefetching using oracle compression and compaction predictors indicate that on our simulations, compression is at least as good as compaction, and it is often a better criterion. Therefore, in this section, we propose a stream-based compression predictor to predict whether a sub-block is compressible.

As the predictor needs to predict the compressibility of blocks that have not been accessed recently or were never accessed, a prediction based solely on the block address is not practical. Instead, we predict the compressibility of a *stream* of memory reads, where a stream corresponds to all accesses performed by a given load instruction in the program. Therefore, the prediction is done based on the PC of the request, not based on the address. Moreover, the prediction is carried out in LLC.

The key idea of stream-based prediction is that all memory locations accessed by a given load instruction are likely to share the same data-type and similar characteristics. Thus, whether prior blocks accessed by a load instruction were compressible is a good indicator of the compressibility of future blocks loaded from the same instruction.

We use a table indexed by a hash of the PC to store the compressibility likelihood of each load instruction and update it after every read miss. The PC table has 256 sets and 4 ways, using the LRU replacement policy. Each entry in this table has a PC tag, a 4-bit saturating counter and a validity bit. The saturating counter and compression bit get updated after every fill response from the memory. Whenever the block is compressible, the saturating counter gets incremented; otherwise, it gets decremented. On every access, the predictor decides whether a block is compressible, based on the counter. Accessing the PC table is based on each PC that is carried in the request.

We have implemented the proposed stream-based compression predictor and compared it to the oracle compression predictor to check the accuracy of the proposed predictor. The number of true positives, true negatives, false positives and false negatives are illustrated in Figure 3.6. Among all the prediction decisions, on average, there are 13% true positives, 76% true negatives, 4% false positives, and 7% false negatives.

Hardware overhead

The overhead of the proposed compression predictor is a table that is addressed by PC. Therefore, every request needs to carry a PC to use for indexing. The table has 256×4 entries and each entry has 45 bits (40-bit PC tag + 4-bit saturating counter +

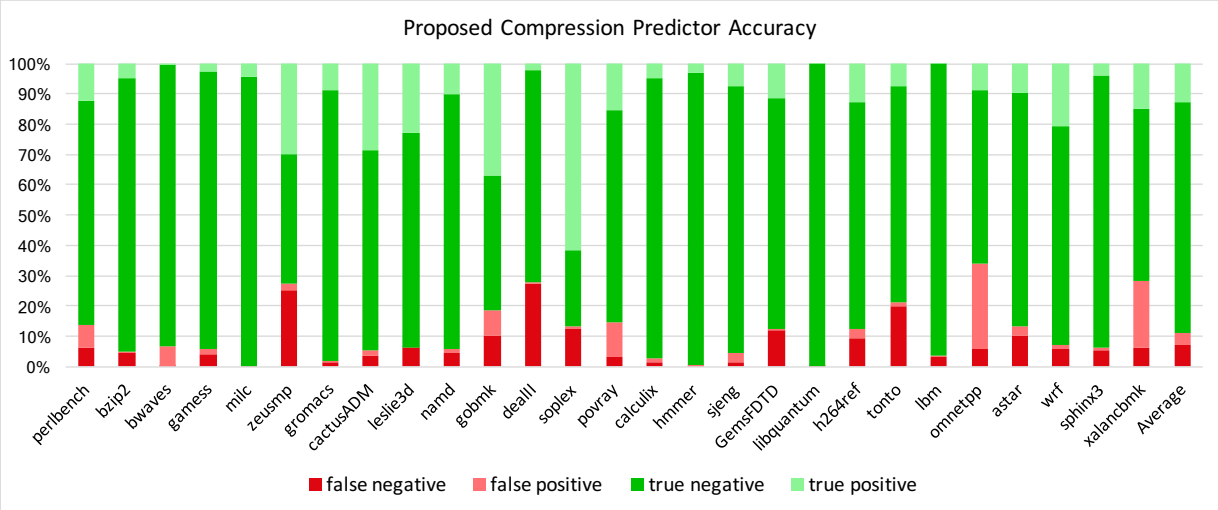


Figure 3.6 – Proposed compression predictor accuracy, comparing to oracle compression predictor

1-bit validity bit). Hence, the table requires less than 6kB of memory, which is 0.58% of the 1MB LLC slice we consider. Increasing the cache size by this amount is unlikely to affect performance.

The size of the PC table has been obtained using empirical evaluations. We have tried different size of tables, such as 128×4 and 1024×4, and we observed a negligible improvement in only two of the benchmarks, in terms of prediction accuracy. Moreover, we changed the LLC size to 2MB and the results show that our predictor is insensitive to the cache size. However, the proposed predictor could be optimized; it is possible to implement the table without PC tags using a direct mapped table indexed by hashed PC. By removing tag bits from the table, the predictor overhead could be 700B, that is, 88% smaller than the table that we have actually implemented.

The diagram of the PC-based table (prefetch table) and the prediction phase is shown in Figure 3.7. On a prediction, if the PC tag matches, the valid bit is set and the compression counter is high enough, the predictor decides that the block is compressible. We implemented the compression counter as a 4-bit saturating counter that is initialized to 0b1000. Besides, the counter's threshold is 0b1000 which is half of the counter value. Therefore, if the compression counter for a given PC is greater than 0b1000, it is predicted that the PC is accessing a compressible data; otherwise, the PC accesses an uncompressible block.

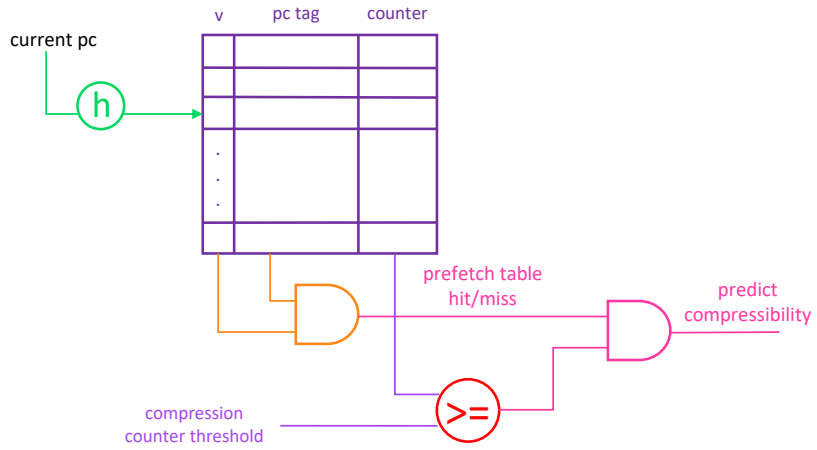


Figure 3.7 – PC table for proposed compression predictor

3.3 Neighbor-CLAP

In order to enhance cache compression and diminish the gradual refill problem of compressed caches, we propose neighbor-CLAP, a compressed cache layout aware prefetcher for the YACC layout. The goal is to avoid misses that happen on a co-allocatable block on a super-block hit, as it was shown in Figure 3.3. Therefore, we can leverage compressed cache layouts by means of the compression predictor and taking advantage of prefetching.

The three-level cache hierarchy integrated with the proposed neighbor-CLAP architecture is illustrated in Figure 3.8.

In this configuration, L1, L2 and main memory are uncompressed and LLC is a compressed cache. Therefore, there is a compressor for transferring data from memory to LLC. Furthermore, there is a decompression logic to decompress data in order to use data in the CPU and store them in L1 and L2.

When there is an access to the compressed LLC (CLLC), the cache looks for the address in the lookup table (Figure 3.9). In a compressed cache layout aware prefetcher, whenever a read request causes a super-block miss, the compression predictor is invoked. The proposed compression predictor predicts whether the block is compressible (Section 3.2.2). Based on the predictor's decision, the compression-aware prefetcher prefetches either three blocks or does not prefetch. If it predicts the block is compressible, the cache sends three requests to prefetch the three neighbor sub-blocks in a super-block. In case the predictor predicts the block is not compressible, the system does not prefetch anything.

Figure 3.9 shows the cache lookup diagram in a CLLC. For ease of understanding,

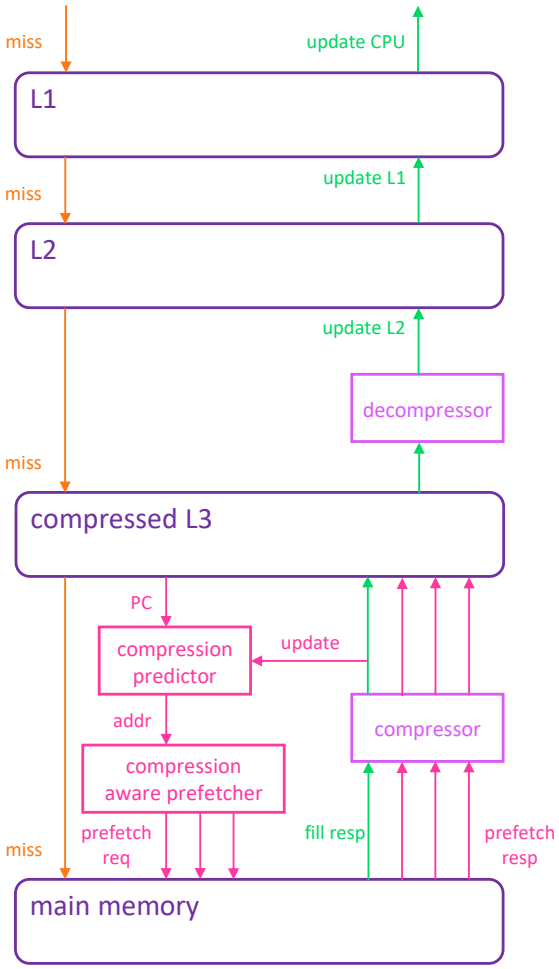


Figure 3.8 – Cache hierarchy diagram integrated with compressed cache layout aware prefetching

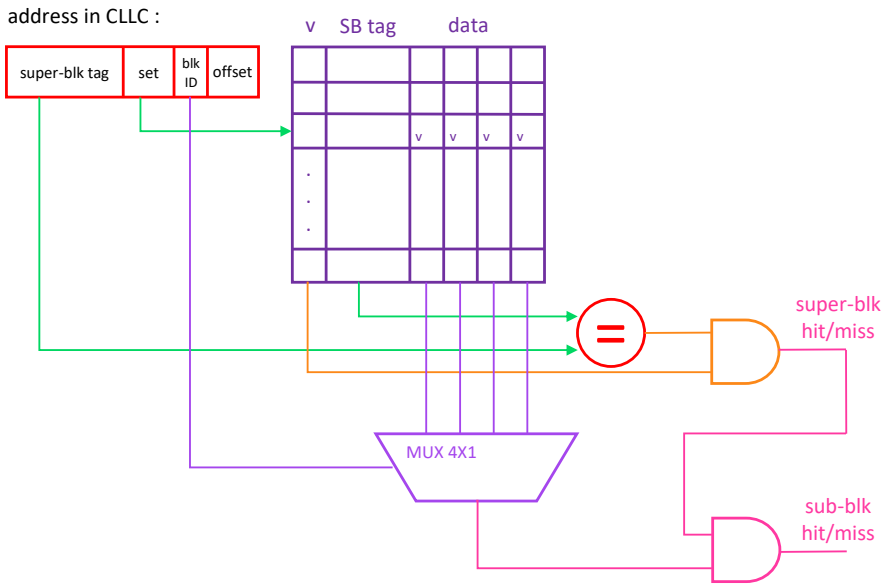


Figure 3.9 – Cache lookup diagram in CLLC (direct mapped cache)

only 1 way is shown in this figure (as in a direct mapped cache). On a cache access, the set bits of a 64-bit memory address index a super-block entry. Since this is a compressed cache, the data entry can map up to 4 sub-blocks. On a super-block hit, the cache checks the blkID part of the address with the blkID of the sub-block in the cache entry (using a multiplexer). On a sub-block hit in CLLC, the data is sent to the upper cache levels and the CPU; otherwise, on a miss, a request will be sent to the main memory to fetch the data.

Based on the predictor's decision, Figure 3.7, the prefetcher is invoked and it prefetches either three neighbor blocks or it does not prefetch any block. Figure 3.10 shows the diagram of a prefetcher after the predictor decides a block is compressible. In this architecture, the super-block tag, set and blkID are fed into an address generator that generates the addresses of neighbor blocks of the demand block. These addresses will be stored in a prefetch queue to be prefetched afterwards.

Prefetching three neighbor blocks in neighbor-CLAP architecture is done in burst mode, by sending three prefetch requests, in a row, to the memory. The advantage of sending a burst of four accesses (a demand request and three prefetch requests) is that we access the same DRAM page.

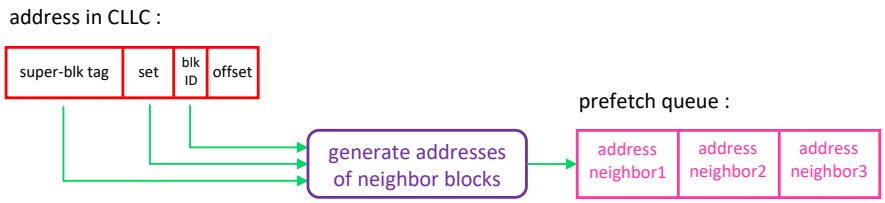


Figure 3.10 – Prefetcher diagram in neighbor-CLAP configuration

3.4 Stride-CLAP

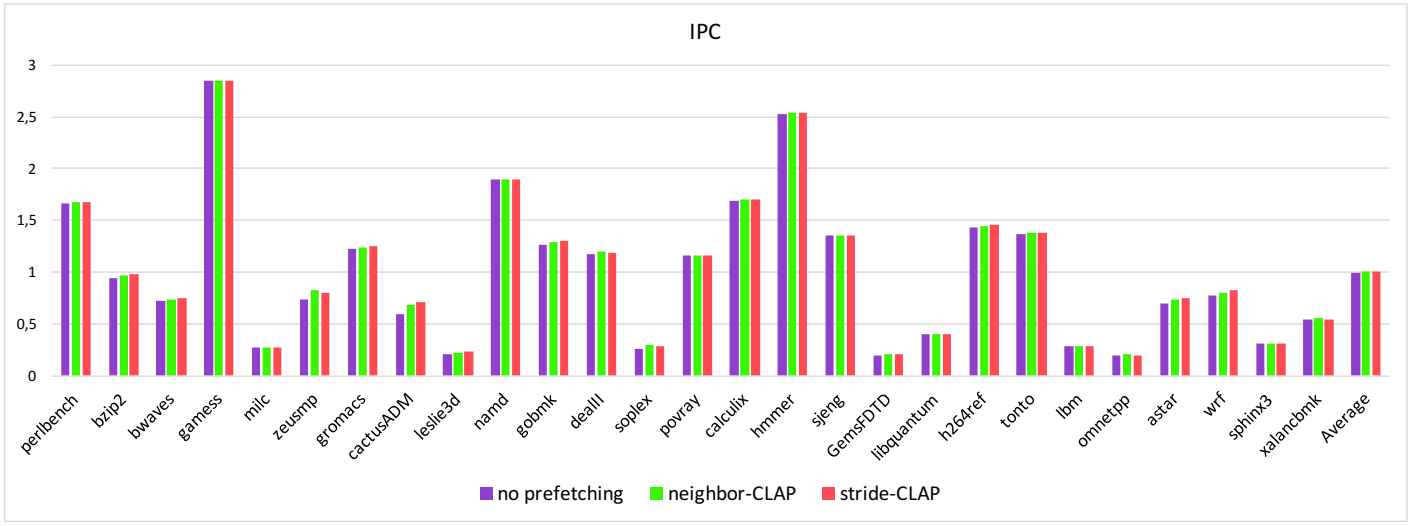
Neighbor-CLAP only prefetches the neighbors in the same super-block as the demand block; however, we can go further and do the prediction across the super-blocks. As previously mentioned in Section 3.2.2, the compression predictor does not rely on the address; it predicts whether a given load in a program (PC) is accessing a compressible block; therefore, we can apply our compression predictor on strided accesses. The stride prefetcher calculates the stride based on a stream of accesses to the same load. If a load is predicted to access a compressible block (using the compression predictor), the other accesses to the same load are likely to be compressible as well. The goal is to have a compression-aware prefetcher that activates the prefetcher if two criteria are met: a high confidence stride and a compressible stream of accesses; we call this configuration stride-CLAP. stride-CLAP carries out stride prefetching on every access if the predictor predicts a block is compressible; if not, it does not send the prefetch requests to the main memory.

Stride-CLAP has the same architecture as the neighbor-CLAP (Figure 3.8). Moreover, in stride-CLAP, PC tables in the compression predictor and the stride prefetcher can be unified in one table without any extra hardware overhead. The predictor's table and the stride prefetching table both are indexed by hash of the PC; consequently, they can share the same storage. The stride-CLAP makes the prefetch decision based on both stride confidence and compressiblity of a load access.

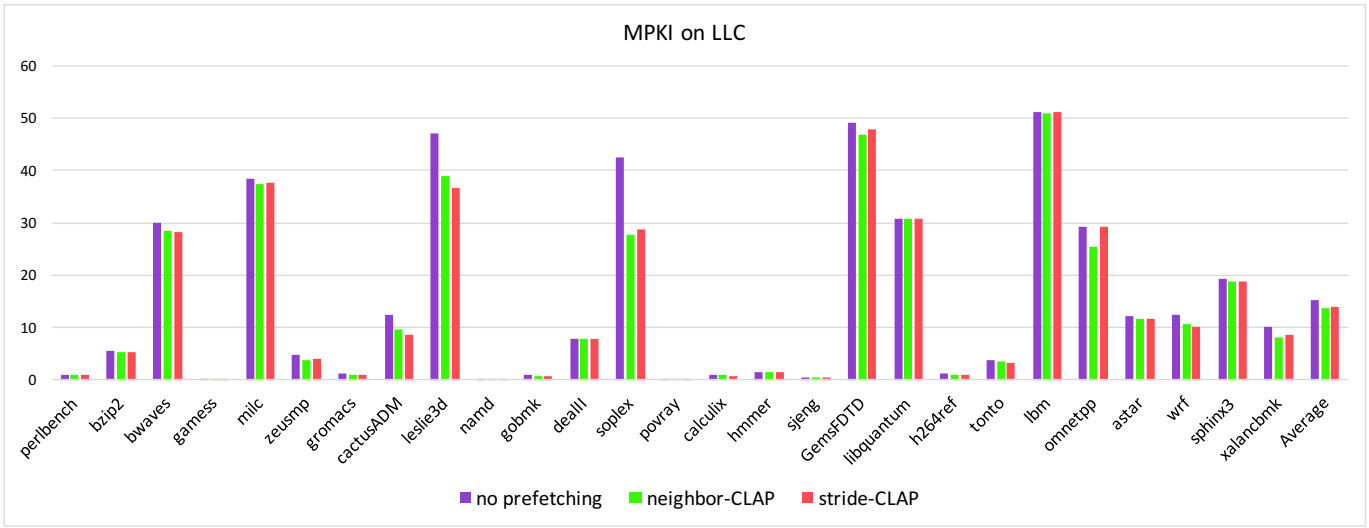
3.5 CLAP evaluation

We have implemented a compressed cache without prefetching, a compressed cache with neighbor-CLAP and a compressed cache with stride-CLAP, using DISH as the compression algorithm. As it is illustrated in Figure 3.11, neighbor-CLAP and stride-CLAP outperform compressed cache without prefetching in terms of IPC and MPKI. They im-

prove the performance by 3% comparing to compressed cache without prefetching, on average (up to 17% for neighbor-CLAP and up to 20% for stride-CLAP, in *cactusADM*). Furthermore, both neighbor-CLAP and stride-CLAP reduce LLC misses of the applications by 9% ,on average (up to 34.8% for neighbor-CLAP and up to 35.3% for stride-CLAP, in *soplex*).



(a) Instructions per cycle (IPC)



(b) LLC misses per 1000 instructions (MPKI)

Figure 3.11 – Comparison of neighbor-CLAP and stride-CLAP against compressed cache without prefetching

3.5.1 Bus utilization

Figure 3.12 shows memory traffic for different benchmarks using three configurations: a compressed cache without prefetching, with neighbor-CLAP and with stride-CLAP. For applications such as *bwaves*, *zeusmp*, *cactusADM*, *leslie3d* and *GemsFDTD*, both configurations of CLAP can reduce the number of LLC misses with just a small cost in increasing memory traffic.

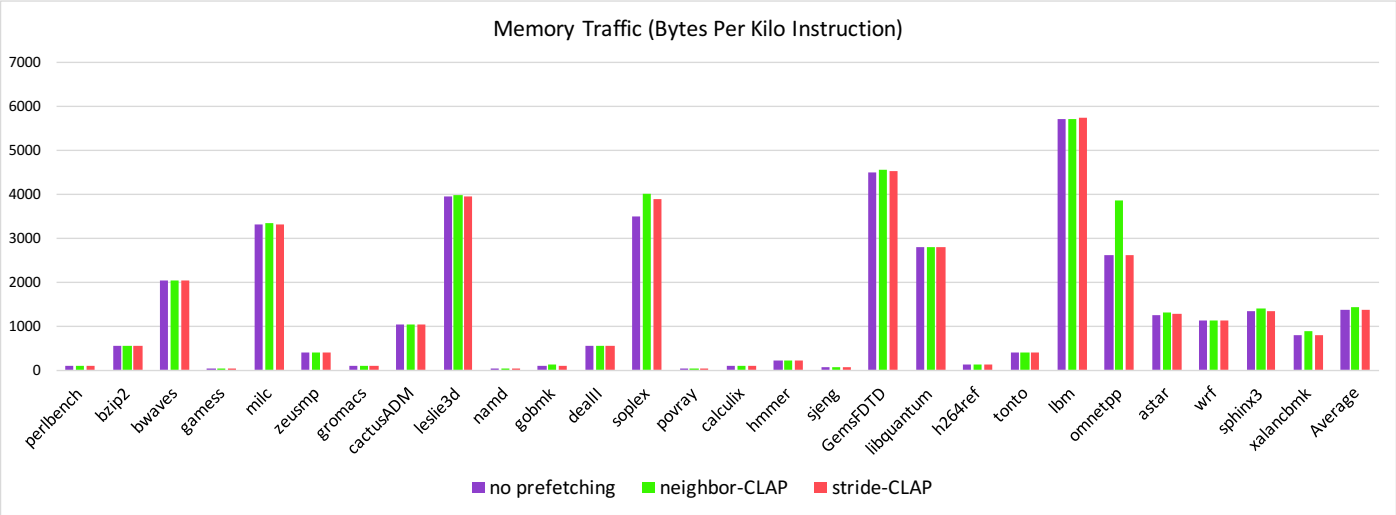


Figure 3.12 – Memory traffic in different configurations of compressed cache

As it is shown in this figure, on average, stride-CLAP has almost the same memory traffic as a compressed cache without prefetching. This signifies the prefetches performed by stride-CLAP are useful in the program and do not increase the bandwidth contention.

3.5.2 Effective cache utilization

In CLAP, our goal is not to increase the compression/compaction ratio, but to fit more valid sub-blocks in a super-block. To achieve this goal and find out how much more data we can fit in the cache, we define the effective cache utilization metric; which is calculated by the average number of valid sub-blocks per valid super-block.

Cache compression increases the effective cache utilization, which can also be increased by taking advantage of prefetching. Figure 3.13 shows the effective cache utilization for a compressed cache without prefetching comparing to neighbor-CLAP and stride-CLAP. On average, a compressed cache without prefetching improves utilization of the cache by 19%; while, neighbor-CLAP increases the effective cache utilization by 23%, comparing

to an uncompressed cache. Thus, we can increase the effective cache utilization by 4% by employing neighbor-CLAP to a compressed cache.

However, stride-CLAP does not increase the utilization of a compressed cache without prefetching. In stride-CLAP, sometimes the stride is larger than the super-block size; therefore, it may prefetch blocks that are far from each other and do not share the same super-block. For each prefetched block, a super-block is allocated in the cache and it only contains one valid block. Thus, in stride-CLAP we may end up having multiple super-blocks with only one valid sub-block, which does not enhance the cache utilization.

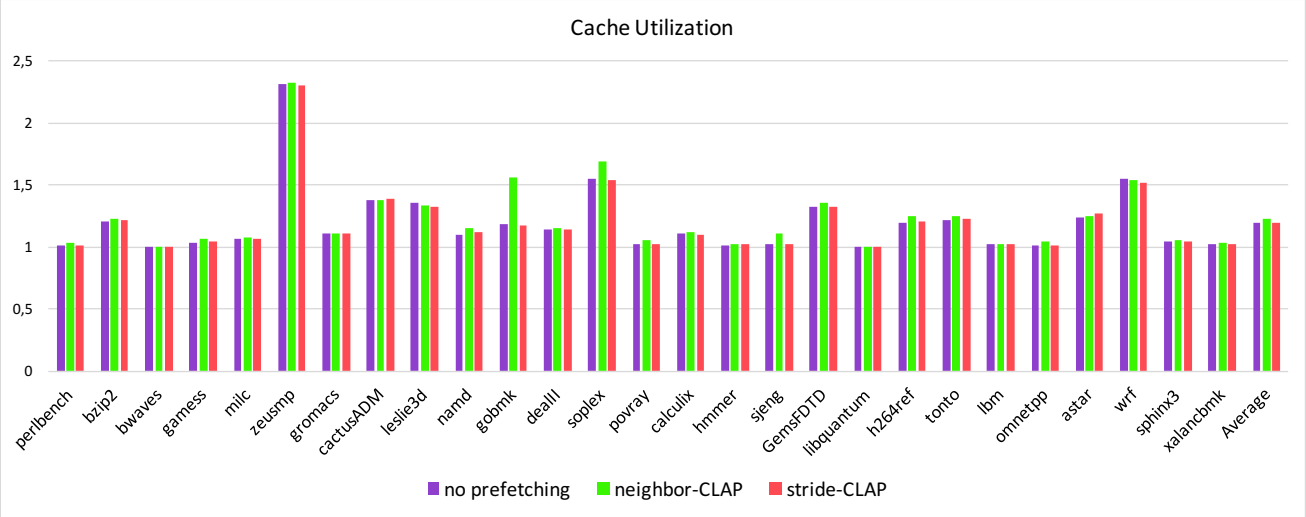


Figure 3.13 – Effective cache utilization relative to an uncompressed cache

3.6 CLAP combination with other prefetchers

CLAP is independent of the baseline prefetcher and it can be added to any existing prefetcher such as Nextline and stride. Therefore, the proposed approach of this study is orthogonal to other prefetching methods. In this section, we combine the two configurations of CLAP with Nextline and stride prefetchers and propose three multi-prefetchers: neighbor-CLAP with Nextline prefetcher, neighbor-CLAP with stride prefetcher, and stride-CLAP with Nextline prefetcher.

3.6.1 Nextline-based prefetcher with neighbor-CLAP

We have employed the neighbor-CLAP strategy to Nextline prefetcher as the baseline. We can take benefit of neighbor-CLAP in a system with Nextline prefetching as baseline and prefetching three neighbor sub-blocks in case the compression predictor decides a block is compressible. As it is shown in Figure 3.14, we can reduce LLC misses by 6%, on average, and improve the performance by a maximum of 6% in *soplex*.

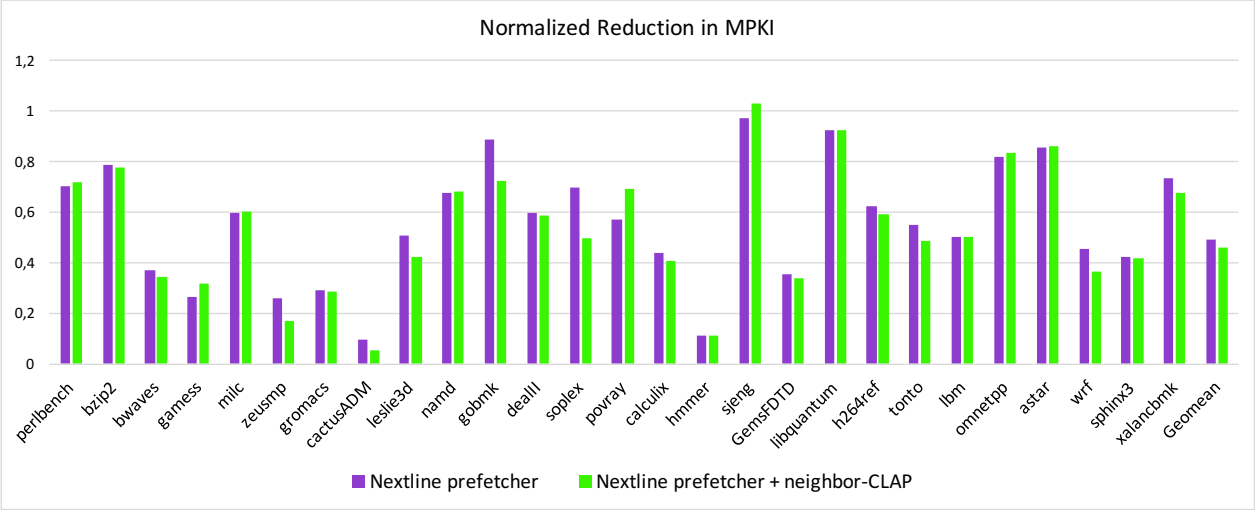
3.6.2 Stride-based prefetcher with neighbor-CLAP

Figure 3.15 shows the comparison between a compressed cache with stride prefetcher and a compressed cache with both stride prefetcher and neighbor-CLAP. The stride prefetcher (Section 2.2.3) in both configurations is implemented in CLLC and the sequence of addresses is generated on the same cache level. In our study, the stride prefetcher has a prefetch degree of 16, which means 16 prefetch requests are sent to the memory, regardless of the block's compressibility, in addition to the demand request. Nevertheless, if the calculated prefetch address is in another page, the prefetching is abandoned for this access. Moreover, as it was explained in Section 2.2.3, the distance between the prefetches is calculated by the stride of the prefetch in CLLC, that is a multiple of block size. In the stride prefetcher, in CLLC, there is a table indexed by PC that keeps track of all memory accesses to that specific PC and the last stride.

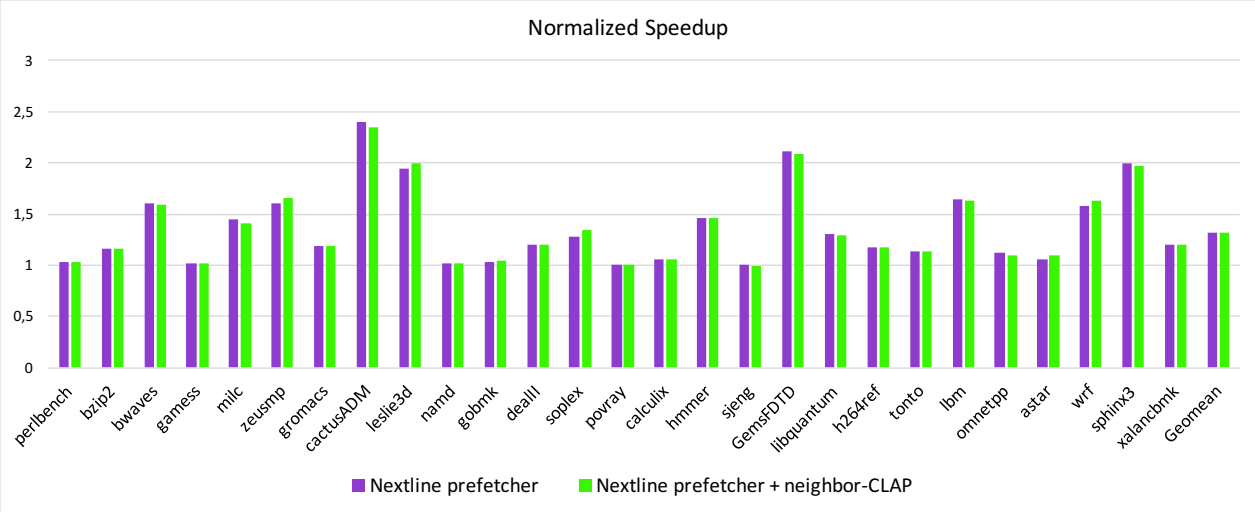
In this configuration, we employed stride and neighbor-CLAP as two independent prefetchers. The stride prefetching is performed on every cache access and it sends 16 prefetch requests to the prefetch queue in case it does not cross the page boundaries. Plus, when the compression predictor predicts a block is compressible, we prefetch three neighbor sub-blocks in the same super-block on top of stride prefetching. Ultimately, for a given access, there can be up to 19 prefetch requests in the prefetch queue. Therefore, using this configuration, the number of LLC misses are reduced by 8%, on average, and the performance is improved by up to 13% in *soplex*.

3.6.3 Nextline-based prefetcher with stride-CLAP

Taking into consideration the fact that stride-based prefetcher with neighbor-CLAP performs stride prefetches with a degree of 16 (performing 16 prefetch requests to the memory in a burst) on every access, this prefetcher can cause pollution in the cache.



(a) Number of LLC misses normalized to compressed cache without prefetching

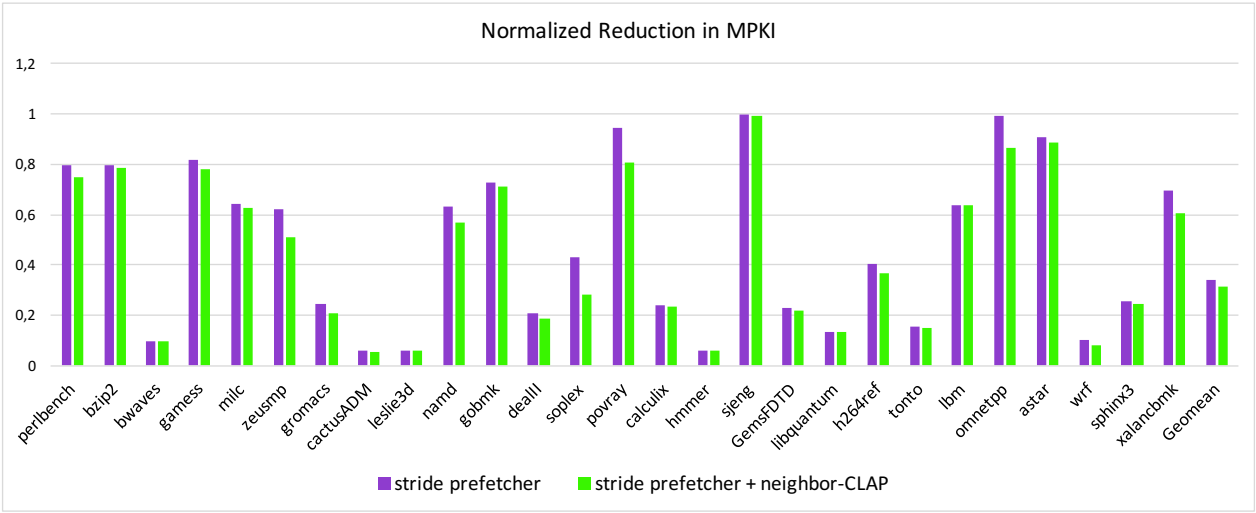


(b) IPC normalized to compressed cache without prefetching

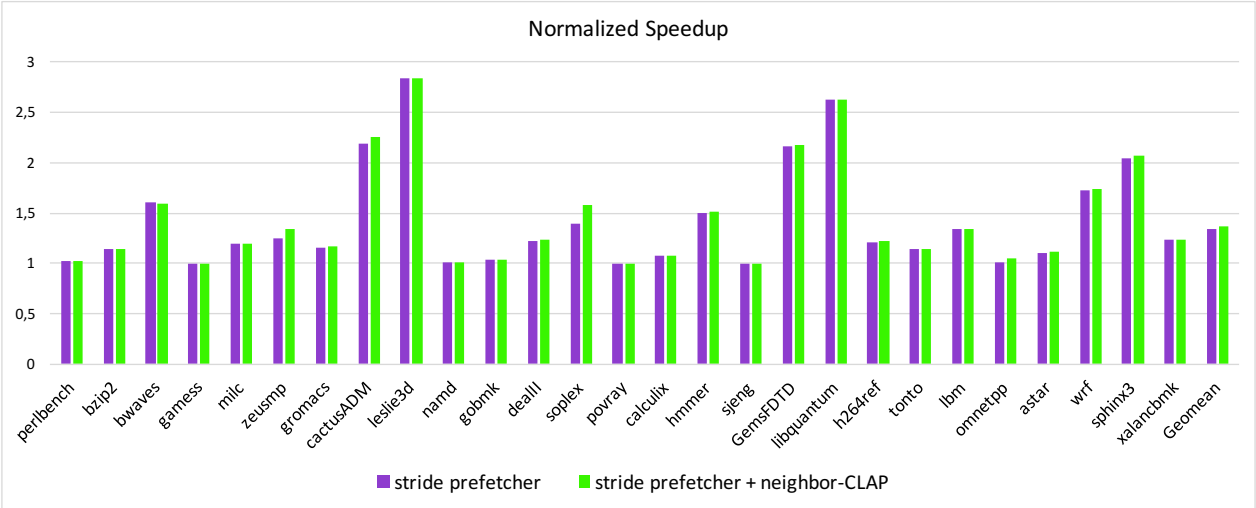
Figure 3.14 – Comparison of compressed cache with only Nextline prefetcher vs. Nextline and neighbor-CLAP, as numbers are normalized to compressed cache without prefetching

therefore, it is worthwhile to activate stride prefetcher only on compressible blocks and disable it for uncompressible blocks; thus, employing stride-CLAP is advantageous.

However, prefetching only on compressible data does not give the best performance in the cache. There are some benchmarks that have less compressible data. Thus, in order to gain the benefit from prefetching, a Nextline prefetcher can be applied to LLC to prefetch the next block regardless of compressibility of it.



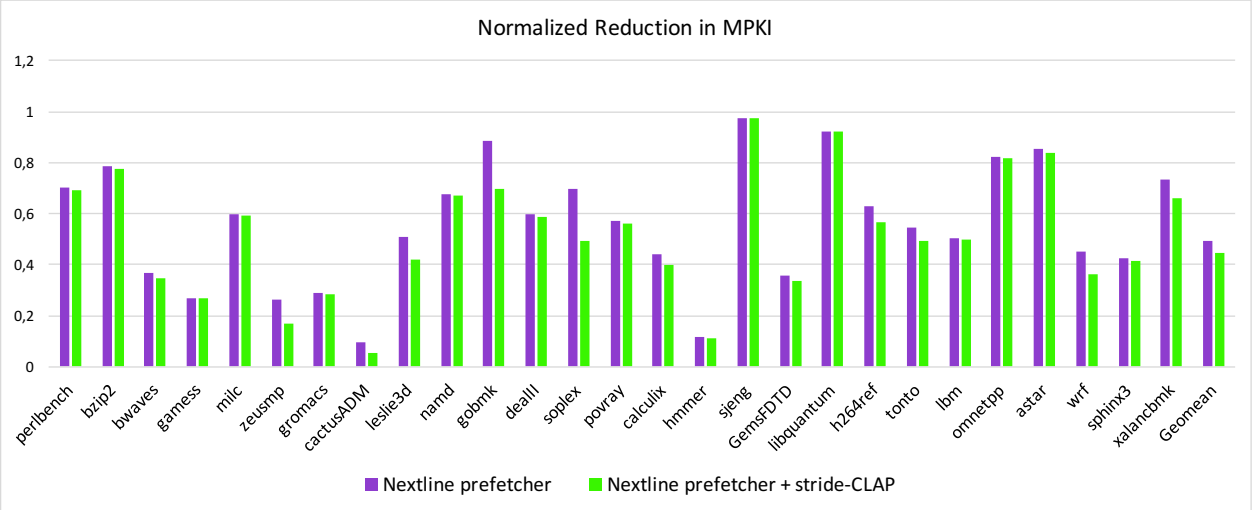
(a) Number of LLC misses normalized to compressed cache without prefetching



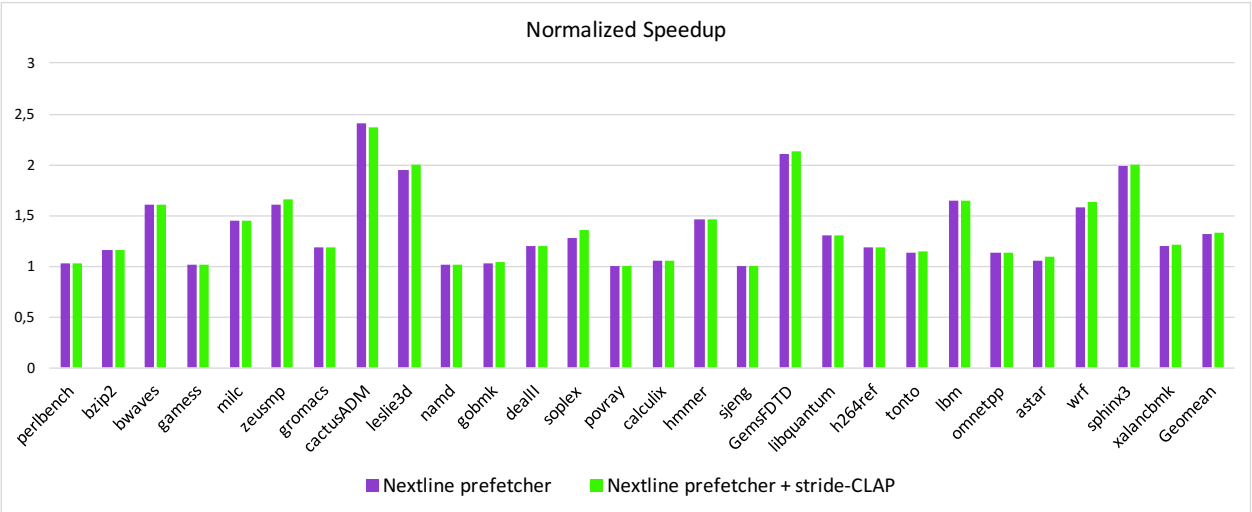
(b) IPC normalized to compressed cache without prefetching

Figure 3.15 – Comparison of compressed cache with only stride prefetcher vs. stride and neighbor-CLAP, as numbers are normalized to compressed cache without prefetching

Figure 3.16 shows the comparison of a compressed cache with only Nextline prefetcher vs. Nextline and stride-CLAP, as numbers are normalized to compressed cache without prefetching. By adding stride-CLAP on top of Nextline prefetching, the number of LLC misses is reduced by 9%, on overage, and the performance is enhanced by up to 6% in *soplex*.



(a) Number of LLC misses normalized to compressed cache without prefetching



(b) IPC normalized to compressed cache without prefetching

Figure 3.16 – Comparison of compressed cache with only Nextline prefetcher vs. Nextline and stride-CLAP, as numbers are normalized to compressed cache without prefetching

3.7 Summary

This chapter evaluates the potential of prefetching neighbors in a compressed cache and shows the importance of having a hardware prefetcher that is adapted to compression (using DISH compression algorithm in the LLC). The first part of this chapter has shown that there is potential for a synergistic interaction of prefetching and cache compression

in processor architecture.

Next, we have proposed a predictor to figure out which blocks are potential candidates for prefetching. Using this compression predictor, we can avoid pollution in the cache because we prefetch the blocks that can be allocatable in the same super-block as the requested block. We have shown that compression is a good indicator for prefetching and our proposed predictor is accurate enough to be employed to the LLC. The proposed predictor has a PC-based prediction and a negligible hardware overhead.

Finally, we apply our compression predictor to CLLC to decide which blocks should be prefetched and we propose a compressed cache layout aware prefetching (CLAP) mechanism. The experiments show that CLAP improves the performance of a compressed cache and also it reduces the number of misses in CLLC. Furthermore, we show that CLAP is not dependent to the prefetcher's baseline and it can be added to any existing prefetcher. In other words, we can have CLAP side by side to other prefetchers such as Nextline and stride.

In the next chapter, we will integrate CLAP and other prefetchers and propose two adaptive CLAPs. An adaptive CLAP chooses the best prefetcher configuration, either globally or locally, depending on the application.

Chapter 4

Adaptive CLAP

The main issue with prefetching is that there is no single prefetcher that works for all applications. For instance, taking into account the combined prefetchers that we proposed in Section 3.6, we find two prefetchers have better performance than others: stride-based prefetcher with neighbor-CLAP (stride+neighbor-CLAP), described in Section 3.6.2, and Nextline-based prefetcher with stride-CLAP (NL+stride-CLAP), introduced in Section 3.6.3). These prefetchers are made from basic CLAP elements proposed in Chapter 3; neighbor-CLAP and stride-CLAP. However, when comparing stride+neighbor-CLAP and NL+stride-CLAP with each other, there is no clear winner.

Figure 4.1 shows the results of MPKI and IPC in these two prefetchers; it indicates that most of the benchmarks have better performance using a stride prefetcher as the baseline and neighbor prefetcher on compressible blocks (stride+neighbor-CLAP) and the number of LLC misses is reduced by 10%, on average, in this prefetcher. But still there are some benchmarks, such as *milc*, *zeusmp*, *lbm* and *omnetpp* in which the NL+stride-CLAP configuration outperforms stride+neighbor-CLAP.

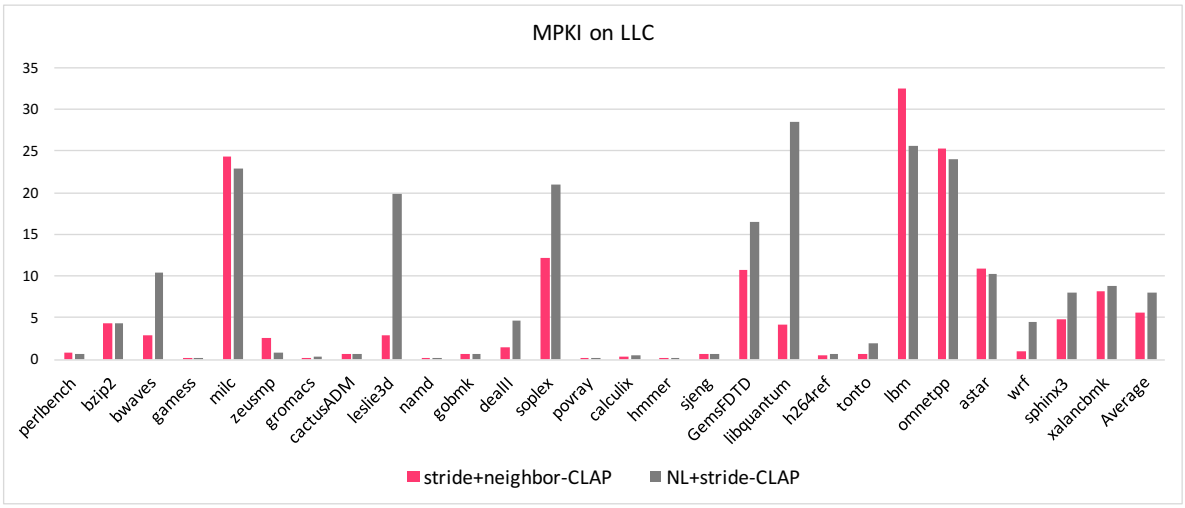
The results in Figure 4.1 show that we need a prefetcher selection mechanism to find the best prefetcher dynamically based on the running application. In this chapter, we propose two adaptive compressed cache layout aware prefetchers (adaptive CLAPs): global adaptive CLAP and local adaptive CLAP. The global adaptive CLAP selects the best prefetcher in each time interval for the whole cache. It estimates which of the prefetchers would have been the best in the previous intervals and employs that prefetcher for the next interval. The local adaptive CLAP chooses a prefetcher for each load access individually. By exploiting local adaptive CLAP, we find the best prefetching technique for each particular load.

Chapter 4 investigates the following questions:

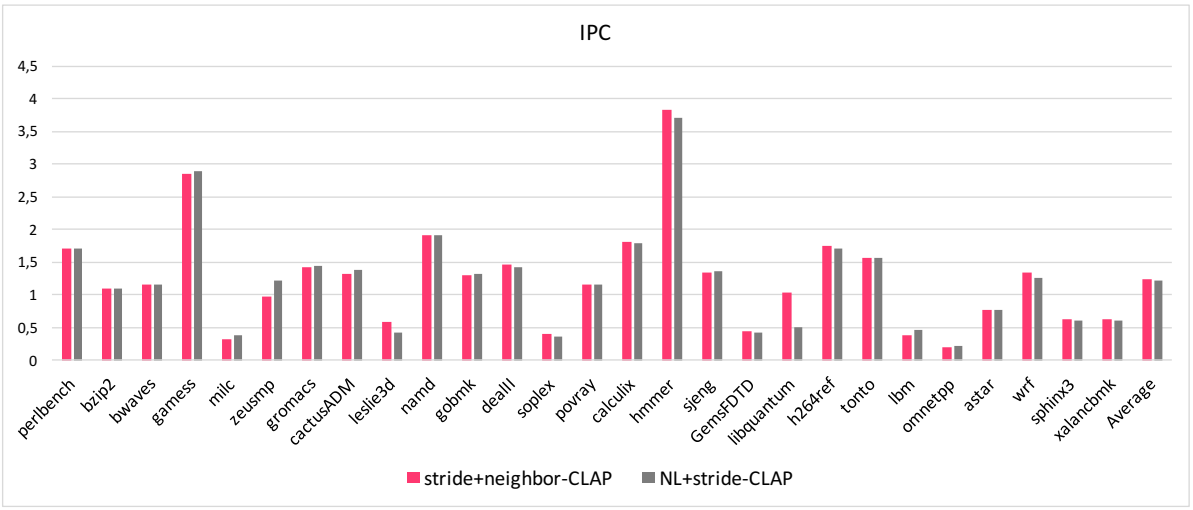
— How different prefetchers affect the cache performance?

— Which prefetcher is better to utilize in each situation?

— What parameters to measure for selecting the best prefetcher?



(a) Number of LLC misses in terms of MPKI



(b) LLC performance in terms of IPC

Figure 4.1 – Comparison of stride+neighbor-CLAP and NL+stride-CLAP

4.1 Global adaptive CLAP

As it was shown in Figure 4.1, sometimes stride+neighbor-CLAP is performing better and sometimes NL+stride-CLAP has better performance, depending on the workload; hence, we need a prefetcher selection mechanism to dynamically choose the best prefetcher. In this section we introduce a global adaptive CLAP which dynamically selects the best prefetcher among two compression-aware prefetchers for each time interval. We propose a generic selection mechanism that can be applied to any arbitrary prefetcher; not only CLAP.

In our study, the global adaptive CLAP architecture selects between two given prefetchers: stride+neighbor-CLAP and NL+stride-CLAP. The global adaptive CLAP enables only one of the two prefetchers for the whole cache for an interval and periodically switches between the two prefetchers. We choose to steer the prefetcher per interval in order to avoid switching between the prefetchers too frequently. We estimate the usefulness history of both prefetchers. At the beginning of each interval, the prefetcher that has a higher estimated usefulness is selected to perform the prefetching in the next interval. As we will explain in Section 4.1.1, in order to estimate the prefetchers usefulness, we propose a usefulness predictor.

In order to improve global adaptive CLAP, we tried steering the prefetcher selector based on other metrics such as cache pollution and prefetch lateness, rather than only usefulness. Nevertheless, we observed that in our study, prefetcher usefulness is the most relevant criterion regarding performance evaluation. Our approach is consistent with the results of an adaptive prefetcher introduced by Srinath [Sri+07]. As it was discussed in [Sri+07], steering the prefetcher according to the prefetcher usefulness metric only brings most of the performance benefit of a more complex prefetcher selector that also considers cache pollution and prefetch lateness. Besides, from hardware overhead perspective, for estimating late prefetches, an ordered set is required; while, implementing an ordered set, such as a queue, is difficult in hardware due to expensive lookups. Furthermore, estimating pollution needs simulating the whole cache that is costly in hardware. Therefore, it is difficult to implement hardware estimators for lateness and cache pollution.

In our mechanism, although there is only one actual prefetcher running at a time, we are able to estimate the usefulness of both prefetchers on every access using usefulness predictors. An alternative approach that we did not retain consists of feedback-directed methods, such as interval-based sampling employed in the adaptive prefetching technique proposed by Srinath [Sri+07]. However, this feedback-directed approach only takes into account the actual prefetcher that is used currently in the program. It takes the information (metrics explained in Section 2.2.5) of the running prefetcher and uses it as a feedback to adjust the aggressiveness of the prefetcher for the next interval. However, this approach does not help for selecting among multiple prefetchers. If we try to steer a prefetcher selector to choose between two prefetchers, using sampling as introduced in [Sri+07], we only have the information about one prefetcher at the time of sampling. Therefore, we do not have any information about the other prefetcher that is switched off in the current interval, and we cannot tell whether that other prefetcher would fare

better or worse than the current prefetcher. In our approach, with the help of usefulness predictor, we are able to keep track of prefetches that could be done by both prefetch mechanisms and decide based on the output of these two predictors.

4.1.1 Usefulness predictor

As we discussed previously, we tried different metrics for steering the prefetcher; and eventually, we decide to choose the best prefetcher based on the prefetcher usefulness (U(pref)) that is defined as Equation 4.1 [Sri+07].

$$\mathrm{U(pref)} = \frac{\text{number of useful prefetches}}{\text{number of total prefetches}} \tag{4.1}$$

A useful prefetch happens when a prefetched data block is available in the cache and there is a request on that data before it gets evicted from the cache. If the prefetched data block gets evicted before it is accessed by a request, the prefetch is unused. In our experiments, we tried steering the prefetcher based on only useful prefetches, and not the usefulness (Equation 4.1); however, the results were worse. Selecting the prefetcher only based on useful prefetches increases the number of LLC misses by 3.7%, comparing to steering the prefetcher based on prefetch usefulness. In this case, the prefetcher selector adjusts the prefetcher regardless of number of total prefetches. To obtain better results, we need to normalize the number of useful prefetches to total prefetches that a prefetcher carries out.

A complete accurate estimation of the two prefetcher behaviours in the previous intervals would necessitate simulating these behaviours. To approach these estimations, we implement a usefulness predictor for each prefetcher.

We need to estimate the usefulness of a prefetcher that is not currently in use; for instance, estimating usefulness of NL+stride-CLAP when stride+neighbor-CLAP is using in the cache. In order to do so, it is necessary to know the state of the cache if we would have benefited from prefetching. Therefore, we can have a cache content estimator for each prefetcher, which could be a FIFO queue, that keeps the prefetched addresses by the prefetchers. These blocks are not *actual prefetches* in the cache, they are just *simulated prefetches* stored in a queue to have a simulated part of the cache.

The cache content estimator knows which prefetched blocks are inserted in the cache, by the prefetcher, and which prefetched blocks are getting evicted from the cache. Moreover, each time that we have a read access, we need to do a lookup in the queue. A cache

content estimator containing prefetched addresses is shown in Figure 4.2. As an example, the queue in this figure shows the prefetched addresses carried out by stride+neighbor-CLAP. It contains three contiguous addresses A, A+1 and A+2 that are brought by the neighbor-CLAP; furthermore, it has addresses D, N and X that are prefetched by the stride prefetcher.

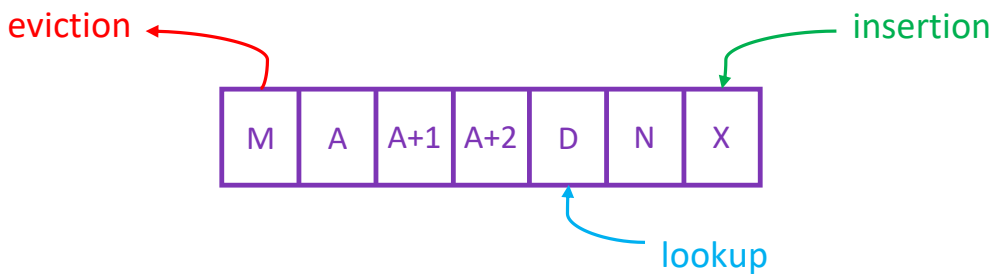


Figure 4.2 – Usefulness predictor layout

Performing lookups in queues are expensive to implement; thus, we need another mechanism to implement the usefulness predictor.

Usefulness predictor implementation

We take advantage of bloom filters [Blo70; Pei+02], a cost-effective mechanism, to be able to keep track of simulated prefetches. A bloom filter is a probabilistic data structure that is used to test membership in a set of elements using hash functions [Blo70]. Bloom filters can have false positives, that means they may report they contains an element that they do not. However; they can not give false negatives. Thus, it is guaranteed to report correctly if a bloom filter contains the requested element.

In this work, we dedicate a bloom filter to each prefetcher (stride+neighbor-CLAP and NL+stride-CLAP) as an approximation of the cache state; thus, they contain the set of blocks that would have been in the cache if we had performed the prefetches. Prefetch bloom filters store the potential addresses that can be prefetched by each prefetcher, which are simulated prefetches, not actual prefetches. The bloom filters in our study use H3 hash function [CW79] to set the bits in the bit vector at the index of the hashes to 1. The number of hashes for each address is four. Moreover, based on our experiments (Section 4.1.3), we determined the size of each bloom filter to be 16K which is 3.1% of the 1MB LLC considering the four bloom filters.

Based on the formulas introduced in [Blo70], the probability of a bit being 0 in a bloom filter (P(0)) is given in Equation 4.2; where d is the number of hashes for each address, N is the size of the bloom filter and n is the number of prefetch addresses, that are stored in the bloom filter, in each time interval. As we will discuss in Section 4.1.3, in our experiments, d is 4, N is 16K; however, the number of prefetches (n) in each time interval may vary.

$$P(0) = \left(1 - \frac{d}{N}\right)^n \tag{4.2}$$

Given Equation 4.2, the probability of a bit being 1 in a bloom filter (P(1)) is as follows:

$$P(1) = 1 - P(0) \tag{4.3}$$

Considering Equation 4.3, the probability of an address, which is mapping to 4 bits in the bloom filter, being a false positive is defined in Equation 4.4.

$$\phi = (1 - P(0))^d \tag{4.4}$$

Assuming d=4, N=16K and n=1K, which is an example of the number of prefetches, P(0)=0.77 and the probability of false positives occurring in the bloom filter is 0.23%. By increasing n o 2K, P(0) is 0.60 which results in $\phi = 0.023$. Therefore, considering our parameters, the probability of having false positives in this case is 2.3%. If the number of prefetches doubles (4K), P(0) is 0.36 that increases the probability of false positive to 15% which is a large number. Therefore, in order to keep the false positives tolerable for our configuration, we need to have less than 4K prefetches.

In the following, we describe the simulation of insertion, lookup and eviction in the cache content estimators that we implement using bloom filters.

Simulating insertion

The diagram of insertion in the cache content estimators is shown in Figure 4.3. Using cache content estimator, we can not simulate the whole cache; therefore, we only simulate the prefetches and insert them in the estimator. The goal is to make the usefulness estimation of the simulated prefetches (in the cache content estimator) independent of the actual prefetcher in the cache. In order to make the estimation independent, we pretend

the actual prefetcher in the cache does not exist and we consider the part of the cache that does not contain prefetched data, so we ignore blocks that are brought by the actual prefetcher to the cache.

In order to track prefetched blocks, we associate a prefetched flag to each block and set the flag when a prefetched block is inserted into the cache. The flag is set until we get a hit access to this block and the flag resets. If a block gets evicted with the prefetched flag set to 1, it means that the prefetched block was unused. If a block with prefetched flag set to 1 is accessed, this implies that the prefetched data block was useful.

For simulating the insertion in cache content estimators, we ignore prefetched blocks by checking the prefetched flag of each block. We perform simulated prefetches on all accesses except the hit accesses on the blocks that were not brought by the actual prefetcher (blocks with prefetched flag of 0). As previously mentioned, the prefetch addresses in the estimators are not actual prefetches which are going to be sent to the memory; they are stored for estimating useful prefetches.

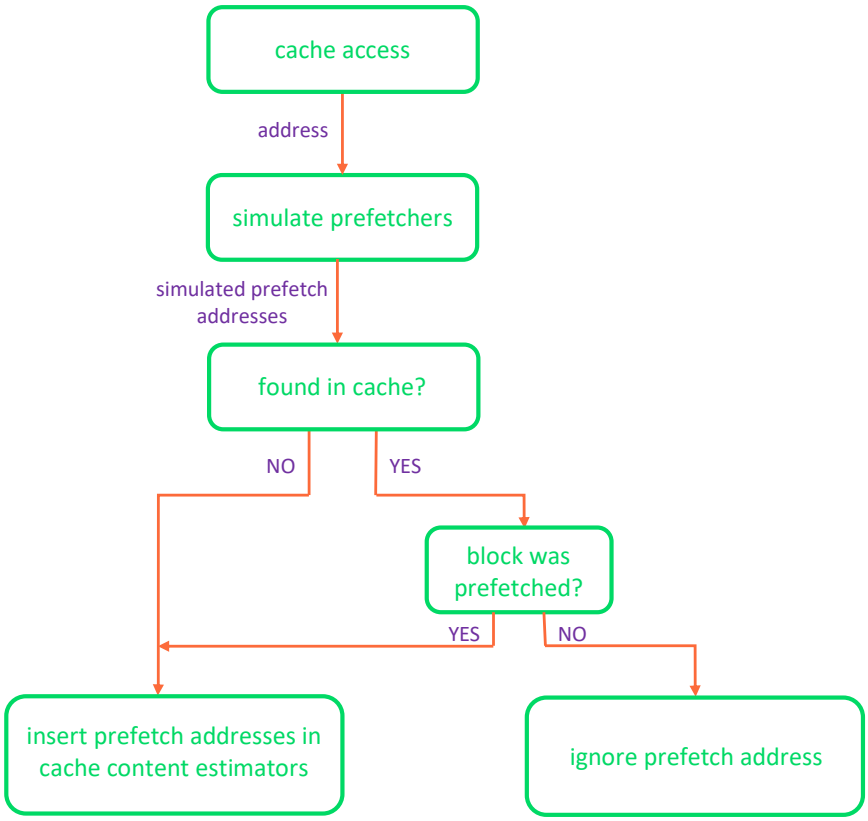


Figure 4.3 – Simulating insertion in cache content estimators

On every cache access, the cache content estimator of stride+neighbor-CLAP stores the prefetch addresses caused by this prefetcher, and the same happens to the cache content estimator of NL+stride-CLAP; it contains the address of the possible prefetches that could happen by activating this prefetcher.

Simulating lookup

As it is shown in Figure 4.4, on every hit on prefetched blocks or miss in the cache, cache content estimators do lookups to check whether the address is already stored there. It must be recalled that we also perform lookups on hits on prefetched blocks in order to ignore actual prefetches in the cache. The number of useful prefetches in the estimators get incremented whenever there is an actual miss in the cache and the address exists in cache content estimator. This means that if we had prefetched this block, using this specific prefetcher, we could have avoid the miss in the cache.

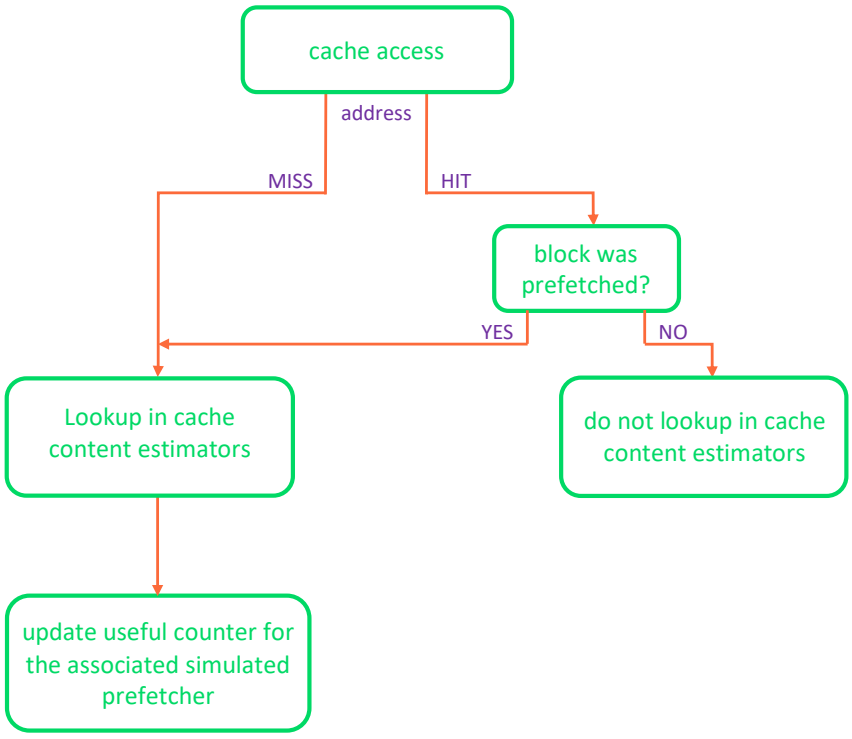


Figure 4.4 – Simulating lookup in cache content estimators

Simulating eviction

Removing an element from a bloom filter is not easy since false negatives can not happen in this kind of data structure. In a bloom filter, an element maps to multiple bits (4 bits in our experiments) and these bits can be shared among other elements. Thus, resetting any bits of an element may result in removing any other elements that share the same bit.

In our study, bloom filters are an approximation of the cache state and they need to be reset from time to time in order to approximate cache evictions. By resetting the whole bloom filter, we lose keeping track of the cache approximation for a specific time until the bloom filter gets filled. To solve this issue, we implemented the switchable bloom filter that has a pair of bloom filters which get erased alternatively. Therefore, when one of the bloom filters resets, the prefetcher selector can still decide based on the data presented in the other bloom filter. As it is illustrated in Figure 4.5, in a switchable bloom filter, composing of a pair of simple bloom filters, there is only one active bloom filter at a time. It is worth mentioning that both bloom filters keep storing the simulated prefetches even when they are deactivated. Therefore, from time to time when we clear the active bloom filters, we still have a history of simulated prefetches in the other bloom filter. Hence, we can switch to the other bloom filter and use the data stored there for steering the prefetcher.

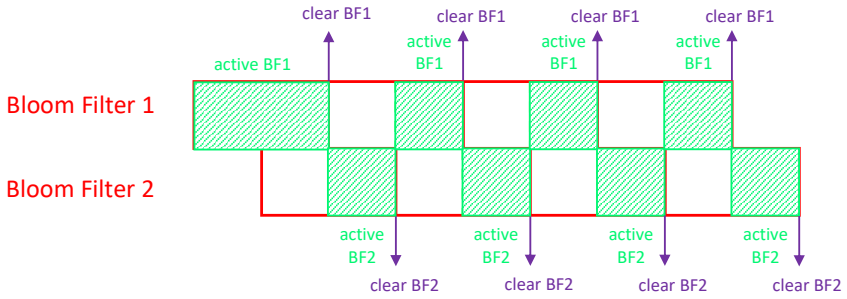


Figure 4.5 – Switchable bloom filters

In order to know when to reset and switch a bloom filter, we need to define an interval, based on lifetime of the blocks. The lifetime of blocks in the cache can vary. Hence, aiming to approximate this lifetime for set of blocks in bloom filters, we rely on the number of evictions and set the intervals based on the number of evicted blocks from the cache. At each time, the number of elements in active bloom filters is between n and $2n$; where

Table 4.1 – Global adaptive CLAP decision making table

	Compressible	Uncompressible
stride+neighbor-CLAP	Stride prefetching and neighbor-CLAP	Stride prefetching
NL+stride-CLAP	Nextline prefetching and stride-CLAP	Nextline prefetching

n is the window size of bloom filters. In the experiments, a specific number of evicted blocks from CLLC is defined as n for each interval. Taking into account that the CLLC size is 1MB and 16-way set associative, and also as we will explain in Section 4.1.3, using a limited number of simulations, we determined a number of 2048 evicted blocks as the lifetime of blocks in the bloom filters. At the end of each interval, the counter that keeps track of evicted blocks resets and a new interval begins.

4.1.2 Prefetcher selector

For implementing global adaptive CLAP, we use two multi-prefetchers discussed at the beginning of this chapter (Figure 4.1), and as indicated previously, it is advantageous to dynamically select between these two configurations. Table 4.1 shows the decision making of global adaptive CLAP using these two multi-prefetchers. In order to steer the prefetcher for an interval, global adaptive CLAP uses a two-level decision making procedure. In the first level, at the end of each interval, global adaptive CLAP selects between the two multi-prefetchers: stride+neighbor-CLAP and NL+stride-CLAP. Each prefetcher internally contains two simple prefetchers: stride+neighbor-CLAP has a stride prefetcher combined with a neighbor-CLAP; NL+stride-CLAP consists of a Nextline prefetcher combined with a stride-CLAP. The second level of the decision making happens on every cache access; the multi-prefetcher that was selected in the first level chooses the internal prefetcher based on the compressibility of the block (Table 4.1).

The prefetcher selector selects the best prefetcher based on the results of the usefulness predictors associated to each prefetcher (stride+neighbor-CLAP and NL+stride-CLAP). For computing the usefulness of a prefetcher in an interval, we exploit Equation 4.5.

$$\mathrm{U(interval)_n} = \frac{\mathrm{U(pref)_n}}{2} + \frac{\mathrm{U(interval)_{n-1}}}{2} \tag{4.5}$$

At the end of interval n, the prefetcher selector checks the prefetcher usefulness in cache content estimators ($\mathrm{U(pref)_n}$), that is computed using Equation 4.1. In order to compute the usefulness of the current interval ($\mathrm{U(interval)_n}$), we performs a moving average on

the current usefulness of the estimator ($U(pref)_n$) and the usefulness of the previous interval ($U(interval)_{n-1}$). We do not directly use the output of the estimator's usefulness ($U(pref)_n$) as the interval's usefulness ($U(interval)_n$) because we want to take into account the previous intervals to avoid sharp changes in number of useful prefetches in the current interval. A moving average gives more weight to the recent usefulness to make the result more accurate. Ultimately, the prefetcher selector decides globally which prefetcher to use based on $U(interval)_n$ and exploits the prefetcher with higher $U(interval)_n$ for the next interval.

4.1.3 Global adaptive CLAP evaluation

By taking advantage of the two proposed prefetch mechanisms, stride+neighbor-CLAP and NL+stride-CLAP, the usefulness predictor and the prefetcher selector, we propose a global adaptive compressed cache layout aware prefetching technique. The global adaptive CLAP enables only one of the prefetchers for the whole cache in each interval (either stride+neighbor-CLAP or NL+stride-CLAP) and switches between the prefetchers at the end of the intervals.

The prefetcher selector chooses either stride+neighbor-CLAP or NL+stride-CLAP, based on the result of the usefulness predictor (Section 4.1.1). As it was shown in Table 4.1, if stride+neighbor-CLAP is selected, depending on the result of the compression predictor (Section 3.2.2), it performs only stride prefetching or stride and neighbor prefetching. Likewise, if NL+stride-CLAP is selected for the interval, depending on the compression predictor result, it performs either only Nextline prefetching or Nextline and stride prefetching.

In this section, we evaluate the usefulness predictor, the prefetcher selector and the performance of our proposed global adaptive CLAP mechanism.

Usefulness predictor evaluation

As previously stated, we proposed a usefulness predictor (Section 4.1.2) to predict the usefulness of each prefetcher using Equation 4.1. In order to implement our usefulness predictor, we associate a cache content estimator (i.e. implemented using bloom filters) to each prefetcher to have an approximation of the cache. We name the prefetches in the bloom filters *simulated prefetches*, and the real prefetches in the cache are called *actual prefetches*.

In this section, we show our usefulness predictor is accurate enough and it is independent of the actual prefetcher. To do so, we consider four combinations of simulated and actual prefetchers and compare their usefulness in Figure 4.6. In this figure, for brevity, we refer to stride+neighbor-CLAP, as prefetcher A; and NL+stride-CLAP, as prefetcher B. Considering the first two bars in this figure, we compare the usefulness of simulated prefetcher A while using prefetcher A or B as the actual prefetcher. The results show that on average, the usefulness of simulated prefetcher A and actual prefetcher A is the same as the usefulness of simulated prefetcher A and actual prefetcher B. This indicates that the simulated prefetches in the bloom filters is independent of the actual prefetcher. The same explanation is applied to simulated prefetcher B in the last two bars of the figure. Nevertheless, in some benchmarks, such as *perlbench*, *milc*, *h264ref*, *tonto* and *lbm*, the usefulness of simulated prefetches changes depending on the actual prefetcher performing in the cache. As it was discussed in Section 4.1.1, although our usefulness estimation is independent of the actual prefetcher, but different actual prefetchers cause different evictions in the cache that changes the cache state used by the cache content estimators.

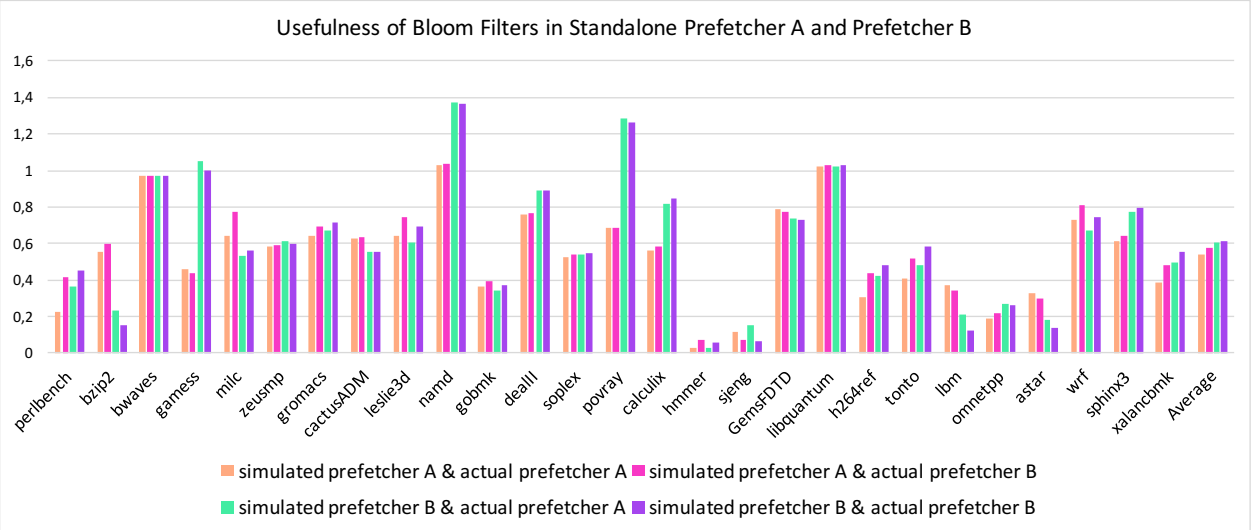


Figure 4.6 – Usefulness of bloom filters in standalone stride+neighbor-CLAP (prefetcher A) and NL+stride-CLAP (prefetcher B)

By using bloom filters as cache content estimators for simulated prefetches in usefulness predictor, we consider some approximations in our mechanism:

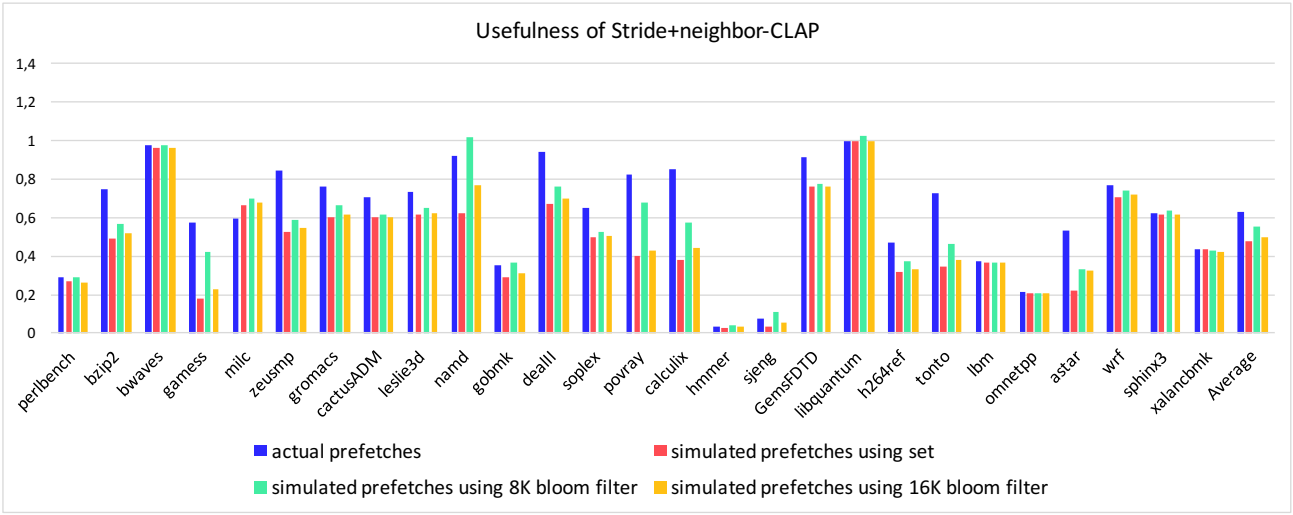
— Block lifetime: we determine the size of the intervals by approximating the life time of blocks that means how long a block stays in the cache. Considering the lifetime of blocks, we decide to reset the bloom filter (end of an interval) after a number of

evictions. The block lifetime can underestimate or overestimate the usefulness of a prefetched block.
  - Lifetime underestimation may happen if we remove an address from the bloom filter, while the block is still presented in the cache.
  - Lifetime overestimation may happen if we keep a block in the bloom filter, while the block is not presented in the cache and has been evicted already from the cache.
- Multiple accesses to the same block: There can be multiple accesses to a block that is already in the bloom filter; this leads to an overestimation of useful prefetches in the bloom filter.
- Conflicts: False positives can happen in a bloom filter due to address conflicts. If two or more addresses map to the same bits in the bloom filter, the number of useful prefetches is overestimated in the bloom filter.

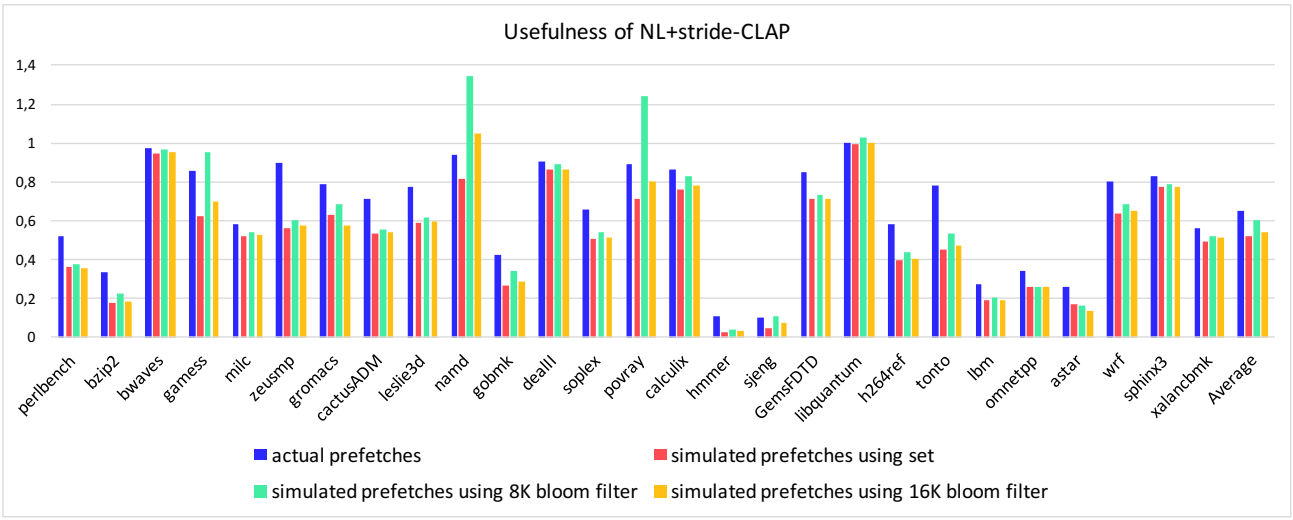
Considering aforementioned approximations, in order to evaluate the accuracy of our proposed usefulness predictor in global adaptive CLAP, we measure the usefulness of the simulated prefetches in bloom filters. In our experiments, we implement global adaptive CLAP with different size of bloom filters: 8K-entry and 16K-entry. Besides, in order to compare the accuracy of the bloom filters in our proposed usefulness predictor with an oracle usefulness predictor, we simulate a usefulness predictor with sets, instead of bloom filters. We dedicate a set to each prefetcher for estimating useful prefetches associated to the prefetcher. In this configuration of usefulness predictor, we apply the same eviction mechanism as the bloom filters.

Figure 4.7 shows the usefulness of each prefetcher in four configurations: actual prefetch usefulness of standalone prefetchers without selection mechanism, usefulness of actual prefetches in global adaptive CLAP using sets in usefulness predictor, usefulness of actual prefetches in global adaptive CLAP using 8K bloom filters in usefulness predictor, and usefulness of actual prefetches using larger bloom filters with the size of 16K. In this experiment, a new interval begins after 512 evictions in the cache.

As it is shown in Figure 4.7, in some benchmarks, such as *bwaves*, *libquantum*, *sphinx3* and *xalancbmk*, the usefulness of our proposed predictor (using either bloom filters or sets) is close to the usefulness of standalone prefetches (blue bars). Moreover, some benchmarks, such as *gamess*, *namd* and *povray* in global adaptive CLAP using 8K bloom filter, show an overestimation of usefulness that happen because of our approximations (mentioned at the beginning of this section). Overall, our proposed usefulness predictor (using either set



(a) Usefulness of stride+neighbor-CLAP



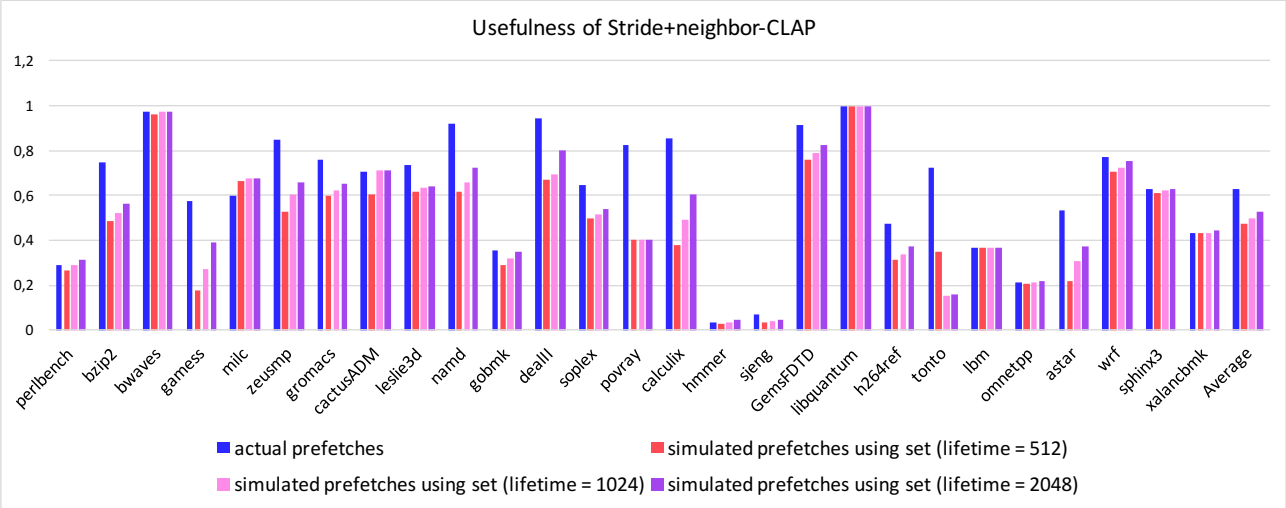
(b) Usefulness of NL+stride-CLAP

Figure 4.7 – Usefulness of stride+neighbor-CLAP and NL+stride-CLAP in four different configurations: standalone without selection mechanism, global adaptive CLAP using sets, 8K bloom filters and 16K bloom filters as selection mechanism

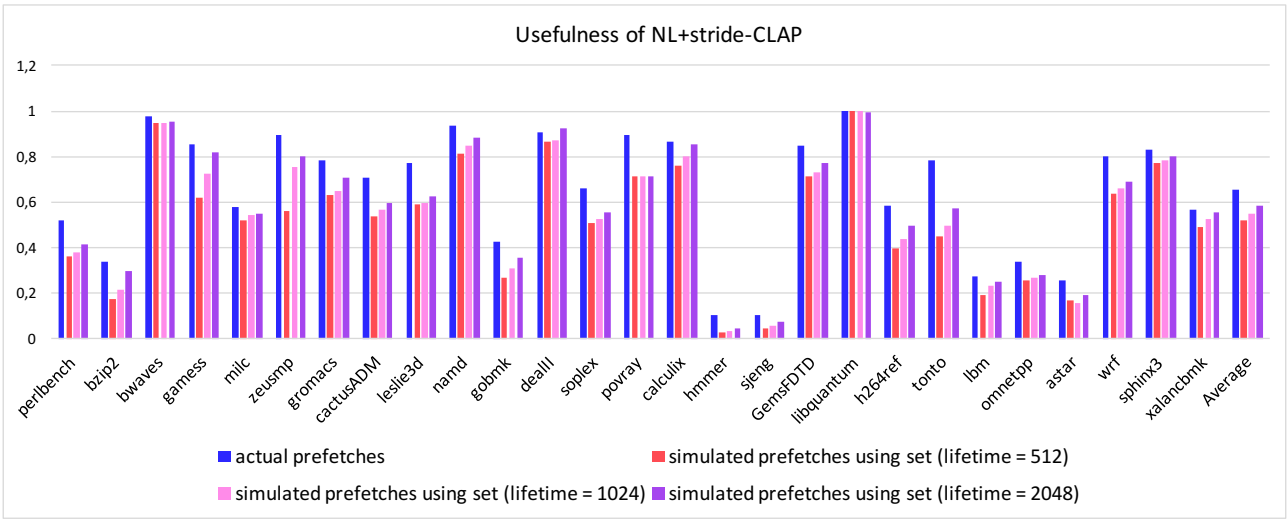
or bloom filter) underestimates the number of useful prefetches in most of the applications, comparing to the usefulness of individual prefetchers, in standalone configurations. These differences may occur due to the lifetime approximation of the blocks in the usefulness predictor. In all of these configurations of global adaptive CLAP, an interval ends after 512 block evictions in the cache.

In Figure 4.7, some benchmarks, such as *namd*, show a usefulness greater than one,

which is due to false positives that can happen in a bloom filter structure. As previously mentioned, false positives can happen because of conflicts or multiple accesses to the same address in the bloom filter. As it is shown in this figure, by increasing the size of the bloom filters from 8K to 16K, we can reduce the number of false positives.



(a) Usefulness of stride+neighbor-CLAP



(b) Usefulness of NL+stride-CLAP

Figure 4.8 – Usefulness of stride+neighbor-CLAP and NL+stride-CLAP in different configurations: standalone without selection mechanism, global adaptive CLAP using sets as selection mechanism with different lifetimes for the sets

As it is observed in Figure 4.7, simulating usefulness predictor using sets instead of bloom filters, gives better usefulness that is closer to the usefulness gained by performing

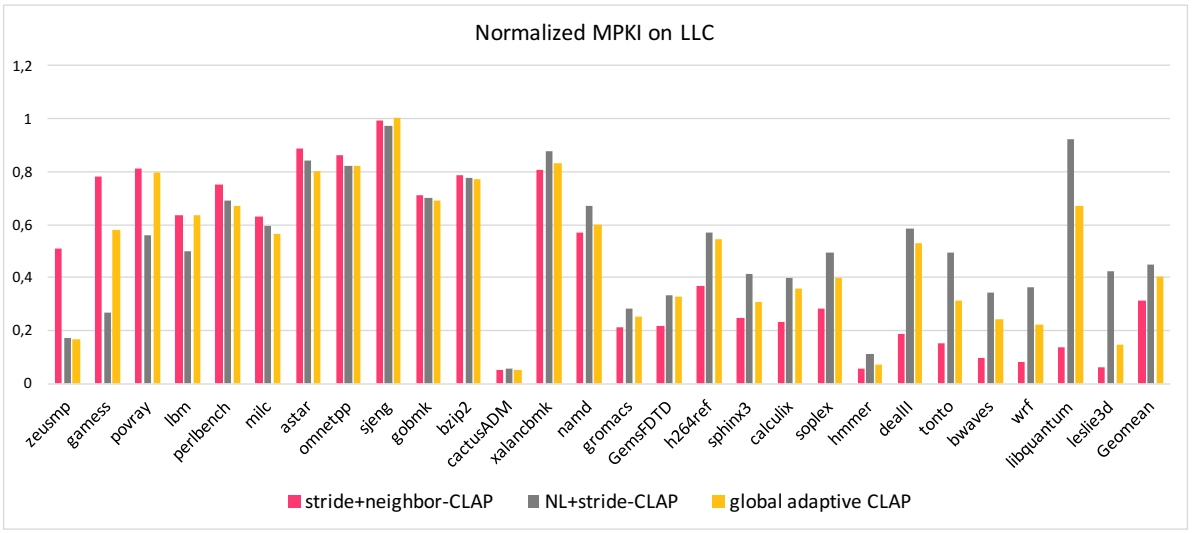
standalone prefetchers (actual prefetches). However, in most of the benchmarks, usefulness is underestimated due to lifetime approximations of blocks. Therefore, we increase the lifetime of blocks, which leads to increasing the time intervals. Figure 4.8 shows the usefulness of prefetchers considering eviction of 512, 1024 and 2048 blocks as a time interval, comparing to the usefulness of actual prefetches in standalone configuration.

Figure 4.8 indicates that on average, as we increase the lifetime of blocks (and consequently time intervals), we get better estimation of usefulness in each prefetcher. However, we can not increase the lifetime more than 2K due to the limited size of the bloom filter. Considering a 1MB CLLC and 64B cache block size, we have 16K blocks in CLLC; hence, the ideal lifetime for blocks in the usefulness predictor would be 16K. Nevertheless, we cannot set the time interval to 16K because then we need a larger bloom filter to store the prefetch addresses. Employing more than 16K-entry bloom filters, adds a large overhead to the hardware. Therefore, for the rest of the experiments of global adaptive CLAP, we employ four 16K-entry bloom filters with lifetime of 2K.

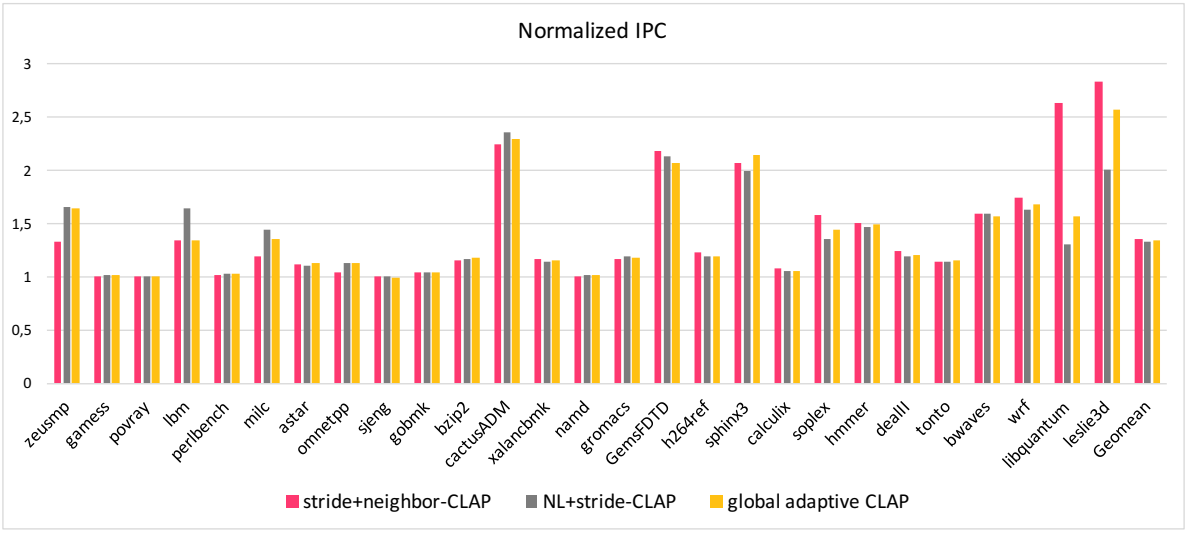
Global adaptive CLAP performance evaluation

In this section, we simulate global adaptive CLAP using either standalone prefetcher (stride+neighbor-CLAP and NL+stride-CLAP) and global adaptive CLAP. Figure 4.9 shows the number of LLC misses and also the performance improvement, normalized to compressed cache without prefetching, in global adaptive CLAP comparing to stride+neighbor-CLAP and NL+stride-CLAP. In this figure, in order to show each benchmark benefits the most from which prefetcher (stride+neighbor-CLAP or NL+stride-CLAPP), we compute the ratio of $\frac{MPKI_{stride+neighbor-CLAP}}{MPKI_{NL+stride-CLAP}}$ for each benchmark and sort the benchmarks according to this ratio. In this manner, benchmarks that benefits the most from NL+stride-CLAP are observed first and the ones that benefits the most from stride+neighbor-CLAP are last. As it is shown in Figure 4.9a in some benchmarks such as *zeusmp*, *perlbench*, *milc*, *astar*, *omnetpp*, *bzip2* and *xalancbmk*, adaptive CLAP reduces the number of misses or it has the same MPKI as the best configuration between the two other prefetchers. Moreover, on average, the number of LLC misses in adaptive CLAP is between the two other proposed prefetchers, which makes the stride+neighbor-CLAP the best of all prefetchers. However, on average, the IPC is almost the same in three configurations (Figure 4.9b).

It should be pointed out that we measured different parameters to adjust the prefetcher; however, the best prefetcher selector works based on prefetch usefulness. In our experiments, we tried steering the prefetcher based on cache pollution, late prefetches and useful



(a) Number of LLC misses normalized to compressed cache without prefetching



(b) LLC performance in terms of IPC normalized to compressed cache without prefetching

Figure 4.9 – Comparison of stride+neighbor-CLAP, NL+stride-CLAP and global adaptive CLAP, as numbers are normalized to compressed cache without prefetching

prefetches (without normalizing to total prefetches) as well; nevertheless, we observed that these metrics are not correlated to MPKI and IPC in our benchmarks.

Prefetcher selector evaluation

In order to verify the usefulness predictor functionality in global adaptive CLAP, we evaluate the usefulness of stride+neighbor-CLAP and NL+stride-CLAP individually. We simulate two separate configurations of compressed cache with prefetching: a compressed cache with only stride+neighbor-CLAP activated, a compressed cache with only NL+stride-CLAP activated. The usefulness for each standalone prefetcher, using Equation 4.1, is shown in figure 4.10.

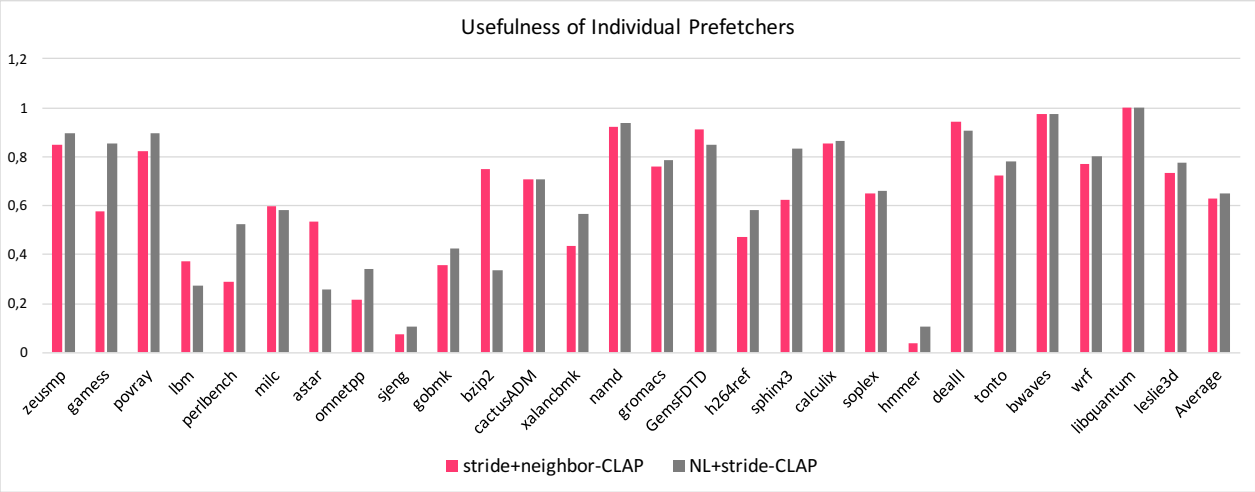


Figure 4.10 – Usefulness for standalone stride+neighbor-CLAP and NL+stride-CLAP

As it is shown in Figure 4.10, based on the application, sometimes stride+neighbor-CLAP has higher average usefulness and sometimes NL+stride-CLAP. In order to verify that global adaptive CLAP selects the prefetcher with higher usefulness and it performs more prefetches with the winner prefetcher, we simulate global adaptive CLAP, which employs stride+neighbor-CLAP and NL+stride-CLAP. Our proposed global adaptive CLAP switches between these two prefetchers in each interval. Figure 4.11 shows the number of prefetches carried out by each prefetcher.

In order to evaluate the functionality of usefulness predictor in global adaptive CLAP, we consider Figure 4.10 and 4.11. Based on Figure 4.10, we expect that for benchmarks such as *lbm*, *astar* and *bzip2*, stride+neighbor-CLAP, which has a higher usefulness than NL+stride-CLAP, is selected in the global adaptive CLAP configuration. Considering Figure 4.11 for the same benchmarks, stride+neighbor-CLAP is selected more often; therefore, the usefulness predictor steers the prefetcher well. There are some benchmarks, such as *zeusmp*, *omnetpp*, *xalancbmk*, *h264ref* and *tonto*, that NL+stride-CLAP have higher

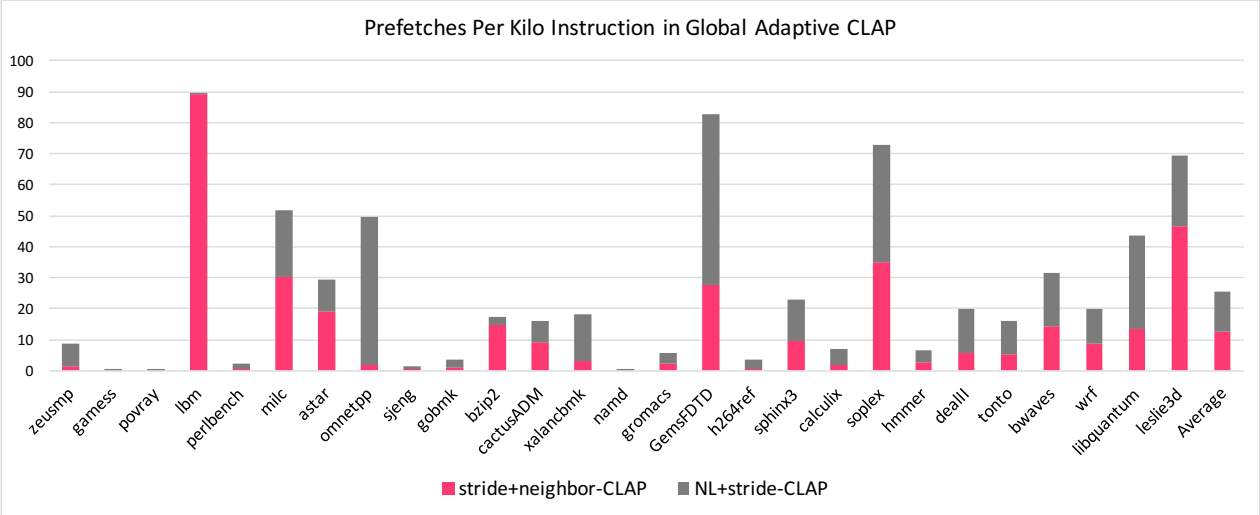


Figure 4.11 – Number of prefetches, per kilo instruction, in global adaptive CLAP for stride+neighbor-CLAP and NL+stride-CLAP

usefulness in Figure 4.10; and by taking a look at Figure 4.11, we observe that NL+stride-CLAP is triggered the most in these applications.

For three applications (*leslie3d*, *dealII* and *GemsFDTD*), the selected prefetcher in global adaptive CLAP in Figure 4.11 is not the same as the standalone prefetcher with higher usefulness in Figure 4.10. For *leslie3d*, stride+neighbor-CLAP is chosen more often in global adaptive CLAP (Figure 4.11); however, in standalone configuration, NL+stride-CLAP shows higher usefulness (Figure 4.10). By taking a look at Figure 4.7 that shows the usefulness in simulated prefetchers, we realize that our usefulness predictor estimates higher usefulness for stride+neighbor-CLAP; nonetheless, in the same figure, the usefulness of actual prefetches is higher in NL+stride-CLAP. The same explanation applies to *dealII*. For *GemsFDTD*, Figure 4.10 shows higher usefulness in standalone stride+neighbor-CLAP; nevertheless, Figure 4.11 indicates most of the prefetches are carried out using NL+stride-CLAP in the global adaptive CLAP. Based on Figure 4.7, stride+neighbor-CLAP is estimated to have higher usefulness in simulated prefetches (U(pref)), on average; and the estimation is close to the usefulness of standalone stride+neighbor-CLAP. However, this benchmark selects NL+stride-CLAP more often because in most of the time intervals, U(interval) of NL+stride-CLAP is higher than U(interval) of stride+neighbor-CLAP, which results in selecting NL+stride-CLAP as the best prefetcher. It is worth mentioning that as it was discussed in Section 4.1.2, U(pref) and U(interval) are two different metrics (Equations 4.1, 4.5).

4.2 Local adaptive CLAP

As it was shown in Figure 4.9a, global adaptive CLAP has an MPKI between the MPKI of stride+neighbor-CLAP and NL+stride-CLAP; however, we expect the lowest MPKI in the adaptive configuration. Global adaptive CLAP's decision to steer the prefetcher is based on the average of useful prefetches in a previous interval. This global decision is made for the whole cache and the selected prefetcher is activated until the interval ends and then, the prefetcher selector recalculates the usefulness of each prefetcher and chooses the prefetcher with higher usefulness. However, the global adaptive CLAP does not work well due to different characteristics of different loads in applications. Therefore, it can be more beneficial if we steer the prefetcher for each load access individually.

In this section, the objective is to select the prefetcher locally, based on each individual memory access. In local adaptive CLAP, prefetcher selection occurs among three simple prefetching mechanisms that are stride prefetching, Nextline prefetching and neighbor prefetching. In this configuration, we can take advantage of different prefetchers based on two factors: stride confidence and block's compressibility.

As mentioned earlier in Section 4.1.2, the global adaptive CLAP has a two-level decision to select the best prefetcher. The first level of decision is to choose between two multi-prefetchers (stride+neighbor-CLAP and NL+stride-CLAP) for each interval and the second level is to choose between the sub-prefetchers of stride+neighbor-CLAP and NL+stride-CLAP on each cache access. In local adaptive CLAP, we can avoid the two-level decision making procedure by steering the prefetcher only on each access. Local adaptive CLAP steers the prefetcher based on the characteristic of each individual load.

4.2.1 Local adaptive CLAP architecture

Our objective is to implement local adaptive CLAP by making the prefetching decision based on two factors of stride confidence and compression. Stride prefetcher is implemented in CLLC and has a PC table to record the prefetch strides. On the other hand, CLAP exploits the compression predictor, in CLLC, that has a PC table as well to monitor the compressibility of blocks. Therefore, we can merge these two tables and create a new guidance for steering the prefetcher in local adaptive CLAP.

We implement a prefetcher selection that works as follows: The local adaptive CLAP takes advantage of both stride confidence and compressibility of the block to select the best prefetcher and create a positive interaction in the prefetching system. This is illustrated

in Figure 4.12.

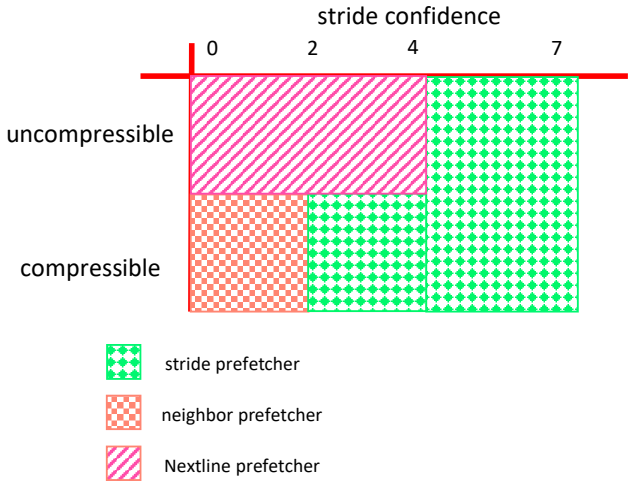


Figure 4.12 – The prefetching map of local adaptive CLAP

This diagram has stride confidence on one axis and compressibility on the other one. Moreover, it shows three different prefetching techniques that can be triggered depending on stride confidence and compressibility of the block. This configuration of local adaptive CLAP consists of stride prefetcher, neighbor prefetcher and Nextline prefetcher. The prefetchers in local adaptive CLAP are activated under different conditions, in CLLC, depending on the value of the two factors:

— **Stride prefetcher** has a table indexed by program counter. On every cache access, it calculates the prefetch stride using the current address and the last referenced address. Furthermore, there is a confidence variable for each stride that gets incremented if the new stride matches the previous calculated stride and gets decremented if they do not match. The stride prefetcher is activated in two different conditions:

1. If the stride's confidence is high enough (higher than a threshold that is 4), the stride prefetcher prefetches a number of blocks based on the prefetching degree (i.e. 16 prefetches in our simulations). This means that we are almost sure that the data that is going to be prefetched will be useful in the near future and this decision is regardless of compressibility of a block. In other words, if a block is going to be used later in the program, it does not matter if it is compressible or not. As we will show in Section 4.2.2, the data prefetched by stride prefetcher is likely to be useful by 77%.

2. If the stride's confidence is between 2 and 4, and the block is predicted to be compressible (using the compression predictor), it is still worthwhile to perform stride prefetching. This prefetcher is the stride-CLAP introduced in Section 3.4. If the stride confidence is between 2 and 4, it is useful to perform stride prefetching (16 blocks ahead) only on compressible blocks. If the stride confidence is not high enough, the prefetched blocks may not be useful in the program; however, if we prefetch compressible blocks that can be co-allocated in one super-block, it is still useful to prefetch them without extra space cost. It is noteworthy to mention that performing stride prefetching on uncompressible blocks may cause pollution in the cache due to evicting useful data. 60% of the blocks that are prefetched using stride prefetcher, when the data is compressible, are useful in the program (Section 4.2.2).

— **Neighbor prefetcher** is only triggered when the predictor predicts compressibility and also the stride confidence is below 2. On every access, by checking the PC table that contains both stride confidence and compressibilty of the load address, neighbor prefetcher decides to prefetch the three neighbor blocks or not.
In case the stride confidence below 2, it does not worth it to carry out stride prefetching; nonetheless, prefetching neighbor blocks that are co-allocated in one super-block is helpful. Prefetching blocks that are not compressible may cause detrimental effects due to cache pollution. As we will discuss in Section 4.2.2, 46% of the prefetched data by neighbor prefetcher is useful during the execution of the program.

— **Nextline prefetcher** prefetches the next block. If none of the above conditions are met, it is favourable to prefetch only the next block. Hence, when the stride confidence is low and also the block is predicted to be uncompressible, Nextline prefetcher is applied to CLLC. In Section 4.2.2, we will show that among all the prefetched data, using Nextline prefetcher, 44% of them are likely to be useful. Furthermore, 11% of these useful prefetches are late due to the Nextline prefetcher behaviour.

Unlike the prefetcher selector in global adaptive CLAP (Section 4.1), which steers the prefetcher globally regarding the average number of usefulness, local adaptive CLAP adjusts the prefetcher locally based on the PC triggering the access. In this configuration different prefetches are activated during different circumstances without any overlap.

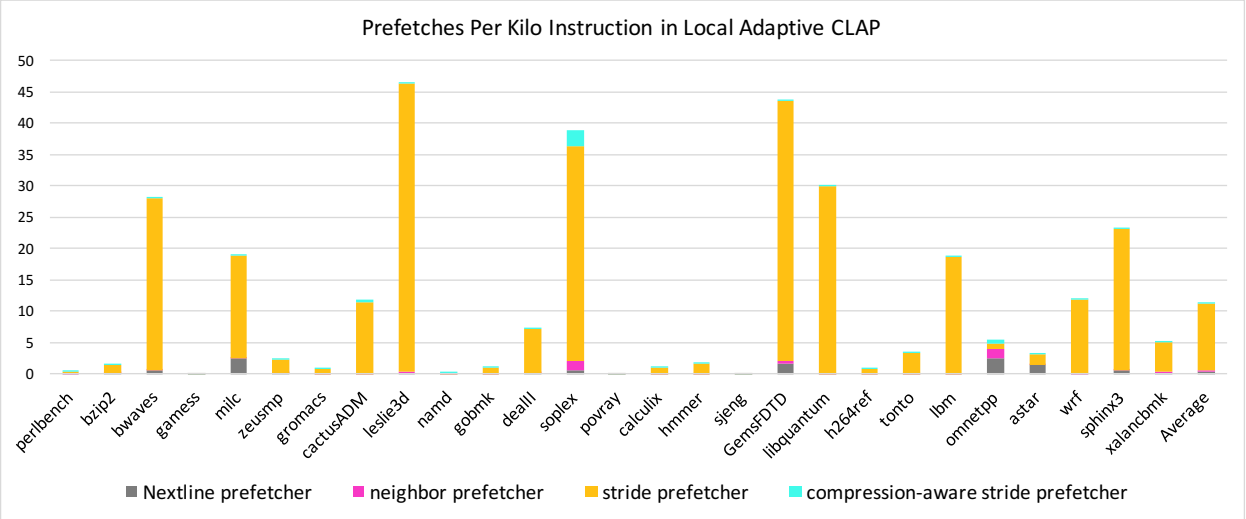


Figure 4.13 – Number of prefetches, per kilo instruction, in local adaptive CLAP for each individual prefetcher

4.2.2 Usefulness in local adaptive CLAP

In this study, for evaluating local adaptive CLAP, we measure the number of useful and unused prefetches (Section 4.1.1). Useful prefetches can be either early or late. An early useful prefetch is a prefetch that arrives to the cache before there is another request for it. Nevertheless, if the prefetched data arrives to the cache after the request, it counts as a late prefetch but still a useful one.

We simulate the local adaptive CLAP in LLC; Figure 4.13 shows the number of prefetches in each individual prefetcher in local adaptive CLAP. This figure indicates that most of the prefetches are cause by stride prefetcher.

In order to evaluate the usefulness of each prefetcher in local adaptive CLAP (with the prefetcher selector mechanism), we measure the early, late and unused prefetches in Figure 4.14 to 4.17. It should be emphasized that we count both early and late prefetches as useful ones.

Figure 4.14 shows the usefulness of Nextline prefetcher in the local adaptive CLAP configuration; only 44% of prefetches are useful. Furthermore, among these useful prefetchers, 11% of them are late because the system is prefetching the next block (only one block ahead) and by the time this prefetch is serviced, there maybe a demand request for the same address.

The usefulness of neighbor prefetcher is shown in Figure 4.15. This prefetcher attains 46% useful prefetches, on average. In local adaptive CLAP configuration, the neighbor

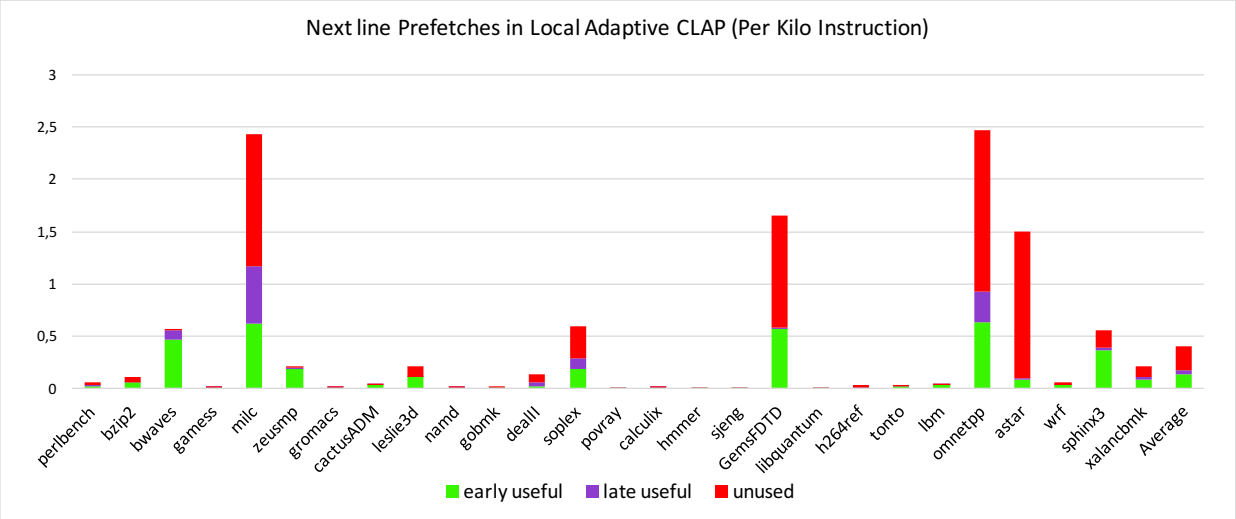


Figure 4.14 – Number of Nextline prefetches in local adaptive CLAP configuration

prefetcher is enabled only when the block is predicted to be compressible and also there is a very low confidence in the stride. This case rarely happens and that explains the low number of total prefetches.

The local adaptive CLAP selector chooses the stride prefetcher as the best prefetcher if there is a high confidence in the stride. When there is enough confidence for a stride, a great number of prefetches are going to be useful in the cache. Based on Figure 4.16, on average, 77% of the prefetches are useful and only 3% of them are late prefetches. The high percentage in prefetcher's usefulness and having low late prefetches is achieved by prefetching up to 16 blocks ahead in time (defined as prefetch degree), and based on the high confidence stride.

If the block is predicted to be compressible and the stride confidence is between 2 and 4, the compression-aware stride prefetching mechanism is chosen as the best prefetcher. Figure 4.17 illustrates the number of early useful, late and unused prefetches which are 57%, 3% and 40% (on average), respectively.

The results of these experiments indicate that stride prefetcher has the most impact on the number of useful prefetches since it prefetches more blocks.

Figure 4.18 shows the details on useful (early and late) and unused prefetches in the stride prefetcher of adaptive CLAP. This figure divides the total number of prefetches into two categories based on the result of the compression predictor which predicts the compressibility of a demand block. As it is illustrated in this figure, most of the prefetches happen when the predictor predicts the block is not compressible, except for the bench-

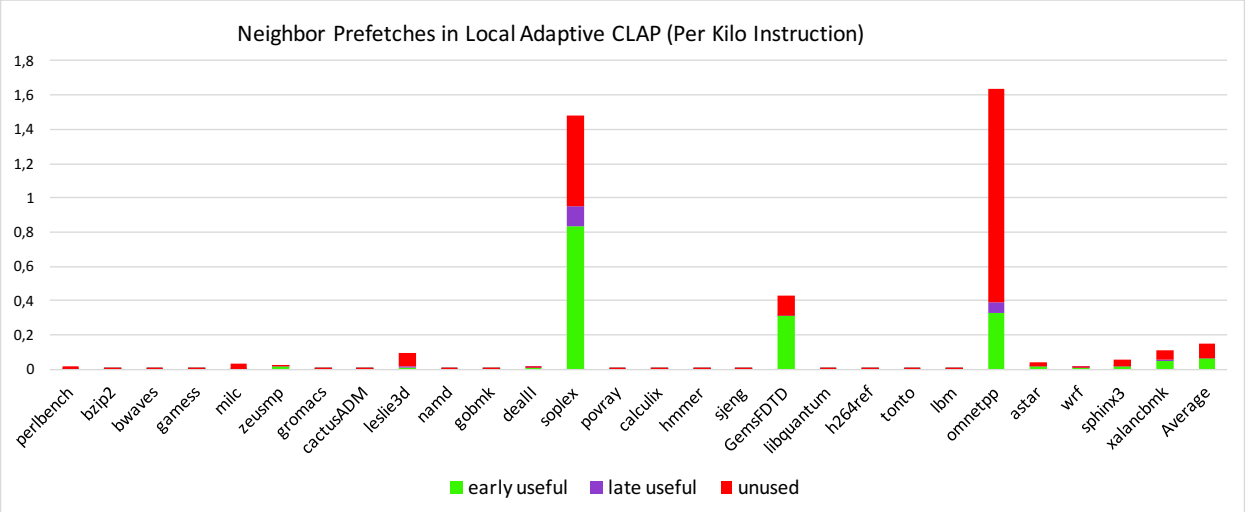


Figure 4.15 – Number of neighbor prefetches in local adaptive CLAP configuration

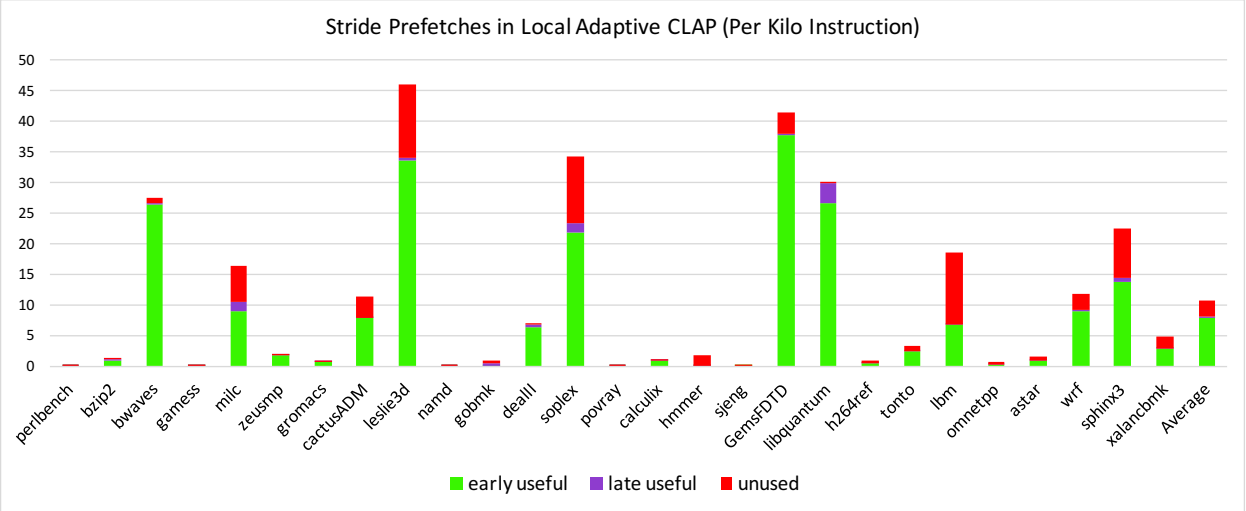


Figure 4.16 – Number of stride prefetches in local adaptive CLAP configuration

mark *soplex* that has a high compression factor.

4.3 Adaptive CLAPs comparison

Based on the results shown in Section 4.2.2, it can be concluded that stride prefetcher prefetches the most and it carries out a large number of useful prefetches in the local adaptive CLAP layout. This can explain better performance of the stride+neighbor-CLAP in Figure 4.9 among other prefetchers. However, in order to get more improvements, we

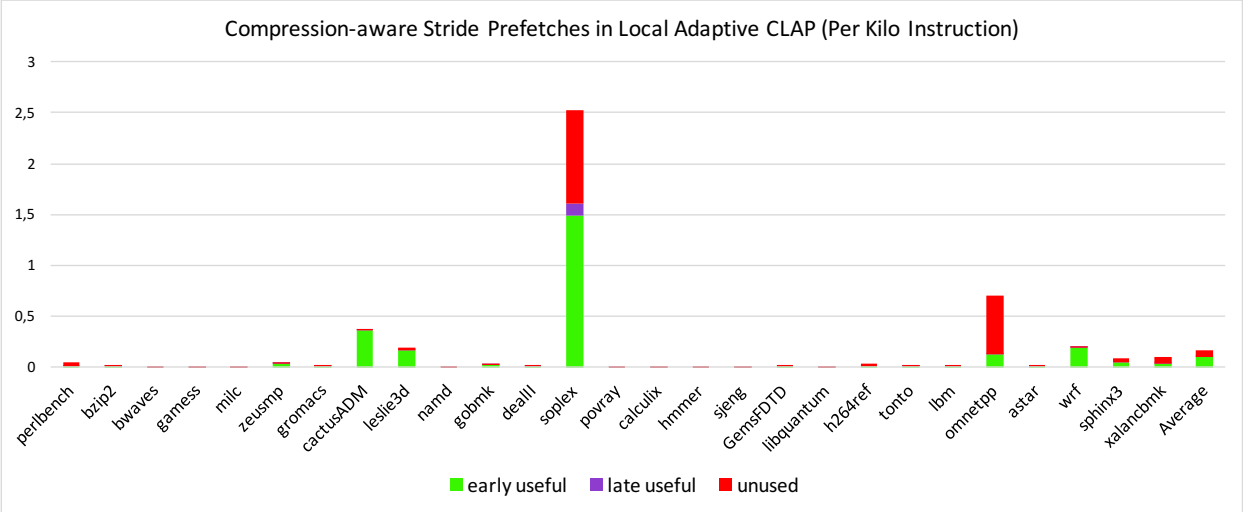


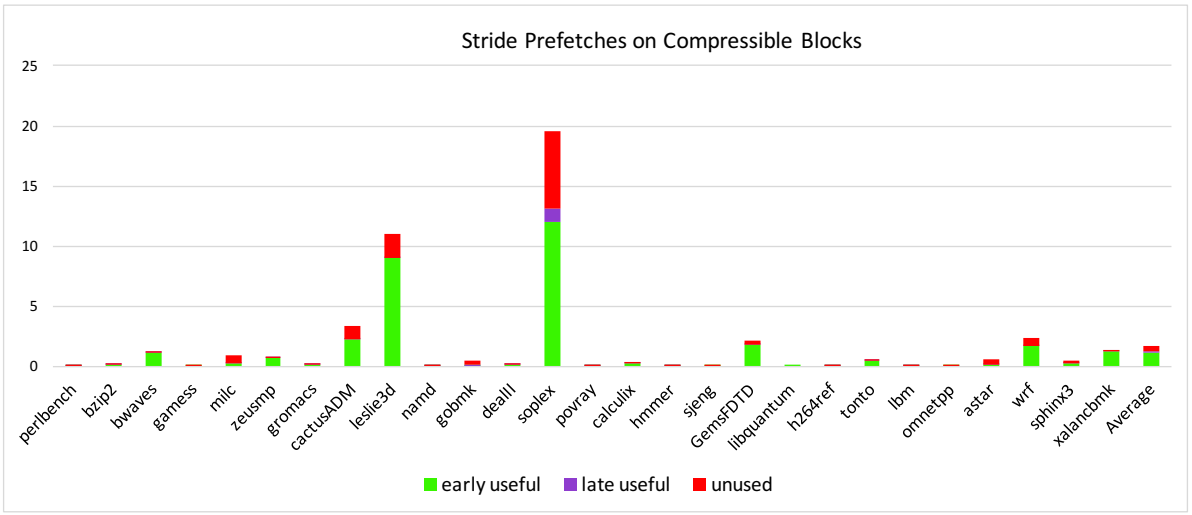
Figure 4.17 – Number of compression-aware stride prefetches in local adaptive CLAP configuration

implemented local adaptive CLAP. Figure 4.19 indicates the comparison of this configuration to other proposed prefetchers shown in Figure 4.9. On average, utilizing the local adaptive CLAP reduces number of LLC misses by 7.4%, 34% and 25% comparing to stride+neighbor-CLAP, NL+stride-CLAP and global adaptive CLAP, respectively. Furthermore, by taking advantage of local adaptive CLAP in CLLC, the performance improves by 1.5%, 3.2% and 3.2% comparing to the three other configurations.

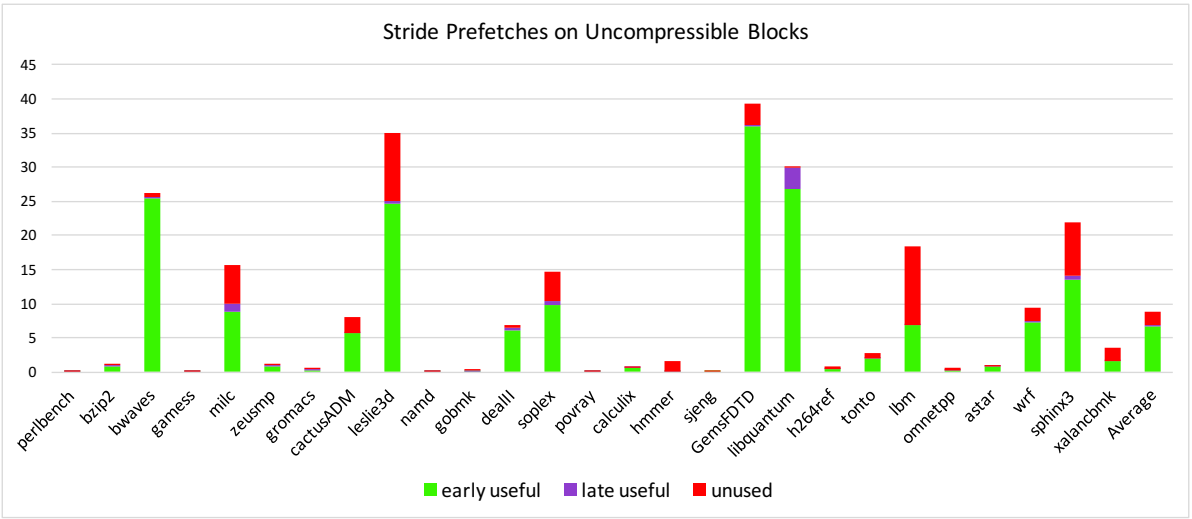
Figure 4.20 shows the number of LLC demand misses, in terms of MPKI, in a compressed cache without prefetching, compared to compressed caches with the global adaptive CLAP and the local adaptive CLAP layouts. Comparing to a compressed cache without prefetching, we can reduce the number of misses by 55% and 66% using global adaptive CLAP and local adaptive CLAP, respectively.

In this figure, the local adaptive CLAP decreases the number of LLC misses significantly in some benchmarks, such as *GemsFDTD* and *libquantum*, comparing to the global adaptive CLAP. Considering these benchmarks in Figure 4.18; the stride prefetcher prefetches compressed blocks more than uncompressed ones. Hence, the local prefetcher, that performs stride prefetching on all data regardless of compressibility, outperforms the global prefetcher, that carries out stride prefetching mostly on compressed data. In other words, benchmarks such as *bwaves*, *leslie3d*, *GemsFDTD* and *libquantum* have more stride accesses to uncompressed data rather than compressed blocks.

The number of write backs in LLC is shown in Figure 4.21. A comparison of different



(a) Number of stride prefetches happening on a block predicted to be compressible

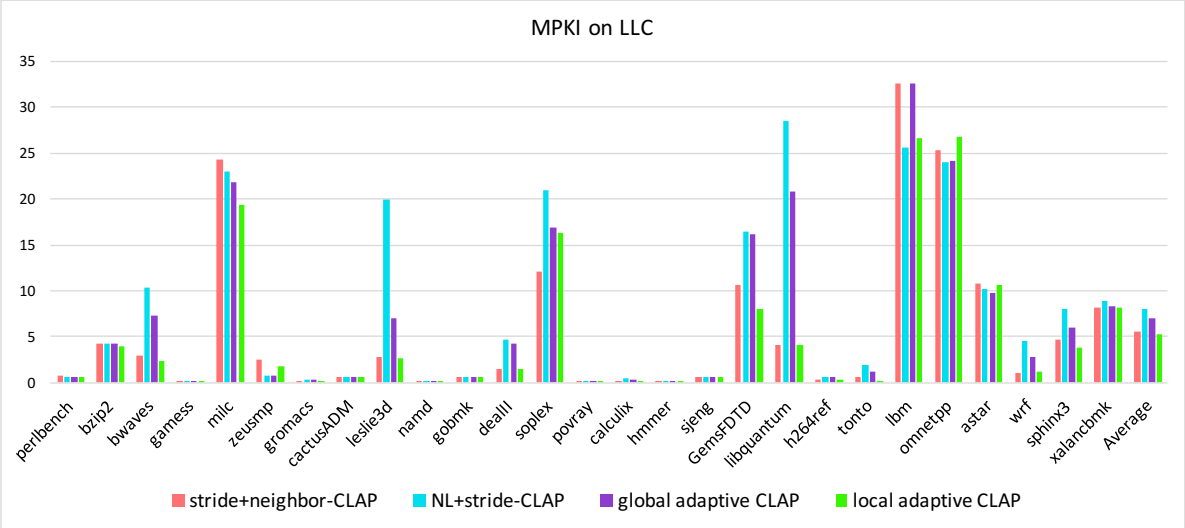


(b) Number of stride prefetches happening on a block predicted to be uncompressible

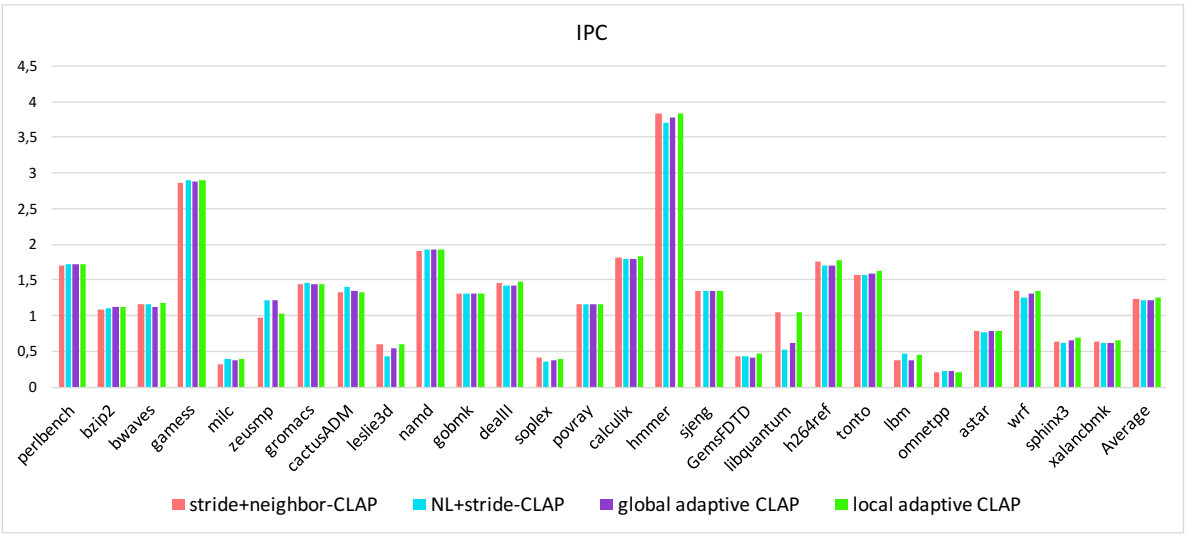
Figure 4.18 – Number of stride prefetches, in local adaptive CLAP, based on block's compressibility

compressed cache configurations is shown in the figure: without prefetching, with global adaptive CLAP, and with local adaptive CLAP. Using the global adaptive CLAP and the local adaptive CLAP, increases the number of write backs by only 1.3% and 1%, respectively, comparing to the layout without prefetching.

As it was shown in Figure 4.20, global adaptive CLAP and local adaptive CLAP decrease the number of LLC misses by 55% and 66%, respectively, comparing to a layout



(a) Number of LLC misses in terms of MPKI



(b) LLC performance in terms of IPC

Figure 4.19 – Comparison of stride+neighbor-CLAP, NL+stride-CLAP, global adaptive CLAP and local adaptive CLAP

without prefetching; nevertheless, they increase the memory traffic by 8.7% and 5.8% (Figure 4.22).

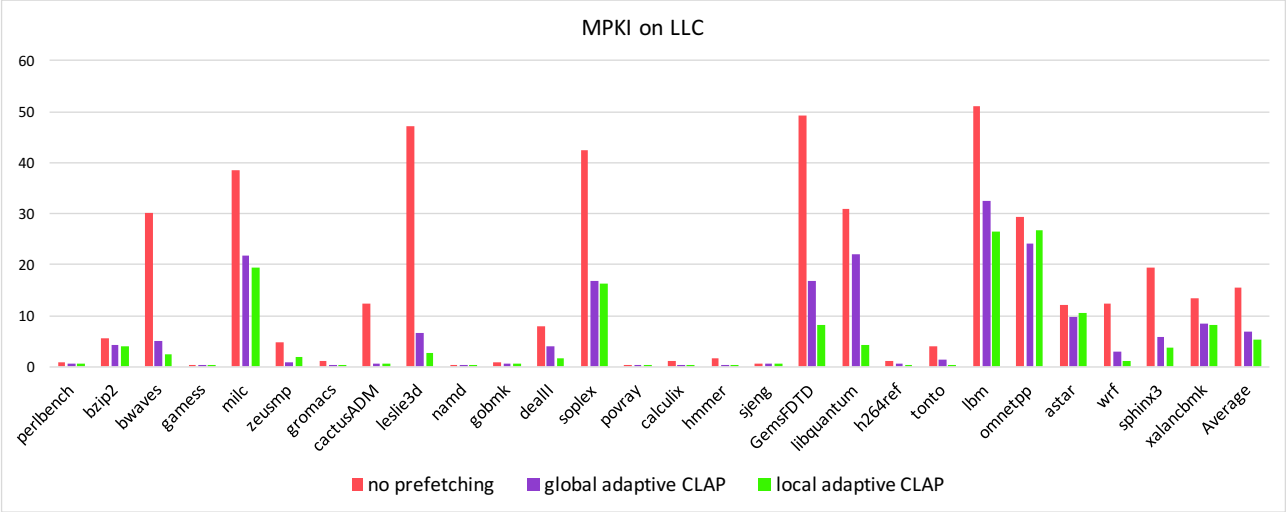


Figure 4.20 – The number of LLC misses per kilo instruction in a compressed cache without prefetching, with global adaptive CLAP and with local adaptive CLAP

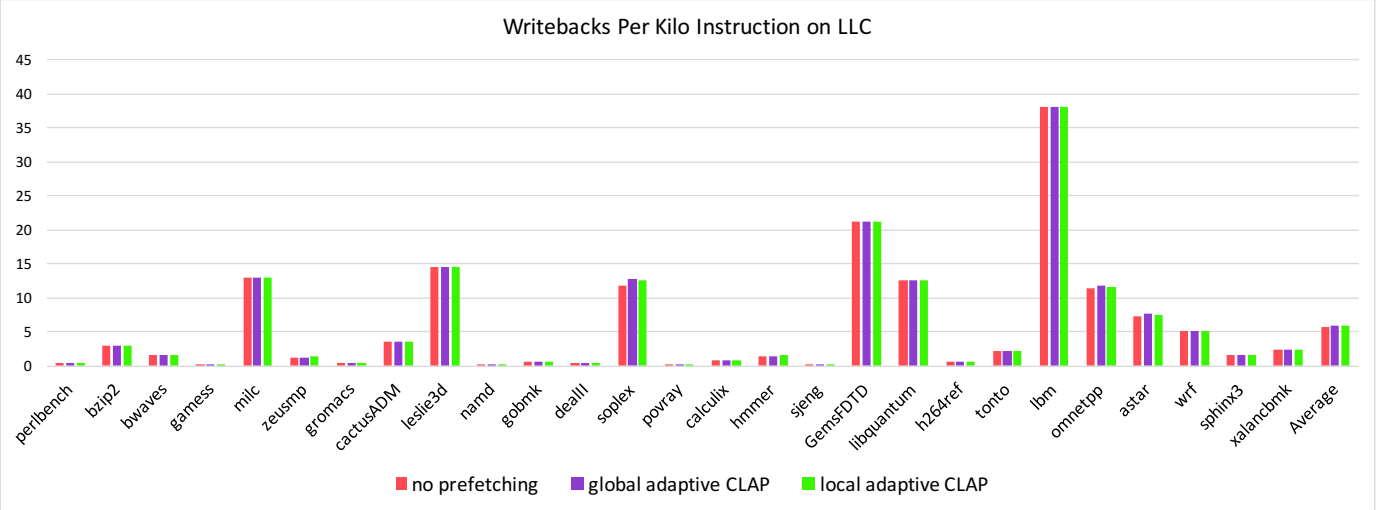


Figure 4.21 – The number of LLC write backs per kilo instruction in a compressed cache without prefetching, with global adaptive CLAP and with local adaptive CLAP

4.4 Summary

This chapter presents two adaptive compressed cache layout aware prefetchers that steer the appropriate prefetcher dynamically depending on the application. The global adaptive CLAP selects between two compression-aware prefetchers. Based on the experiments, we observe that in some applications the first prefetcher outperforms the second one and in some other applications, the second one is performing better. Thus; we pro-

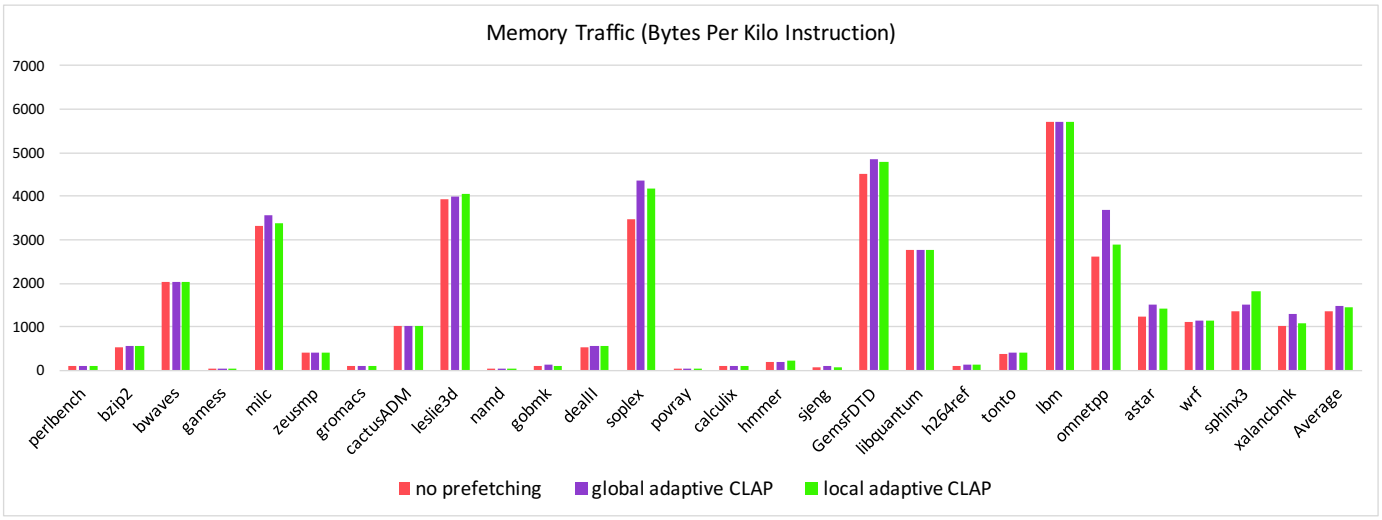


Figure 4.22 – Memory traffic, in terms of bytes per kilo instruction, in a compressed cache without prefetching, with global adaptive CLAP and with local adaptive CLAP

pose a prefetcher selector that decides which prefetcher to choose dynamically based on usefulness of the prefetches during intervals.

In the second part of this chapter, we propose a local adaptive CLAP architecture. This configuration takes benefit from various prefetchers such as stride prefetcher, compression-aware stride prefetcher, neighbor prefetcher and Nextline prefetcher. The selection among these prefetching methods is done locally based on each memory access. Afterwards, we evaluate usefulness for each prefetcher individually. It could be deduced that the stride prefetcher has the most impact on the performance of the local adaptive CLAP configuration. The chapter ends with a comparison of all proposed prefetchers. The results show a reduction in number of LLC misses and also a better performance in the local adaptive CLAP configuration.

Chapter 5

Conclusion

Due to the speed gap between processors and memories, the memory system becomes a crucial bottleneck in processor design. Off-chip memory latency is one of the major problems in this area and multiple approaches have been proposed in order to confront this issue, such as compression and prefetching.

However, prefetching and compression suffer from some drawbacks when they are used solely in a system. Prefetching may cause cache pollution by bringing useless data to the cache and evicting useful data to make space for the prefetched blocks. On the other hand, compression and decompression units add latency and complexity to the system.

In this thesis, our goal was to leverage both compression and prefetching techniques to create a synergy between them and make a trade-off between the cache capacity and the latency. Prefetching reduces the latency but it decreases the cache capacity, simultaneously, by bringing data into the cache. On the other hand, compression increases the capacity of the cache; however, it adds latency because of the decompression logic. In our proposed approach, due to the decompression latency of the compression unit, we considered applying compression algorithm on the last level cache (LLC) to avoid adding the latency overheads to L1 and L2 caches.

We introduced a compressed cache layout aware prefetcher (CLAP) that makes a trade-off between cache compression and data hardware prefetching. The key idea of this study is to prefetch the blocks that are likely to be compressible. These blocks are the neighbors of the demand block and due to spatial locality, they will be accessed in the future with a high likelihood. CLAP reduces the off-chip memory latency by decreasing the number of accesses to the memory.

Employing a single prefetcher to all benchmarks is not always advantageous. Some applications need more aggressive prefetchers and some less aggressive. Therefore, we leveraged CLAP and proposed two adaptive prefetchers that are able to steer the prefetcher dynamically, using a prefetcher selector, through the execution time of the program. We presented a global adaptive CLAP that adjusts a prefetcher, between two CLAP-based

prefetchers, globally based on the prefetchers usefulness. Besides, we proposed a local adaptive CLAP which selects the best prefetcher, among four prefetching techniques, locally for each memory access depending on its characteristics: stride confidence and compressibility.

This chapter presents the contributions of the thesis. Furthermore, we discuss the opportunities for the future work.

5.1 Contributions

We showed that there is an interaction between cache compression and hardware prefetching, and we discussed the opportunities for prefetching in compressed cache layouts. Due to bulk evictions in compressed caches, a single super-block eviction can generates up to four subsequent misses; thus, prefetching neighbor blocks of the demand block is beneficial.

As the first contribution of this dissertation, we show that if the demand block and its neighbors are compressible and co-allocatable in one data entry, we can prefetch and insert them in the cache to avoid the future misses. On the block allocation for the demand request, a super-block is allocated for the block; therefore, prefetching and inserting the co-allocatable neighbor blocks occur without any extra cost in space. We show that we can reduce the number of LLC misses (up to 20%) by prefetching compressible blocks.

The second contribution of the thesis is to find the prefetching candidates in compressed caches. In order to do so, we propose a compression predictor to predict whether a block is compressible. Our predictor operates based on a stream of memory accesses of a given load instruction. If a load instruction accesses a compressible block, it is likely that the future accesses to the same instruction will access a compressible block.

We observe that predicting based on compression outperforms predicting based on co-allocatability. If we prefetch compressible blocks, regardless of their compactability in the super-block, it results in prefetching more blocks that are going to be useful in the future and reducing the number of LLC misses. Furthermore, we show that our stream-based compression predictor is accurate enough to predict the compressible blocks in a program and it has a negligible hardware overhead (0.58% of the 1MB LLC).

Proposing a compressed cache layout aware prefetcher (CLAP), a unified system that takes advantage of cache compression and hardware prefetching, is the third achievement of this work. By applying CLAP on all accesses, if the block is predicted to be compressible,

we can prefetch multiple blocks to LLC. Neighbor-CLAP prefetches contiguous blocks that are neighbor with the accessed block. Stride-CLAP prefetches a stream of accesses that predicted to be compressible. Based on our experiments, CLAP outperforms compressed cache by 9%, on average, (up to 35%) in terms of MPKI and it gains an average speedup of 3% (up to 20%).

Moreover, we show that our proposed techniques (neighbor-CLAP and stride-CLAP) can be combined with other prefetchers such as stride and Nextline. By applying stride-CLAP in parallel with a Nextline prefetcher, the number of LLC misses decrease by maximum of 9%, on average, and the performance enhances up to 6%, comparing to a system that utilizes Nextline-only prefetcher. Taking advantage of stride prefetcher and neighbor-CLAP side by side reduces the LLC MPKI by 8%, on average, and it achieves a maximum speed up of 13%, comparing to a stride-only configuration.

Different applications have different characteristics and employing a single prefetcher does not give the best performance. For instance, a stride prefetcher, that prefetches 16 blocks, may be useful for one application but disadvantageous for another one due to bringing lots of useless data and causing cache pollution.

The fourth contribution of the dissertation is implementing a global adaptive CLAP that dynamically switches between two CLAP-based prefetchers to adjust the aggressiveness of the prefetching mechanism. The metric that is used to steer the prefetcher is the prefetch usefulness which is the number of useful prefetches divided by the number of total prefetches sent to the memory. The prefetcher with a higher usefulness is chosen to carry out prefetching for the whole cache in the next interval. The intervals are defined by the number of evicted blocks from LLC. Based on our experiments, we use a value of 512 evicted blocks for an interval; therefore, after each 512 block evictions, the best of the two prefetchers is triggered and a new interval starts.

Finally, the last contribution of this study is proposing a local adaptive CLAP. The global adaptive CLAP adjusts a prefetcher for the whole cache; however, it is more favourable if we can select the proper prefetcher for each memory access. For this purpose, we activate a prefetcher according to two factors: stride confidence and block's compressibility. By taking advantage of these two factors, simultaneously, we can select the best prefetcher for each individual memory access. The prefetcher is selected among four different approaches: stride, compression-aware stride, Nextline and neighbour prefetching techniques. Based on the experimental evaluations, local adaptive CLAP reduces the number of LLC misses by 25% and enhances the performance by 3.2%, comparing to the global

adaptive CLAP.

Based on the observations, we conclude that CLAP creates a synergy between cache compression and hardware prefetching. In addition, the proposed global and the local adaptive CLAPs improve the system performance due to adjusting the prefetcher behaviour dynamically during the execution of the program.

5.2 Perspectives

Our research on compressed cache layout aware prefetching and the contributions can open new doors to further studies on compression and prefetching. In this section, we present the potential researches as future work: improving usefulness predictor in global adaptive CLAP, utilizing CLAP in a compressed memory hierarchy and employing CLAP in multi-core systems.

Improving usefulness predictor in global adaptive CLAP

As we discussed about the lifetime of blocks in usefulness predictor in Section 4.1.3, the ideal time interval to reset the bloom filters would be each 16K block evictions, which requires large bloom filters; nevertheless, the bloom filters in usefulness predictor can not be larger than 16K due to adding extra hardware overhead. A solution can be exploiting time intervals of 16K block evictions and keeping bloom filters size as 16K but not storing all the simulated prefetch addresses in the bloom filters. If we do not want to increase the size of the bloom filters, we can not store all the prefetched addresses; hence, we need to store a part of them using sampling. This approach does not store all the simulated prefetch addresses, so when there is an access to a simulated prefetched data block that had not been stored in the bloom filter in the first place, it can not not counted as a useful prefetch. Thus, in order to enhance the accuracy and have a good estimation of usefulness, we need to find a trade-off between the approximation of the blocks lifetime and the sampling method.

CLAP in a compressed hierarchy

We could apply compression not only on LLC, but also on L2. By having L2 compressed, we can transfer compressed data between L2 and L3 and save bandwidth. There

are some works carried out on cache and memory compression [PS16; Sha+14] and proposing a unified compressed memory hierarchy [HR05].

We could go further in our study and implement CLAP in a unified compressed memory hierarchy, and compress L2 cache and the memory as well. Therefore; the data need to be compressed only once and could be sent in compressed format among L2, LLC and main memory. However, implementing compression on all levels of the memory hierarchy requires significant modifications to the simulator, including cache coherency handling. Besides, our compression predictor would be useful for accessing the data in a compressed memory. If the predictor predicts that a block is compressible, it can prefetch it directly from the compressed memory without accessing the metadata. Metadata keeps the compression status information of all the blocks in the memory and it is allocated in a region of the memory.

In this thesis on every access, if the block is predicted to be compressible, the cache sends three prefetch requests to prefetch the neighbor blocks; nevertheless, by exploiting CLAP in LLC of a compressed hierarchy, if the block is predicted to be compressible, the cache sends a single prefetch request to the memory and it prefetches the whole super-block, including the neighbor blocks. Thus, instead of sending three requests for each neighbor block, the cache sends only one request to prefetch the super-block, which results in reducing the bandwidth traffic.

An interesting avenue for future research would be to apply CLAP on other compressed cache layouts. In this thesis, we implement CLAP based on YACC [SSW16] layout; however, it is applicable to other layouts such as SCC [SSW14], DCC [SW13] and Touché [Hon+19]. In these layouts, we could exploit our compression predictor to predict whether a given load access is accessing a compressible block. If the block is predicted to be compressible, we can activate the prefetcher, regardless of the prefetching mechanism, and start prefetching the blocks that are likely to be compressible, as the requested block.

We can take advantage of CLAP in SCC and due to the similiarities between SCC and YACC (except the indexing policy), we can expect the same improvement in SCC, as YACC. Employing CLAP in DCC is more complicated. In DCC, a data entry is shared by two tags, i.e. two super-blocks. By activating CLAP in DCC, we may end up prefetching the blocks correlated to a single super-block. These blocks occupy the space reserved for the second super-block; thus, CLAP may cause cache pollution in DCC layout. Touché is not a super-block-based compression layout; nevertheless, CLAP might be useful. For instance, considering a stride-CLAP interacting with Touché layout; the stride prefetcher

is triggered whenever a block is predicted to be compressible and it prefetches a stream of blocks ahead. However, there is no guarantee that they will be co-allocatable in one data entry. Further studies is required for this layout.

Furthermore, CLAP is independent of the compression method. In this dissertation, we employed DISH [PS16] algorithm because of its low decompression latency; nonetheless, other cache compression techniques, such as C-pack [Che+09], BDI [Pek+12], ... are applicable to our work.

CLAP in multi-core systems

Another possible approach for the future work could be the implementation of CLAP and adaptive CLAPs in multi-core architectures. Sending prefetch requests in multi-core systems hurts the performance due to interference with demand requests in shared resources; nevertheless, some studies have been done on managing the interference caused by prefetching [SPS17]. By applying CLAP in a compressed multi-core system, we can enable the prefetching only when the blocks are likely to be compressible and co-allocatable in one data entry; otherwise, we can disable the prefetcher in order to avoid the interference.

Publications of the thesis

Niloofar Charmchi and Caroline Collange, « Toward compression-aware prefetching », in: COMPAS 2019 - Conférence d'informatique en Parallélisme, Architecture et Système, 2019, pp. 1–9.

Niloofar Charmchi, Caroline Collange, and André Seznec, « Compressed cache layout aware prefetching », in: 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), IEEE, 2019, pp. 25–28.

BIBLIOGRAPHY

[] *gem5 simulator*, `http://www.gem5.org`, Accessed: 2019-11-20.

[AA18] Alaa R Alameldeen and Rajat Agarwal, « Opportunistic compression for direct-mapped DRAM caches », *in*: *Proceedings of the International Symposium on Memory Systems*, 2018, pp. 129–136.

[ADS15] Angelos Arelakis, Fredrik Dahlgren, and Per Stenstrom, « Hycomp: A hybrid cache compression method for selection of data-type-specific compression methods », *in*: *Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 38–49.

[Ahm+15] Masab Ahmad et al., « Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores », *in*: *2015 IEEE International Symposium on Workload Characterization*, IEEE, 2015, pp. 44–55.

[Ala+03] Alaa R Alameldeen et al., « Simulating a $2MCommercialServerona2$ K PC », *in*: *Computer* 36.*2* (2003), pp. 50–57.

[APD01] Murali Annavaram, Jignesh M Patel, and Edward S Davidson, « Data prefetching by dependence graph precomputation », *in*: *Proceedings 28th Annual International Symposium on Computer Architecture*, IEEE, 2001, pp. 52–61.

[AS12] Angelos Arelakis and Per Stenström, « A case for a value-aware cache », *in*: *IEEE Computer Architecture Letters* 13.*1* (2012), pp. 1–4.

[AS14] Angelos Arelakis and Per Stenstrom, « SC 2: A statistical compression cache scheme », *in*: *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, IEEE, 2014, pp. 145–156.

[AS16] Ayaz Akram and Lina Sawalha, « A comparison of x86 computer architecture simulators », *in*: (2016).

[AW04a] Alaa R Alameldeen and David A Wood, « Adaptive cache compression for high-performance processors », *in*: *ACM SIGARCH Computer Architecture News* 32.*2* (2004), p. 212.

[AW04b] Alaa Alameldeen and David Wood, *Frequent pattern compression: A significance-based compression scheme for L2 caches*, tech. rep., University of Wisconsin-Madison Department of Computer Sciences, 2004.

[BC07] Jean Christophe Beyler and Philippe Clauss, « Performance driven data cache prefetching in a dynamic software optimization system », *in*: *Proceedings of the 21st annual international conference on Supercomputing*, 2007, pp. 202–209.

[BC91] Jean-Loup Baer and Tien-Fu Chen, « An effective on-chip preloading scheme to reduce data access penalty », *in*: *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, 1991, pp. 176–186.

[Ben00] SPEC Benchmarks, *Standard performance evaluation corporation*, `http://www.spec.org/`, 2000.

[Ben06] SPEC Benchmarks, *Standard performance evaluation corporation*, `http://www.spec.org/`, 2006.

[Bin+06] Nathan L Binkert et al., « The M5 simulator: Modeling networked systems », *in*: *Ieee micro* 26.*4* (2006), pp. 52–60.

[Bin+11] Nathan Binkert et al., « The gem5 simulator », *in*: *ACM SIGARCH Computer Architecture News* 39.*2* (2011), pp. 1–7.

[Bis+06] Surupa Biswas et al., « Memory overflow protection for embedded systems using run-time checks, reuse, and compression », *in*: *ACM Transactions on Embedded Computing Systems (TECS)* 5.*4* (2006), pp. 719–752.

[Blo70] Burton H Bloom, « Space/time trade-offs in hash coding with allowable errors », *in*: *Communications of the ACM* 13.*7* (1970), pp. 422–426.

[CC19] Niloofar Charmchi and Caroline Collange, « Toward compression-aware prefetching », *in*: *COMPAS 2019 - Conférence d'informatique en Parallélisme, Architecture et Système*, 2019, pp. 1–9.

[CCS19] Niloofar Charmchi, Caroline Collange, and André Seznec, « Compressed cache layout aware prefetching », *in*: *31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, IEEE, 2019, pp. 25–28.

[Che+09] Xi Chen et al., « C-pack: A high-performance microprocessor cache compression algorithm », *in*: *IEEE transactions on very large scale integration (VLSI) systems* 18.*8* (2009), pp. 1196–1208.

[CW79] J Lawrence Carter and Mark N Wegman, « Universal classes of hash functions », *in*: *Journal of computer and system sciences* 18.*2* (1979), pp. 143–154.

[DDS95] Fredrik Dahlgren, Michel Dubois, and Per Stenstrom, « Sequential hardware prefetching in shared-memory multiprocessors », *in*: *IEEE Transactions on Parallel and Distributed Systems* 6.*7* (1995), pp. 733–746.

[Dou93] Fred Douglis, « The Compression Cache: Using On-line Compression to Extend Physical Memory. », *in*: *Usenix Winter*, Citeseer, 1993, pp. 519–529.

[DPS09] Julien Dusser, Thomas Piquet, and André Seznec, « Zero-content augmented caches », *in*: *Proceedings of the 23rd international conference on Supercomputing*, 2009, pp. 46–55.

[DS11] Julien Dusser and André Seznec, « Decoupled zero-compressed memory », *in*: *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, 2011, pp. 77–86.

[ES05] Magnus Ekman and Per Stenstrom, « A robust main-memory compression scheme », *in*: *32nd International Symposium on Computer Architecture (ISCA'05)*, IEEE, 2005, pp. 74–85.

[Fer+08] Michael Ferdman et al., « Temporal instruction fetch streaming », *in*: *2008 41st IEEE/ACM International Symposium on Microarchitecture*, IEEE, 2008, pp. 1–10.

[FF07] Michael Ferdman and Babak Falsafi, « Last-touch correlated data streaming », *in*: *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, IEEE, 2007, pp. 105–115.

[FKF11] Michael Ferdman, Cansu Kaynak, and Babak Falsafi, « Proactive instruction fetch », *in*: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 152–162.

[FPJ92] John WC Fu, Janak H Patel, and Bob L Janssens, « Stride directed prefetching in scalar processors », *in*: *ACM SIGMICRO Newsletter* 23.*1-2* (1992), pp. 102–110.

[FW14] Babak Falsafi and Thomas F Wenisch, « A primer on hardware prefetching », *in*: *Synthesis Lectures on Computer Architecture* 9.*1* (2014), pp. 1–67.

[GAS16] Jayesh Gaur, Alaa R Alameldeen, and Sreenivas Subramoney, « Base-victim compression: An opportunistic cache compression architecture », *in*: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2016, pp. 317–328.

[Ham+14] Per Hammarlund et al., « Haswell: The fourth-generation intel core processor », *in*: *IEEE Micro* 34.*2* (2014), pp. 6–20.

[Hon+19] Seokin Hong et al., « Touché: Towards Ideal and Efficient Cache Compression By Mitigating Tag Area Overheads », *in*: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 453–465.

[HP11] John L Hennessy and David A Patterson, *Computer architecture: a quantitative approach*, Elsevier, 2011.

[HR05] Erik G Hallnor and Steven K Reinhardt, « A unified compressed memory hierarchy », *in*: *11th International Symposium on High-Performance Computer Architecture*, IEEE, 2005, pp. 201–212.

[ILS05] Martin Isenburg, Peter Lindstrom, and Jack Snoeyink, « Lossless compression of predicted floating-point geometry », *in*: *Computer-Aided Design* 37.*8* (2005), pp. 869–877.

[IS09] Mafijul Md Islam and Per Stenstrom, « Zero-value caches: Cancelling loads that return zero », *in*: *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, IEEE, 2009, pp. 237–245.

[Jou90] Norman P Jouppi, « Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers », *in*: *ACM SIGARCH Computer Architecture News*, vol. 18, 2SI, ACM, 1990, pp. 364–373.

[Kim+11] Soontae Kim et al., « Residue cache: a low-energy low-area L2 cache architecture via compression and partial hits », *in*: *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2011, pp. 420–429.

[KSH14] Muneeb Khan, Andreas Sandberg, and Erik Hagersten, « A case for resource efficient prefetching in multicores », *in*: *2014 43rd International Conference on Parallel Processing*, IEEE, 2014, pp. 101–110.

[LFF01] An-Chow Lai, Cem Fide, and Babak Falsafi, « Dead-block prediction & dead-block correlating prefetchers », *in*: *Proceedings 28th Annual International Symposium on Computer Architecture*, IEEE, 2001, pp. 144–154.

[LHK99] Jang-Soo Lee, Won-Kee Hong, and Shin-Dug Kim, « Design and evaluation of a selective compressed memory system », *in*: *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors (Cat. No. 99CB37040)*, IEEE, 1999, pp. 184–191.

[Lip68] John S. Liptay, « Structural aspects of the System/360 Model 85, II: The cache », *in*: *IBM Systems Journal* 7.*1* (1968), pp. 15–21.

[LPM97] Chunho Lee, Miodrag Potkonjak, and William H Mangione-Smith, « Mediabench: A tool for evaluating and synthesizing multimedia and communications systems », *in*: *Proceedings of 30th Annual International Symposium on Microarchitecture*, IEEE, 1997, pp. 330–335.

[LWS96] Mikko H Lipasti, Christopher B Wilkerson, and John Paul Shen, « Value locality and load value prediction », *in*: *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, 1996, pp. 138–147.

[Mag+02] Peter S Magnusson et al., « Simics: A full system simulation platform », *in*: *Computer* 35.*2* (2002), pp. 50–58.

[Mar+05] Milo MK Martin et al., « Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset », *in*: *ACM SIGARCH Computer Architecture News* 33.*4* (2005), pp. 92–99.

[Mic16] Pierre Michaud, « Best-offset hardware prefetching », *in*: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2016, pp. 469–480.

[Mit16] Sparsh Mittal, « A survey of recent prefetching techniques for processor caches », *in*: *ACM Computing Surveys (CSUR)* 49.*2* (2016), p. 35.

[MV15] Sparsh Mittal and Jeffrey S Vetter, « A survey of architectural approaches for data compression in cache and main memory systems », *in*: *IEEE Transactions on Parallel and Distributed Systems* 27.*5* (2015), pp. 1524–1536.

[MV99] Nihar R Mahapatra and Balakrishna Venkatrao, « The processor-memory bottleneck: problems and solutions », *in*: *XRDS: Crossroads, The ACM Magazine for Students* 5.*3es* (1999), p. 2.

[Pat+97] David Patterson et al., « A case for intelligent RAM », *in*: *IEEE micro* 17.*2* (1997), pp. 34–44.

[Pei+02] Jih-Kwon Peir et al., « Bloom filtering cache misses for accurate data speculation and prefetching », *in*: *ACM International Conference on Supercomputing 25th Anniversary Volume*, 2002, pp. 347–356.

[Pek+12] Gennady Pekhimenko et al., « Base-delta-immediate compression: practical data compression for on-chip caches », *in*: *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, ACM, 2012, pp. 377–388.

[Pek+13] Gennady Pekhimenko et al., « Linearly compressed pages: a low-complexity, low-latency main memory compression framework », *in*: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 172–184.

[PHM15] Bhargavraj Patel, Nikos Hardavellas, and Gokhan Memik, « SCP: Synergistic cache compression and prefetching », *in*: *2015 33rd IEEE International Conference on Computer Design (ICCD)*, IEEE, 2015, pp. 164–171.

[PS16] Biswabandan Panda and André Seznec, « Dictionary sharing: An efficient cache compression scheme for compressed caches », *in*: *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, IEEE, 2016, pp. 1–12.

[Qur+07] Moinuddin K Qureshi et al., « Adaptive insertion policies for high performance caching », *in*: *ACM SIGARCH Computer Architecture News* 35.*2* (2007), pp. 381–391.

[RCA99] Glenn Reinman, Brad Calder, and Todd Austin, « Fetch directed instruction prefetching », *in*: *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, IEEE, 1999, pp. 16–27.

[RKB06] Paruj Ratanaworabhan, Jian Ke, and Martin Burtscher, « Fast lossless compression of scientific floating-point data », *in*: *Data Compression Conference (DCC'06)*, IEEE, 2006, pp. 133–142.

[RKP00] Sumit Roy, Raj Kumar, and Milos Prvulovic, « Improving system performance with compressed memory », *in*: *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, IEEE, 2000, 7– pp.

[Rog+09] Brian M Rogers et al., « Scaling the bandwidth wall: challenges in and avenues for CMP scaling », *in*: *Proceedings of the 36th annual international symposium on Computer architecture*, 2009, pp. 371–382.

[RPM16] Kanakagiri Raghavendra, Biswabandan Panda, and Madhu Mutyam, « PBC: Prefetched blocks compaction », *in*: *IEEE Transactions on Computers* 65.*8* (2016), pp. 2534–2547.

[Sar+15] Somayeh Sardashti et al., « A primer on compression in the memory hierarchy », *in*: *Synthesis Lectures on Computer Architecture* 10.*5* (2015), pp. 1–86.

[SB93] André Seznec and Francois Bodin, « Skewed-associative caches », *in*: *International Conference on Parallel Architectures and Languages Europe*, Springer, 1993, pp. 305–316.

[Sha+14] Ali Shafiee et al., « MemZip: Exploring unconventional benefits from memory compression », *in*: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2014, pp. 638–649.

[SK10] Daniel Sanchez and Christos Kozyrakis, « The ZCache: Decoupling ways and associativity », *in*: *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE, 2010, pp. 187–198.

[SK13] Daniel Sanchez and Christos Kozyrakis, « ZSim: Fast and accurate microarchitectural simulation of thousand-core systems », *in*: *ACM SIGARCH Computer architecture news* 41.*3* (2013), pp. 475–486.

[Smi78] Alan Jay Smith, « Sequential program prefetching in memory hierarchies », *in*: *Computer 12* (1978), pp. 7–21.

[Smi82] Alan Jay Smith, « Cache memories », *in*: *ACM Computing Surveys (CSUR)* 14.*3* (1982), pp. 473–530.

[Som+09] Stephen Somogyi et al., « Spatio-temporal memory streaming », *in*: *ACM SIGARCH Computer Architecture News*, vol. 37, 3, ACM, 2009, pp. 69–80.

[SP11] Avinash Sodani and C Processor, « Race to exascale: Opportunities and challenges », *in*: *Keynote at the Annual IEEE/ACM 44th Annual International Symposium on Microarchitecture*, 2011.

[Spr07] Cloyce D Spradling, « SPEC CPU2006 benchmark tools », *in*: *ACM SIGARCH Computer Architecture News* 35.*1* (2007), pp. 130–134.

[SPS17] Aswinkumar Sridharan, Biswabandan Panda, and Andre Seznec, « Band-pass prefetching: an effective prefetch management mechanism using prefetch-fraction metric in multi-core systems », *in*: *ACM Transactions on Architecture and Code Optimization (TACO)* 14.*2* (2017), pp. 1–27.

[Sri+07] Santhosh Srinath et al., « Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers », *in*: *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, IEEE, 2007, pp. 63–74.

[SSC00] Timothy Sherwood, Suleyman Sair, and Brad Calder, « Predictor-directed stream buffers », *in*: *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, 2000, pp. 42–53.

[SSW14] Somayeh Sardashti, André Seznec, and David A Wood, « Skewed compressed caches », *in*: *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 2014, pp. 331–342.

[SSW16] Somayeh Sardashti, André Seznec, and David A Wood, « Yet another compressed cache: A low-cost yet effective compressed cache », *in*: *ACM Transactions on Architecture and Code Optimization (TACO)* 13.*3* (2016), p. 27.

[SW13] Somayeh Sardashti and David A Wood, « Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching », *in*: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM, 2013, pp. 62–73.

[Tre+01] R Brett Tremaine et al., « IBM memory expansion technology (MXT) », *in*: *IBM Journal of Research and Development* 45.*2* (2001), pp. 271–285.

[WM95] Wm A Wulf and Sally A McKee, « Hitting the memory wall: implications of the obvious », *in*: *ACM SIGARCH computer architecture news* 23.*1* (1995), pp. 20–24.

[Wu+09] Xiaoxia Wu et al., « Hybrid cache architecture with disparate memory technologies », *in*: *ACM SIGARCH computer architecture news* 37.*3* (2009), pp. 34–45.

[Wu+11] Carole-Jean Wu et al., « PACMan: prefetch-aware cache management for high performance caching », *in*: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 442–453.

[YG02] Jun Yang and Rajiv Gupta, « Frequent value locality and its applications », *in*: *ACM Transactions on Embedded Computing Systems (TECS)* 1.*1* (2002), pp. 79–105.

[YZG00] Jun Yang, Youtao Zhang, and Rajiv Gupta, « Frequent value compression in data caches », *in*: *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*, IEEE, 2000, pp. 258–265.

[ZCS10] Huaiyu Zhu, Yong Chen, and Xian-He Sun, « Timing local streams: improving timeliness in data prefetching », *in*: *Proceedings of the 24th ACM International Conference on Supercomputing*, 2010, pp. 169–178.

[ZYG00] Youtao Zhang, Jun Yang, and Rajiv Gupta, « Frequent value locality and value-centric data cache design », *in*: *ACM SIGPLAN Notices* 35.*11* (2000), pp. 150–159.

UNIVERSITÉ DE RENNES 1

Titre : Préchargement adapté à la structure d'un cache compressé

Mot clés : compression de cache, compactage, préchargement matériel

Résumé : Le fossé de vitesse entre le processeur et la mémoire dégrade la performance. La compression de cache et le préchargement matériel sont deux techniques qui pourraient éviter ce goulet d'étranglement en diminuant les échecs du dernier niveau de cache. Pourtant, la compression et le préchargement ont des interactions positives, car le préchargement bénéficie d'une capacité de cache plus élevée et la compression augmente la taille effective du cache.

Cette étude propose *Compressed cache Layout Aware Prefetching* (CLAP) pour exploiter les schémas de cache sectoriels compressés, comme SCC ou YACC, pour créer une synergie entre les caches compressés et le préchargement. L'idée de cette approche est de précharger des blocs contigus qui peuvent être compressés et co-alloués ensemble avec le bloc demandé sur un défaut de cache. Les blocs préchargés qui partagent le stockage avec les blocs existants n'ont pas besoin d'expulser une entrée existante valide ; par conséquent, CLAP évite la pollution de cache.

Afin de choisir les blocs co-allouables à précharger, nous proposons un prédicteur de compression. Sur nos évaluations expérimentales, CLAP réduit le nombre de défauts de cache de 9% et améliore les performances de 3% en moyenne, par rapport à un cache compressé. De plus, afin d'obtenir plus d'améliorations, nous unifions CLAP et d'autres politiques de préchargement et introduisons deux CLAP adaptatifs qui sélectionnent le meilleur politique de préchargement selon l'application.

Title: Compressed Cache Layout Aware Prefetching

Keywords: cache compression, compaction, hardware prefetching

Abstract: The speed gap between CPU and memory is impairing performance. Cache compression and hardware prefetching are two techniques that could confront this bottleneck by decreasing last level cache misses. However, compression and prefetching have positive interactions, as prefetching benefits from higher cache capacity and compression increases the effective cache size.

This study proposes *Compressed cache Layout Aware Prefetching* (CLAP) to leverage the recently proposed sector-based compressed cache layouts, such as SCC or YACC, to create a synergy between compressed caches and prefetching. The idea of this approach is to prefetch contiguous blocks that can be compressed and co-allocated together with the requested block on a miss access. Prefetched blocks that share storage with existing blocks do not need to evict a valid existing entry; therefore, CLAP avoids cache pollution.

In order to decide the co-allocatable blocks to prefetch, we propose a compression predictor. Based on our experimental evaluations, CLAP reduces the number of cache misses by 9% and improves performance by 3% on average, comparing to a compressed cache. Furthermore, in order to get more improvements, we unify CLAP and other prefetchers and introduce two adaptive CLAPs which select the best prefetcher based on the application.