Towards Compression At All Levels In The Memory Hierarchy

Daniel Rodrigues Carvalho

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

Daniel RODRIGUES CARVALHO

Towards Compression At All Levels In The Memory Hierarchy

Thèse présentée et soutenue à Visioconférence, le 9 Avril 2021
Unité de recherche : Inria Rennes - Équipe PACAP
Thèse N° :

Rapporteurs avant soutenance :

Daniel ETIEMBLE	Professeur, Université Paris Sud
Bernard GOOSSENS	Professeur, Université Perpignan

Composition du Jury :

Président :	Prénom Nom	*(à préciser après la soutenance)*
Examinateurs :	Arthur PERAIS	Chargé de Recherche, TIMA Grenoble
	Caroline COLLANGE	Chargée de Recherche, Inria Rennes
	Olivier SENTIEYS	Professeur, Université de Rennes 1
Dir. de thèse :	André SEZNEC	Directeur de Recherches, Inria Rennes

Table of Contents

Acronyms

BAI Bandwidth-Aware Indexing. 41

BCR best compression ratio. 97

BDI Base-Delta-Immediate. 16, 44, 67, 71, 72, 74, 86, 89, 92, 95, 99, 100, 102, 105

BFPC Burtscher's FPC. 57, 58

BLEM Blended Metadata Engine. 54

BP block pair. 18, 19, 78–81, 83

BPC Bit-Plane Compression. 71, 74, 99, 105

C-Pack Cache Packer. 16, 66, 67, 72–74, 86, 96, 99, 101

CABLE CAche-Based Link Encoder. 57, 58

CAM Content-Addressable Memory. 70

CAMP Compression-Aware Management Policy. 42

CC Compression Cache. 45

CF compression factor. 78–80

CID Compression ID. 54

COCO Cross-Object COmpression. 74

COPR Compression Predictor Unit. 54

CPU Central Processing Unit. 21, 58

DBI Data Bus Inversion. 58

DBP Delta-BitPlane. 71

DCC Decoupled Compressed Cache. 37, 45

DFPC Dynamic Frequent Pattern Compression. 74

DICE Dynamic-Indexing Cache Compression. 38, 45

DISH Dictionary Sharing. 72, 74, 94

DRAM Dynamic RAM. 4, 21, 22, 38, 47, 48, 50, 56, 81

DSF Decision Switching Filter. 40

DVSC Decoupled Variable-Segment Cache. 34, 45

DZC Decoupled Zero-Compressed. 51, 56, 59

ECC Error-Correcting Code. 58

ECM Effective Cache Maximizer. 42

FCMS Fine-grained Compressed Memory System. 45

FPC Frequent Pattern Compression. 66, 67, 74, 99

FPC-D Frequent Pattern Compression with limited Dictionary support. 16, 67, 68, 73, 74, 86, 96, 97, 99, 105

FVC Frequent Value Cache. 38

GB Gigabyte. 48

GPU Graphics Processing Unit. 48, 58

HoPE Hot-cacheline Prediction and Early decompression. 42

HyComp Hybrid Compression. 72

I/O Input/Output. 21

IIC Indirect-Index Cache. 34

IIC-C Indirect Index Cache with Compression. 40, 45, 59

IPC Instructions Per Cycle. 22, 83, 97, 99

KB Kilobyte. 50, 52, 60

LBE Large-Block Encoding. 68, 70, 74

LCP Linearly Compressed Pages. 51, 52, 56

LLC Last-Level Cache. 22, 39, 59

LLP Line Location Predictor. 55

LRU Least-Recently Used. 42, 81

LSB Least-Significant Bits. 17, 19, 31, 38, 55, 57, 78, 85, 87, 88, 91, 93, 105

LUT lookup table. 56, 57

LZ Lempel-Ziv. 74

MB Megabyte. 22, 23, 48, 52

MBZip-C MBZip at the Cache. 52

MBZip-M MBZip at the Memory. 52, 53

MORC Manycore-Oriented Compressed Cache. 38, 45

MPKI Misses Per Kilo-Instruction. 22, 80

MRU Most-Recently Used. 42

MSB Most-Significant Bits. 17, 19, 35, 38, 57, 66, 78, 85, 87, 88, 91, 93, 94, 105

MSHR Miss-Status Holding Register. 81

MXT Memory Expansion Technology. 50, 56, 59

NVM Non-Volatile Memory. 48

O2W Opportunistic 2-Way. 45

OOO out-of-order. 81

OS Operating System. 4, 26, 47, 49, 52–54, 56

PBC Phased-In Binary Codes. 64

PBC Prefetched Blocks Compaction. 43

PCM Phase-Change Memories. 48

PSS Pairwise Space Sharing. 76–83, 104, 105, 132

PTMC Practical and Transparent Memory Compression. 54, 56

RAM Random-Access Memory. 21

RRIP Re-Reference Interval Prediction. 42, 81

SB superblock. 34, 35, 39, 42, 45, 52, 54, 55, 78, 79, 82

SC^2 Statistical Compressed Cache. 45

SCC Skewed Compressed Cache. 32, 39, 42, 45, 80

SCMS Selective Compressed Memory System. 32, 45, 50, 56

SPEC Standard Performance Evaluation Corporation. 18, 65, 76, 80, 86, 92

SRAM Static RAM. 22

SRC Synergistic cache layout for Reuse and Compression. 42

STT Sector Translation Table. 50

SWC Significance-Width Compression. 55, 58

TAD Tag and Data. 38

TB Terabyte. 22

TLB Translation Look-aside Buffer. 47, 51, 52, 81

X-RL X-Match and Run Length. 74, 99

XID Exclusive ID. 54

YACC Yet Another Compressed Cache. 32, 39, 42, 45, 78–80, 82

ZC Zero Content. 51, 59

Résumé en Français

0.1 La hiérarchie de la mémoire

Lors de l'exécution des charges de travail, un ordinateur doit stocker ses calculs en cours dans un stockage appelé mémoire. Malheureusement, l'accès à la mémoire prend un temps excessivement élevé. En tant que tel, l'ajout d'une hiérarchie de mémoires progressivement plus petites avec des latences et un coût énergétique proportionnelles [NW15] — également connu sous le nom de *caches* — est devenu la norme pour les systèmes actuels [CKD+10, HP12, LH11].

Bien qu'ils représentent une fraction du contenu de la mémoire, les caches deviennent réalisables en raison de la localité des références mémoire: il a été observé que les emplacements mémoire ont une forte probabilité d'être accédés à plusieurs reprises, et que les emplacements adjacents ont tendance à avoir une corrélation d'accès élevée; par conséquent, dans un laps de temps spécifique, seul un petit sous-ensemble de l'espace d'adressage est utilisé par un processus, et les caches en profitent pour fournir un accès plus rapide dans cette situation courante.

Des tailles de cache plus grands permettent d'améliorer la performance du système: dans notre configuration simulée, augmenter naïvement la taille du cache de dernier niveau (LLC) de 1 Mo à 2/4 Mo générerait une accélération de 5,0/13,4%, et une réduction du nombre d'échecs de la LLC de 12,2/32,0%; cependant, cette amélioration n'est pas gratuite et implique de plus grandes latences d'accès en raison des circuits plus complexes, ce qui réduit ces améliorations.

De plus, l'augmentation de la taille du cache entraîne des coûts de surface et d'énergie plus élevés [ZIM+07]: les caches représentent une grande quantité de dissipation d'énergie dans une puce [GH96, VKI+00, MKG98, Yel11]; ainsi, même s'il est souhaitable d'avoir de plus grandes tailles de cache pour obtenir des ratios d'erreurs plus bas [HSPE08], avoir de plus grandes capacités entraîne une consommation d'énergie plus élevée. Néanmoins, les caches impliquent également des économies d'énergie, car ne pas les avoir impliquerait plus d'accès à la mémoire hors puce, qui ont une consommation d'énergie mille fois plus élevée [NW15]; par conséquent, les architectes informatiques visent à développer des conceptions de cache plus grandes, mais économes en énergie.

0.2 Compression de la mémoire

Au fil des années, plusieurs travaux ont proposé des idées pour améliorer la mémoire. Cela comprend l'augmentation de la capacité [QSP07, AW04a], la réduction du taux de défauts de

cache [AP93, PHM15], la réduction des exigences d'aire des étiquettes [Sez94, WSY95], l'ajout de la vérification d'erreurs [CH84, CCZ13], entre autres. Ce travail se concentre sur les techniques qui améliorent la mémoire grâce à la compression.

La compression de mémoire peut avoir multiples objectifs, mais l'approche générale comprend la compression et le compactage de données de sorte que les avantages de mémoires plus grandes soient obtenus sans les principaux inconvénients d'une augmentation physique de leurs tailles. Cependant, cette mise à niveau n'est pas gratuite et des étapes de compression et de décompression doivent être ajoutées au processus d'accès, ce qui augmente légèrement l'énergie et la latence des accèss au cache réussis.

Dans les caches conventionnels, la taille d'une entrée physique dans le tableau de données (*entrée de donées*) correspond généralement à la taille de la ligne de cache. Dans les caches compressés, par contre, plusieurs lignes peuvent être assignées à une seule entrée de données [LHK00]. Cela permet effectivement d'augmenter la taille du cache avec peu d'augmentation de surface: au lieu d'augmenter le nombre de entrées de données, on ajoute un compresseur, un décompresseur, et quelques circuits pour gérer l'organisation des blocs compressés.

Par conséquent, la complexité du compresseur devient un compromis crucial — les algorithmes de compression génèrent généralement du matériel plus gros et plus complexe à mesure que de meilleurs facteurs de compression sont obtenus [PSM+12]. De plus, à mesure que plus de données s'insèrent dans la mémoire, plus d'étiquettes sont nécessaires pour référencer ces données [AS14b]. Si le nombre d'entrées étiquettes est bas, le taux de compression devient limité par ce nombre; mais s'il est trop élevée, de la surface et de l'énergie sont gaspillées. Un autre problème rencontré est que, puisque la latence de décompression affecte de manière critique la latence de la mémoire, elle doit rester minimale [AA18].

Pour cette raison, bien que, en théorie, la compression du cache puisse être appliquée à tous les niveaux, il peut être difficile de compresser les caches les plus proches du cœur; puisqu'ils sont les plus fréquemment utilisés, l'ajout d'une latence de décompression à ce chemin critique pourrait dégrader considérablement les performances. De plus, trouver le mot critique n'est pas trivial lorsque les données sont compressées; ainsi, la décompression doit avoir lieu avant d'être envoyée. Heureusement, il existe certaines techniques qui peuvent être appliquées pour réduire le temps moyen d'attente pour que les données soient décompressées [LHK99, LHK00, AW04a].

Chaque fois qu'une entrée compressée est écrasée, il peut y avoir un débordement de données, c'est-à-dire que la nouvelle taille compressée peut devenir plus grande que la précédente. Cela s'appelle *l'expansion de données*, et cela nécessite un traitement spécial [LHK00]. S'il reste suffisamment d'espace dans la tranche de données, les données sont mises à jour, aussi bien que les informations de métadonnées existantes (telles que la taille compressée); cependant, si l'entrée n'a pas assez d'espace, les données co-allouées doivent être déplacées ou expulsées pour générer de la place pour l'entrée redimensionnée.

Les étapes de déplacement et d'expulsion peuvent être effectuées en dehors du chemin critique, mais elles impliquent une consommation d'énergie supplémentaire, car plusieurs expulsions/mouvements peuvent être nécessaires pour faire suffisamment de place. De plus, les métadonnées des entrées co-allouées doivent être lues et analysées pour calculer les opérations qui doivent être realisées. L'utilisation de segments de taille fixe peut atténuer le problème d'expansion de données, car l'espace alloué est toujours arrondi, donc il peut donc y avoir de l'espace pour des petites extensions.

Lorsqu'elle est appliquée à la mémoire principale, la compression élargit encore plus l'ensemble des défis: alors que dans les caches, les lignes sont localisées à l'aide des étiquettes, le volume de ces étiquettes devient insupportable pour les mémoires plus grandes; ainsi, en général, la mémoire principale ne contiennent pas ces structures. Cela complique la localisation des lignes compressées. De plus, le Système d'Exploitation (OS) s'appuie sur la taille de la mémoire qui, avec la compression, n'est plus statique. Comme la quantité de mémoire visible par l'OS est plus grande que la quantité physique, il ne peut pas savoir quand la mémoire physique est épuisée; donc, bien qu'il puisse penser qu'il reste de la mémoire à utiliser, il se peut qu'il soit en fait à court [AF00].

Enfin, on peut également concevoir un système où toute la hiérarchie mémoire est compressée. Dans un tel système, ayant plusieurs méthodes de compression (*e.g.*, une pour la hiérarchie du cache, et une autre pour la mémoire principale), bien que ce soit une décision attrayante — on pourrait choisir des schémas plus rapides pour les caches plus proches du cœur et des schémas efficaces pour des mémoires plus éloignés — elle pourrait ne pas être optimale. C'est le cas parce que les transitions entre les niveaux de mémoire nécessiteraient décompressions et compressions constantes; pourtant, si tous les niveaux utilisaient le même compresseur, les données pourraient être envoyées compressées pour économiser la bande passante. Par conséquent, décider d'utiliser ou non un schéma de compression unique pour tous les niveaux est d'une grande importance.

Néanmoins, une hiérarchie de mémoire entièrement compressée peut considérablement améliorer les performances, et la consommation d'énergie et de bande passante par rapport aux hiérarchies de mémoire conventionnelles [HR05, LHK00]. Avoir une capacité effective plus élevée implique moins d'accès à des niveaux de mémoire plus élevés, et moins de défauts de cache et page; ainsi, la performance globale du système est améliorée. En outre, bien qu'il y ait plus de matériel et généralement plus de consommation d'énergie statique, la puissance globale peut être réduite en raison du nombre plus bas de défauts des mémoires plus éloignées du cœur — la bande passante globale est réduite, car moins de demandes d'accès à un niveau de mémoire moins élevé sont générées [PSK+13].

De plus, le temps de transferts peut être réduit en raison du fait que des données compressées sont envoyées — *i.e.*, plus de données peuvent être envoyées dans la même intervalle de temps [CR95, STBD14]. Enfin, les avantages de la compression de l'ensemble du système sont propor-

tionnels aux latences de la mémoire; par conséquent, les systèmes avec des latences élevées voient leurs performances améliorées en raison de la réduction des accès à des niveaux de mémoire plus élevés.

0.3 Compression Region-Chunk

Les algorithmes de compression matérielle sont généralement des dérivations simplifiées d'algorithmes de compression de données. C'est le cas pour deux raisons: premièrement, la complexité matérielle doit rester limitée; deuxièmement, la latence supplémentaire inhérente, ajoutée en raison de l'étape de décompression, ne doit pas interférer gravement avec la latence de succès [SASW15, PSM+12].

La compression du cache a tendance à s'appuyer fortement sur les localités spatiales et temporelles des données; en substance, on s'attend à ce que les valeurs vues précédemment soient parfaitement ou partiellement répétées. Par conséquent, il existe une prédominance de compresseurs basés sur des dictionnaires — compresseurs qui utilisent les premières valeurs d'une ligne comme références pour les valeurs suivantes, appliquant des modèles pour comparer et faire correspondre les valeurs, généralement au niveau de l'octet. Ces références sont ensuite utilisées pour supprimer les bits répétés dans les valeurs suivantes (déduplication de valeur) [KGJ96, CYD+10, AA18].

La déduplication suppose généralement qu'un seul type de données de base soit utilisé à plusieurs reprises, ce qui n'est pas vrai pour toutes les charges de travail. Pour faire face à cela, les compresseurs peuvent ajouter des modèles qui seraient visibles si l'hypothèse sous-jacente était d'un type de données plus petit. Par exemple, le modèle qui correspond à tous les octets à l'exception du moins significatif d'une valeur de 32 bits est représenté par MMMX — M est une correspondance d'octet, et X est une incompatibilité d'octets, dans une notation semblable à des travaux antérieurs [CYD+10, AA18].

Pour capturer un comportement similaire pour les types de données 16 bits, tout en supposant toujours que les charges de travail contiennent des types de données de 32 bits, le modèle MXMX peut être ajouté. Cela signifie que pour pouvoir compresser tous les types de données de base, les compresseurs devraient fournir des modèles pour couvrir toutes les permutations possibles d'octets correspondants/non correspondants, ce qui est coûteux. Avec l'utilisation croissante des valeurs de 64 bits, le fait de comprendre toutes les permutations devient encore plus prohibitif.

De plus, avoir plus de modèles améliore l'efficacité de la compression, mais complique le matériel de décompression, en augmentant sa surcharge de latence. Par exemple, BDI [PSM+12] peut réaliser une décompression en 1 cycle en ne couvrant que deux modèles. Son taux de compression moyen (rapport entre les tailles compressées et non compressées) est, cependant, élevé. Des propositions telles que C-Pack [CYD+10] et FPC-D [AA18] ajoutent plus de modèles

et atteignent des ratios inférieurs; cependant, leur décompression peut être aussi lente qu'un mot par cycle.

En général, il a été observé que le contenu de la partie de poids plus fort (MSB) d'une valeur présente moins de variabilité que son homologue moins fort (LSB); et donc les compresseurs ont tendance à mieux compresser ce premier [CYD+10, PSM+12, PS16]. Par conséquent, il pourrait être avantageux de diviser davantage les morceaux en différentes parties, qui sont compressées différemment. L'intuition est que la probabilité de voir des valeurs égales est proportionnelle à la taille du bloc (*chunk*) ($\frac{1}{tailleChunk}$), donc la probabilité de se référer aux entrées précédentes du dictionnaire est plus élevée, ce qui augmente l'efficacité de la compression. De plus, étant donné que chaque partie est toujours censée représenter les mêmes bits respectifs du chunk, l'hypothèse globale du type de données représentatif de la charge de travail est conservée.

Dans ce manuscrit **on présente la compression Region-Chunk (RC), une nouvelle perspective sur le problème de correspondance**. La compression Region-Chunk est un *concept* qui permet de mieux isoler ce qui est mis en correspondance et d'explorer la granularité des types de données des charges de travail. *Au lieu de compresser les lignes de cache avec une granularité de valeur, RC divise chaque valeur en sous-valeurs plus petites — mais de même taille – dont le contenu est plus susceptible d'être similaire, puis compresse chaque sous-valeur différemment.* Cela permet d'améliorer la déduplication globale et d'augmenter la couverture des modèles.

Le principal avantage de l'ajout de sous-divisions est que la taille du morceau correspond toujours au type de données prédominant attendu de la charge de travail globale, mais les occurrences de types de données égaux ou plus petits sont compressées plus efficacement — la granularité plus fine avec laquelle les dictionnaires sont construits réduit la duplication. De plus, le nombre de modèles couverts est implicitement augmenté, car la combinaison des modèles des compresseurs de chaque région génère un spectre plus large. Par exemple, pour les chunks de 64 bits, le modèle MMMMMXMM ne fait généralement pas partie des modèles sélectionnés; cependant, il serait assuré comme une combinaison possible des quatre régions dans un compresseur $R_{16}C_{64}$ contenant les modèles MM et MX (MM + MM + MX + MM).

Un autre avantage du raffinage de la granularité est que le compresseur de chaque région peut être modifié pour couvrir ses besoins. Par exemple, en s'attendant à ce que les régions MSB aient moins de variabilité que les régions LSB, il serait raisonnable de réduire le nombre maximum d'entrées de dictionnaire différentes autorisées ou le nombre de modèles couverts, ce qui réduirait à son tour le nombre de bits de métadonnées nécessaires. Une autre façon de personnaliser ces compresseurs serait de modifier les modèles eux-mêmes — *e.g.*, en augmentant ou en diminuant le nombre de bits non correspondants, on pourrait augmenter la probabilité de déduplication des entrées, ou réduire la taille des données compressées, respectivement.

Bien que ces avantages puissent être directement exploités par les compresseurs à modèles

en général, l'augmentation de la couverture de modèles est grandement favorable pour élever l'efficacité des compresseurs base-delta [PSM+12] au niveau de l'état de l'art; par conséquent, **on introduit plusieurs nouveaux compresseurs qui étendent et améliorent les compresseurs base-delta pour atteindre des bons taux de compression *et* une latence de décompression corte**. Ces nouveaux compresseurs couvrent un large éventail de types de données, décompressent rapidement et améliorent l'efficacité de la déduplication des données, tout en offrant une complexité de compression atteignable.

0.4 Pairwise Space Sharing

Une fois qu'une ligne est compressée, un schéma de compactage décide où la placer et si elle peut co-allouer avec d'autres lignes. Certaines techniques de compactage limitent la compression à des tailles fixes (*e.g.*, 25% et 50% de la taille de la ligne), en ajoutant un remplissage aux lignes plus petites que ces tailles [SSW14, SSW16]. Ces méthodes *contraintes* nécessitent un faible volume de métadonnées, mais limitent les opportunités de co-allocation.

De plus, alors que les compresseurs de cache peuvent réussir dans certaines sections du calcul, il reste encore beaucoup de données qui ne parviennent pas à atteindre des tailles compressées compatibles avec le compactage; la taille moyenne compressée sur les SPEC 2017 [Cor17] pour plusieurs compresseurs à la pointe de la technologie est toujours bien supérieure à 50% de la taille non compressée, ce qui rend difficile la co-allocation efficace de blocs avec de telles limitations.

D'autres propositions suppriment ces limites, permettant aux blocs d'être compressés à n'importe quelle taille [AW04a, CYD+10] — un concept que nous appellerons méthodes *non contraintes*. Bien que ces méthodes permettent à la compression d'atteindre son plein potentiel, elles augmentent considérablement le volume des métadonnées en raison du nombre de bits nécessaires pour représenter la taille compressée. En outre, localiser les lignes dans le cache devient non trivial: elles peuvent être trouvées n'importe où dans le tableau de données. Cela entraîne l'ajout de quelques cycles supplémentaires au chemin d'accès.

Pour profiter des avantages des tailles non contraintes mais à une fraction de son coût matériel et sans pénalité de latence, **Pairwise Space Sharing** (**PSS**) a été introduit. PSS est une technique de compactage partiellement contrainte qui nécessite une volume de métadonnées minimale, tout en fournissant des résultats équivalents. *Avec PSS les blocs sont co-alloués dans des **paires de blocs (BP)**, de sorte que la somme des tailles compressées de la paire doit tenir dans l'espace de données d'un bloc non compressé.* De plus, contrairement aux approches précédentes, *PSS stocke implicitement les métadonnées et réduit la probabilité d'expansion des données.*

Le placement des blocs dans une entrée de données est simplifié avec PSS. Alors que les représentations non contraintes utilisent des pointeurs ou des tailles de sous-blocs environnants

pour localiser les sous-blocs dans les entrées de donées, PSS utilise les extrémités — MSB et LSB — des BPs comme positions de placement fixes implicites. Ces marqueurs définissent le début d'un sous-bloc — par exemple l'un des sous-blocs doit être stocké dans l'ordre inverse (le MSB devient le LSB et vice-versa). Un autre avantage du placement des extrémités est que, puisque les bits entre les sous-blocs sont inutilisés, le recompactage n'est pas nécessaire lorsque la taille d'un bloc est modifiée mais tient toujours dans le BP.

Enfin, puisque les tailles et les emplacements des deux sous-blocs dans une paire sont connus, *un seul champ de taille par BP est nécessaire*, et la taille de l'autre sous-bloc (*e.g.*, celui non inversé) est implicitement défini. Si seul le bloc non inversé est présent dans la paire, la taille stockée représente l'espace disponible pour le sous-bloc inversé. Ces modifications permettent de réduire le volume global des métadonnées, tout en améliorant la probabilité de co-allocations réussies.

Chapter 1

Introduction

Loosely speaking, a basic computer is composed of a microprocessor and a memory unit. In addition, it communicates with the outer world via I/O devices such as main storage (secondary memory), keyboards, mouses, network adapters and monitors. In its normal behavior, the computer processes input data with operations such as addition, comparison and division, among others; however, the in-flight computation results can be quite sizeable to keep track of in the slender microprocessor's register set. Besides, although these results could be temporarily stowed in the main storage along saved files, transferring data from and to it has an exceedingly high cost. Because of that a smaller memory is used as an intermediate storage to read and modify its contents (primary memory).

The next sections provide a brief overview of the memory system, and while in this study we will focus on volatile (i.e., that needs power to maintain information) Random-Access Memory (RAM), the concepts can be analogously applied to non-volatile memories.

1.1 Memories

The Dynamic RAM (DRAM) is a storage medium composed of DRAM cells disposed as a rectangular array with rows and columns that acts as a buffer between CPU and the secondary memory. Its cells usually consist of a capacitor to keep the data bit, and a transistor to regulate access to the capacitor. Therefore, by definition their charge is constantly leaking over time, so they must be refreshed periodically, and accesses discharge the capacitor, so reads must always feed the values back to the cells [Dre07]. A DRAM access consists basically of three stages: *1) activation*, where a row is opened for access; *2) restoration*, which restores the accessed cells' values; and *3) precharge*, which closes the row, preparing the memory for further accesses [CKH+16].

Missing in these memories implies in having to access the main — slower — storage. This is undesirable, so diverse techniques to improve the management of this limited space have been proposed [ZPS+04]. Nonetheless, although faster than the secondary memories', these memories' access latencies are still unduly high for swift program execution. Besides memory technology improvements, there have been multiple proposals to try to reduce this reliability. For example, *prefetching* techniques try to overcome this issue by predicting which addresses are likely to be

subsequently accessed, and thus try to fetch them in advance so that they are ready to be used when needed [Jou90, SWAF09, Mic16a]. Other methods analyse the access pattern to exploit physical capabilities [LKP+15, HPV+16], alter the scheduling management [USCM16], among other approaches.

Unfortunately, even with these improvements the access latency is prohibitive for large memory sizes. As such, adding a hierarchy of progressively smaller memories with proportional latencies and energy cost [NW15], also known as *caches*, has become standard for current systems [CKD+10, HP12, LH11].

1.2 Caches

Part of the problem of using DRAMs is that their upkeep implies in greater latencies; thus, when moving towards solving the latency issues, any extra delay must be either removed or kept to a minimum. Because of that caches use Static RAM (SRAM) cells. These cells have an increased manufacturing cost due to the use of more transistors, but manage to retain a steady state even after accesses — and thus avoid wasting execution cycles refreshing.

Caches are tables composed of data indexed by tags [Smi82]. As such, they are composed of *sets* (rows) and *ways* (columns). These tags are unique ids for the data location in the memory's address space, and are required because caches are significantly smaller than the address space, thus multiple entries map into the same set. The compound of decisions regarding the cache line size, associativity and total capacity have a direct impact on workloads' performance [Ost10].

Figure 1.1 shows an example 4MB 16-way set-associative cache in a system with a physical address space of 256TB (which uses 48-bit pointers) [1], and 64-byte cache lines and thus each line (or *block*) is stored in a 64-byte data entry. This means that block addresses are apart in a 6-bit granularity (block offset), and that a tag of 30 bits would be required to resolve its accesses (48 bit pointers minus 12 bits implicitly defined by the set a tag is stored minus 6 bits of the block offset).

Although they represent a fraction of the memory contents, caches become feasible due to the locality of memory references. It has been observed that memory locations have a high likelihood of being accessed repeatedly (*temporal locality*), and that adjacent locations tend to have a high access correlation (*spatial locality*). Therefore, within a specific time frame only a small subset of the address space is used by a process (*working set*), and caches take advantage of that to provide faster access in this prevalent situation.

Greater cache sizes allow system performance improvements. In our simulated configuration, naively increasing the Last-Level Cache (LLC) size from 1MB to 2MB/4MB generates an Instructions Per Cycle (IPC) enhancement of 5.0/13.4%, and a reduction of the LLC's Misses Per

1. 256TB address spaces are quite common nowadays; however, there are proposals to increase it [Int17], which would incur in a significant tag size increase

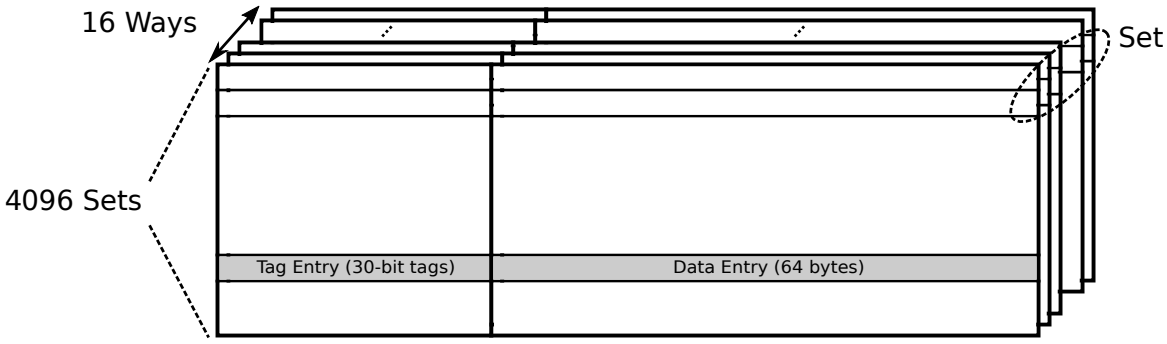


Figure 1.1 – A 4MB cache with 16 ways and cache line size of 64 bytes contains 4096 sets.

Kilo-Instruction (MPKI) of 12.2/32.0%; however, this amelioration is not free of charge, and implies in slower access latency due to the more complex circuitry, reducing these improvements. Moreover, increasing the cache size results in higher area and energy costs [ZIM+07]: caches account for a large amount of energy dissipation in a chip [GH96, VKI+00, MKG98, Yel11]; so, even though it is desirable to have greater cache sizes to achieve lower miss ratios [HSPE08], having greater capacities incurs in higher energy consumption overheads. Nonetheless, they also imply in energy savings, as not having them would entail more off-chip memory accesses, which have a thousand-fold higher power consumption [NW15]; therefore, computer architects aim on developing larger, yet energy-efficient, cache designs.

Current computer systems are designed with two or three cache levels, but there are some designs that exploit a fourth cache level [HKO+14, Mär14, DKL+17]. Throughout this text we will adopt the nomenclature that the closer to the processor, the higher is the cache level, yet the lower is the cache number. That is, L1, the closest cache to the processor, is the highest level cache.

1.2.1 Replacement Policies

As mentioned previously, both memories and caches are tables with limited storage capabilities when compared to the address space they portray, and, since they cannot provide a perfect one-to-one mapping of table entry and address, tag conflicts happen. Deciding when and which entry must be replaced on a conflict is the role of their *Replacement Policy*.

Replacement policies try to predict which data will be the most useful in the future, and which will not. Using this information they can decide which entries should be kept to minimize the number of misses [Mic16b]. Sometimes, however, misses bring entries that are not likely to be useful in the short term, and evicting another entry to make room for them would in fact degrading performance (*memory pollution*); therefore, some policies include *bypass policies*,

which allow them to bypass such allocations [JL19].

In order to yield a perfect prediction, one would have to carry access trace information throughout program execution, which is infeasible due to its required infinite storage [JL16]. Therefore, simpler policies compel less resources by focusing on tracking specific workload characteristics, such as usage recency [MGST70, JTSE10], usage frequency [CD73], and hybrids [JTSE10]. Complex policies, on the other hand, juggle the trade-off between hardware budget and precision, adding predictors based on PC [KTJ10, WJH+11], simulating optimal replacement [JL16], and even implementing simple neural networks [JT17].

1.3 Handling the Hierarchy

Until now we have assumed that the system consists of a single core with full control of its data. This happens because, in a way, all caches are private to that core, so all data changes are caused and perceived by it; however, when systems have multiple cores, a new issue arises: if there are several copies of a block at different *private caches*, changes to one of the copies must be forwarded to the others, otherwise the computations may use stale, incorrect data [SHW11]. This is known as the *Cache Coherence* problem, and many solutions have been proposed throughout the years, either through hardware or software [MHS12].

Before diving into those solutions, let's first step back and examine the effects of having multiple caches and working sets. Suppose we are dealing with a N-level cache hierarchy; If every core had its own set of N-level private cache hierarchy, the overall area of the chip would become humongous in server stations and supercomputers with their myriad of cores [TOP19], for example. *Shared caches* have the advantage of reducing the memory space and coherence control. Besides, they allow exploring the commonality of working sets to reduce data redundancy, and improving usage efficiency, since if a core does not need to use many memory resources, others can appropriate its share [TS07]. It is a common practice to make the first cache level private to keep it at a low latency, have the intermediate cache levels be shared among some cores (e.g., per pairs or tetrads of cores), and the last level cache be shared amongst all cores.

Unfortunately, making all cache levels shared by all cores would incur a gargantuan wiring mess, and in a adverse disturbance of the access latencies, so we are forced to use private caches and use controllers to cope with their drawbacks. Some coherence controllers work by having state machines that snoop on bus transactions, taking the appropriate action according to the cache line's state and protocol being used [PP84, SS86, GH09]. Coherence controllers can also rely on the use of directories (distributed, centralized or in hierarchies) to keep track of the blocks' copies without the constant need to broadcast data, as it can easily saturate system traffic [LLG+90, Ste90].

Cache coherence protocols generally consist of a subset of the following states: *Modified*,

Shared, *Invalid*, *Exclusive*, *Owned* and *Forward*. A *modified* cache line contains the only up to date copy of the data, and every other copy is stale. The *shared* state implies that several unmodified copies of a data entry are distributed across different caches. A block in the *invalid* state must be fetched from lower memory levels. An *exclusive* block is an unique identical copy of data in lower levels. The *owned* state assigns a cache line to a cache. Lastly, a cache line in a cache is attributed the *forward* state if it is one of the many unmodified copies of its data entry, but its container cache has the responsibility of responding to all requests made to that given line.

1.4 Improving Memories

Over the years several works have proposed ideas to improve memories. This includes increasing capacity [QSP07, AW04a], reducing miss ratio [AP93, PHM15], lowering tag area requirements [Sez94, WSY95], adding error checking [CH84, CCZ13], among others. In this section we briefly describe *cache and memory compression*, which is the focus of this work. More information on this topic, and how it interacts with other ideas will be presented in the next sections.

Memory compression can have multiple goals, but the general approach comprises compressing and compacting data so that the benefits of larger memories are achieved without the main drawbacks of physically increasing it. Howbeit, this upgrade is not free, and compression and decompression steps must be added to the access process, which slightly increase energy and hit latency.

On the one hand, in conventional caches the size of a physical entry in the data array (*data entry*) generally matches the size of the cache line. In compressed caches, on the other hand, multiple lines can be assigned to a single data entry [LHK00]. This allows effectively increasing the cache size with little area overhead: instead of increasing the number of data entries, one adds a compressor, a decompressor, and some circuits to handle the organization of compressed blocks. Consequently, the compressor's complexity becomes a crucial trade-off — compression algorithms usually require bigger and more complex hardware as better compression factors are achieved [PSM+12]. Furthermore, as more data fits into the memory, more tags are needed to reference this data [AS14b]. If the number of entries is low the compression ratio becomes limited by this number; but if it is too high area and energy are wasted. Finally, the decompression latency critically affects memory latency, so it should be minimal [AA18].

Because of that, although, theoretically, cache compression can be applied to all levels, it may be tricky to compress the caches closest to the core; since they are the most frequently accessed, adding a decompression latency to this critical path could greatly degrade performance. Moreover, some caches prioritize sending the critical word of requested blocks first; however,

finding it is not trivial when the data is compressed. Therefore, decompression must happen before it is sent. In spite of that, this cache level can be compressed to reduce cache miss rate if the compression scheme used is simple enough to provide immediate decompression [PA05, KAK06, TZ07]. Fortunately, there are some techniques that can be applied to reduce the average time waiting for data to decompress [LHK99, LHK00, AW04a].

Whenever a compressed entry is overwritten there may be a data overflow, that is, the new compressed size is bigger than the previous one. This is called *fat write* or *data expansion*, and it requires special handling [LHK00]. If there is sufficient space left in the data entry, the data is updated, along with the existing metadata information (such as compressed size); however, if the entry does not have enough space, the co-allocated data must be moved or evicted to generate area for the resized entry. Both move and eviction steps can be done off the critical path, but they imply in extra energy consumption, as multiple evictions/movements may be needed to make enough room. In addition, the co-allocated entries' metadata must be read and parsed to calculate the operations that must be done. Using fixed size segments may alleviate the fat write problem, as the allocated space is always rounded up, so there may be space for small expansions.

When applied to memory, compression further expands the set of challenges: while in caches lines are located using tags, the tag overhead becomes unbearable for larger memories; thus, in general, the main memory does not contain such structures. This complicates locating compressed lines. Furthermore, the Operating System (OS) relies on the memory size information, which, with compression, is no longer static. As the amount of memory provided to the OS is bigger than the physical amount, it cannot know when the physical memory is exhausted; so, while it may think there is still memory left to use, it may actually be running out [AF00].

Finally, one can also design a system where the whole memory hierarchy is compressed. In such a system, having multiple compression methods (*e.g.*, one for the cache hierarchy, and another for the main memory), although compelling — one could choose faster schemes for caches closer to the core, and effective schemes for memories farther away — may not be optimal. This is the case because transitions between memory levels would require constant decompression and compression; yet, if all levels used the same compressor, data could have been sent compressed to save bandwidth. Therefore, deciding whether to use a single compression scheme for all levels or not is of great significance.

Nonetheless, a fully-compressed memory hierarchy can greatly improve performance, and energy and bandwidth consumption when compared to conventional memory hierarchies [HR05, LHK00]. Having a higher effective capacity implies fewer accesses to lower memory levels, and fewer cache misses and page faults; so the overall system performance is improved. Besides, although there is more hardware and generally more static energy consumption, the overall power requirements can be reduced due to the lower number of misses to the memories farther

from the core, which would have spent more dynamic energy [NW15]. Moreover, the transfer time can be decreased due to the fact that compressed data is being sent — *i.e.*, more data can be sent in the same time interval [CR95, STBD14]. Finally, the benefits of compressing the whole system is proportional to the memory latencies; therefore, systems with high latencies can achieve higher performance improvement due to the reduction of accesses to higher memory levels.

1.5 Dissertation Structure

In this dissertation we analyze the strengths and weaknesses, point the key challenges, and introduce some ideas to improve both the efficiency and efficacy of cache compression. These are the main contributions of the work developed in this thesis:

— We present a thorough survey of compression, showing how proposals handle the multiple problems that arise from having compressed systems.

— We scrutinize the dependency of cache compressors on the data type granularity of the workloads. By achieving a better understanding of this relation we can detect design flaws and propose ways to overcome them.

— We formally define a generic base-delta compressor encompassing any number of bases. We also describe optimizations to its representation, which greatly increase its effectiveness when compared to the naive approach.

— Leveraging on the knowledge established by the granularity exploration, and the optimized generic base-delta, we propose multiple compressors. These attain high efficiency at a low decompression cost.

— We boost the co-allocatability of cache lines through Pairwise Space Sharing. Pairwise Space Sharing is an expansion to compaction algorithms which allows the benefits of unconstrained compaction with minimal tag overhead and no extra latency.

Chapters 2 and 3 present an in-depth description of systems with hardware cache, link, and memory compression, as well as the problems that must be managed when having a compressed layout. The background-related chapters end with Chapter 4, which scrutinizes hardware data compression algorithms from an evolutionary standpoint. Chapter 5 explores data similarity and co-allocation probability to introduce Pairwise Space Sharing, a method to co-allocate blocks that requires low metadata overhead and improves co-allocation efficiency. Chapter 6 scrutinizes cache compressors, analysing their relationship with data types. Moreover, it introduces multiple new compressors that build upon the acquired knowledge to achieve minimal decompression latency and high compression efficiency. We conclude with a summary of this dissertation and a few research directions for future work (Chapter 7). Chapters 2 and 4 have been accepted for publication [CS21].

We will use the term *memory* interchangeably for all cache levels and main memory, unless otherwise stated. The terms cache line, line, and block will also be used interchangeably.

Chapter 2

Understanding Compressed Caches

This chapter provides an overview of some cache compression schemes that have been proposed over the years, and how they handle the multiple problems that arise when designing systems with compressed caches. We hereby elucidate the problems and solutions in general terms, and explain what are the typical strategies adopted to handle cache compression.

Previous works [MV15, SASW15] group and describe the hardware compression proposals created over the years. They cover proposals on a high-level taxonomy perspective — compression algorithms, and compressed cache organization — and focus on describing each technique isolatedly. This manuscript takes a different approach, dissecting these techniques into their underlying solutions for each of the problems faced by compressed systems. We hereby elucidate the problems and solutions in general terms, and explain what are the typical strategies adopted to handle cache compression, referring to techniques as practical examples.

The following terms will be used throughout the manuscript: **compression ratio** is the ratio between the compressed size and the cache line size (Equation 2.1) [Sal04]; and **compaction ratio** — sometimes referred to in the literature as *effective cache capacity* — is the average number of valid sub-blocks in a data entry. The former measures how *efficient* a compression algorithm is, while the latter exposes the *efficacy* of the compression system (compressor + organization).

$$CR = \frac{original\ block\ size}{compressed\ block\ size} \tag{2.1}$$

2.1 Motivations

Data compression is modern day's alchemy. From circuits to web navigation, the goal of turning a big data chunk into a smaller one goes far beyond increasing the density of information; by storing more data in less space, one can potentially *save* area, energy, and bandwidth. Increasing the effective memory *capacity* is one of the most recurrent goals in systems with compressed caches. Capacity-focused compressed caches present the benefits of larger caches at a lower cost than physically augmenting the memory size. On the one hand, a physically enlarged

cache has larger silicon area, greater static power consumption, and slower accesses; on the other hand, compression circuitry is only a fraction of a cache's area [CYD+10, PSM+12, PS16].

Another common reason to apply compression is to reduce *energy* consumption: smaller caches have smaller footprints, leading to lower leakage and driving energy utilization [VZA00, YG02, KAM02, KPA04]. Furthermore, smaller data can be transferred using less energy [KL13]. Compression can also be used to improve *bandwidth*: smaller data transfers can be translated into faster transfers, higher throughput, or even bus-width reduction [KAM02, HR05, PSK+13, STBD14]. Another possible use of compression is to enhance tolerance for partially faulty entries — that is, compressing data so that it fits in the non-faulty sub-entries of cache entries [FSGAB+16].

Finally, compression can be used to reduce the *storage overhead of other techniques* for memory enhancement. For instance, techniques to prevent, detect and correct errors in memory units typically need to store specialized codes along the data [CH84]. These extra bits can incur a high area penalty, specially if high efficacy is desired [CCZ13]. In compressed caches some of the data space is freed, making room to partially or entirely fit these codes [JZZ+12, CCZ13, PKL15, KSGE15, YJ18]. This can be leveraged to reduce the size of the extra storage, as well as its access frequency.

2.2 Handling Data

Prior to selecting the compression algorithm — which will be discussed in Chapter 4 — multiple decisions must be made regarding the data being compressed; questions such as "What is the granularity of the compressor's input?", "Must the decompressed line perfectly match the original cache line?", and "What should be done in case the compressed size changes?" must be answered first.

2.2.1 Input Granularity

In general compression, a data set's potential to compress grows with its size. The larger the input size, the more values are likely to repeat, generating more compression opportunities. While compression typically does not restrict the input size, cache compression has a particularity: the input size is well-defined as the size of a cache line. Yet, the complexity of a compressed cache's compressor and decompressor increase with the line size; thus, enlarging the line has a direct negative impact in their latencies and areas.

It is possible, though, to simulate higher input sizes by changing the granularity at which the cache compressors compare values. Instead of trying to find value similarities within isolated lines — a concept called **intra-line compression** — one can find more compression opportunities by comparing the contents of multiple lines (**inter-line compression**) [TKJL14, NW15, PS16,

GNL20]. These granularities are not exclusive, so a compactor can use both the intra- and inter-line approaches simultaneously, in a hybrid scheme [GNL20]. More information on how these approaches work will be provided in Chapter 4.

2.2.2 Precision

Computing systems commonly require calculations to be deterministic. As a result, most compression algorithms tackle compression as a **precise** problem, and compression must act as an invertible function — a given line A must be uniquely compressed to line A', and the decompression of A' must regenerate A precisely.

Howbeit, a recent field of study called **approximate** computing proposes to consciously applying value approximation to speed calculations up, or reduce power consumption [Mit16]. This concept can be applied to compression under specific contexts and constraints. For example, images are composed of pixels, and the Least-Significant Bits (LSB) of these pixels's data representation contain information that, if removed, will likely not affect much the image quality. Compression could leverage this to remove such bits.

Compressed caches using approximation techniques typically use error/difference thresholds to determine which values should be considered similar, and classify their degree of similarity. Then, lines whose contents are considered similar are mapped to the same data entry (more on that in Section 2.3). Since the hardware is unable to automatically distinguish whether it can approximate contents, the programmer must inform which parts of the address space are approximable. This means that these compressed cache layouts allow both precise and approximate blocks to co-exist, either by splitting the cache into different areas [MAMJ15], or by adding metadata to the blocks to inform their approximation state [SMAJJ16].

2.2.3 Overwriting Data

Co-allocating blocks in a data entry is not a one-time problem that is solved after a line has passed through the compressor and has been stored in the cache. Further updates to the line can make its contents change, possibly requiring the line to be re-located. Three cases can arise when data is overwritten: the contents of the (un)compressed line do not change; the new compressed data is smaller than the previous contents (**data contraction**); or the new data is compressed to a size larger than the previous one (**data expansion** or **fat write**).

The first case is trivial: nothing needs to be done, so the line can be kept in its original location. Data expansions, on the other hand, are generally undesirable: an expanding block may not fit in its current allocated space anymore, requiring special handling. Finally, data contractions are somewhat easier to handle: one can either decide to keep the line in its current location, filling the emptied space with padding bits; or handle it analogously to expansions. Handling the resizing follows the compaction scheme's replacement guidelines: one can either

remove the overwritten line itself from its current location, re-applying the co-allocation process (*re-insertion*); or move/evict other co-allocated lines in the data entry to *make room* for the expansion to happen. This handling, although inconvenient, is rare, and can be done off the critical path [AW04a, SSW16, PHC+15].

As a side note, compression schemes that enforce that all co-allocated blocks must fit in a given size may require lines whose size changed to always re-locate if the size threshold is crossed, even if there is space available for the block to fit in its current location [SSW16, SSW14]. For example, in Skewed Compressed Cache [SSW14] blocks are indexed as a hash function of their address and compressibility; thus, if the compressibility of a block decreases [1], so does its expected location.

2.3 Mapping Compressed Data

Data stored in a cache must be located in order to be accessed. To that end, conventional cache designs contain both a tag-metadata storage, and the data storage itself [Smi82]. This metadata storage encompasses information that is relevant to identify which memory data is stored at a given place in the data storage. Typically, the mapping between these storages is bijective; each cache line fits exactly in a data entry, which is associated with a unique tag identifier, even if not valid. When data is compressed, however, this is not the case anymore, and multiple cache lines can be stored in a data entry. As a result, the mapping between storages must be modified.

The straightforward solution is to remove the injection property by increasing the number of tags. This is the standard approach adopted by most proposals in the literature, being applied by associating multiple tag entries to either a single (many-to-one) or multiple (many-to-many) possible data entries. Examples of how state-of-the-art proposals generally approach such designs are presented in the following sub-sections.

2.3.1 Many-to-One Mapping

Mapping multiple tags onto a single data entry is the simplest approach; thus, it is the go-to strategy among many compressed cache layouts (*e.g.*, SCMS [LHK99, LHK00] and YACC [SSW16]). Although such techniques come in multiple flavors, each with its with minor nuances, the general idea can be described as follows (see Figure 2.1): (1) each data entry is uniquely coupled with a few fixed tag entries. (2) When a block's data is inserted into a data entry, one of its respective tags is used — its coherence bits are set (C bits), and the respective compression metadata is stored (M bits) — and the block's contents are copied over. (3) If another block is

1. When the compressibility of a block increases the extra space can be filled with padding bits to avoid moving the block.

assigned to this data entry, and is able to co-allocate with the existing block, another tag entry is used, and the new block is stored alongside the previous one. This process is repeated until this data entry has no spare tag entries, a point at which tags would need to be evicted following the replacement policy's guidelines (R bits).

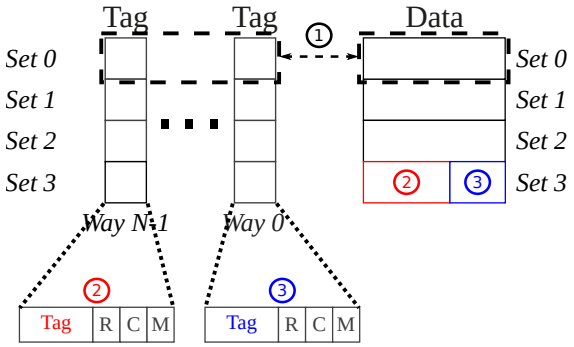


Figure 2.1 – A generic many-to-one compressed-block mapping.

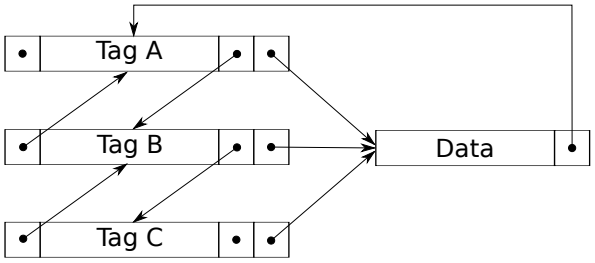


Figure 2.2 – A generic tag-data doubly linked list. Each tag entry has extra metadata to inform which are the previous entry, next entry, and corresponding data block. Each data entry contains a pointer to the head of its corresponding tag list.

The previously described design works well for compression in general; however, for the specific case of inter-line granularity (see Section 2.2.1), one can opt not to compress data entries, but to associate multiple tags with a specific data entry instead — that is, a *variable* number of tags can map to a single data entry. This flexibility can be achieved by connecting tag entries through a doubly linked list, and mapping each list to a single data entry [TKJL14, MAMJ15, GNL20]. Eviction of a data block requires that each tag entry contains a pointer to the previous and next tags, as well as a pointer to the data entry it is associated with. Finally, since the storages must be effectively fully decoupled, each data entry must hold a pointer to its respective tag list's head. The general design is shown in Figure 2.2.

To avoid having to perform full storage scans, these techniques typically employ hash tables, which allow quickly locating blocks based on their contents; if two blocks exhibit similar values, they should be mapped to the same data entry. On lookups and writes this map value is generated, and the cache is searched for matches, first in the tag array, and then in the map tag array (hash value), to find the corresponding data block position. Special care must be taken on replacements, as the removal of a data block implies on the removal of several tag entries. As a result, previous to removing an entry to insert a new one, the cache can be searched for similar values: on matches it is simply appended to the match's linked list; otherwise, all tag entries referencing the to-be-evicted data block must be properly invalidated.

2.3.2 Many-to-Many Mapping

When there is an M to N relationship between the tag and data storages to accommodate for compression, the cache design becomes slightly more complex; yet, the tag and data entries have more flexibility, allowing for better placement decisions. Compressed caches that decouple tag and data are typically subject to at least one level of indirection, and the ideas usually derive from similar schemes conceived for conventional caches, such as the Indirect-Index Cache (IIC) [HR00] and the Decoupled Sectored Cache [Sez94]. The main difference between decoupling in uncompressed and compressed caches is the granularity: since compressed blocks can fit in spaces smaller than the size of data entries, such entries can be partitioned into multiple small *segments* [AW04a, HR05, SW13, AS14b, PHC+15] (more on segments in Section 2.3.4).

For instance, in the Decoupled Variable-Segment Cache (DVSC) [AW04a] the data entries are divided into eight-bit segments, and despite the fact that each set contains four data entries, the tags assume a 8-way set-associative configuration — *i.e.*, there can be at most eight blocks simultaneously present in a set. Each tag is modified to contain the compression status and a counter of the number of segments used by the data it refers to. On compression, a block is rounded up to a number of segments, and inserted in the set in contiguous address order, that is, at the segment after the last used one — a block's segments can then span over two physical data entries. Due to the compelled contiguity, a block update may require segments to be re-compacted if the new size is different from the old.

2.3.3 Reducing Tag Overhead

Increasing the number of tags has an exceedingly high cost: the tag storage overhead is increased manyfold. Besides, the usefulness of the extra tags is directly proportional to the workload's compressibility; thus, some approaches group multiple tags into a single tag entry, notably reducing the number of tag bits [ATGK07, SW13, SSW14, SSW16]. This can generally be achieved through the exploitation of two remarks: neighbor values tend to exhibit approximate similarity, and workloads tend to manifest high spatial locality [SW13, SSW14, SSW16].

The former remark implies that the contents of neighbor blocks are approximately equal. This can happen, for example, in neighbor pixels of images, which likely contain similar values; and when safeguarding from worst case scenarios — *e.g.*, using data structures bigger than the average value [MAMJ15]. Besides, applications tend to display a high frequency of zeros [ES05], as it is regularly used to nullify pointers and initialize data. Finally, the second remark states that neighbor blocks are usually brought in to the cache within a small time interval. This means that neighboring blocks are usually located in the cache simultaneously, and their compressibility will likely be similar [SW13]

Therefore, adjacent blocks can be grouped into bigger blocks, called superblocks (SBs) — blocks that share a single common tag to reduce tag overhead (Figure 2.3) — in a concept

derived from sector caches [Oln85]. This can be done because the individual tags of neighbor blocks differ by very few bits, so the common bits can be shared under a common tag field. Under this configuration, blocks within a superblock have minimal tag-storage cost, needing a couple of extra bits to inform per-block metadata (coherence state, replacement policy, etc.), and possibly some bits to store their tag offsets. This allows significantly reducing the inherent metadata overhead required to track multiple compressed blocks.

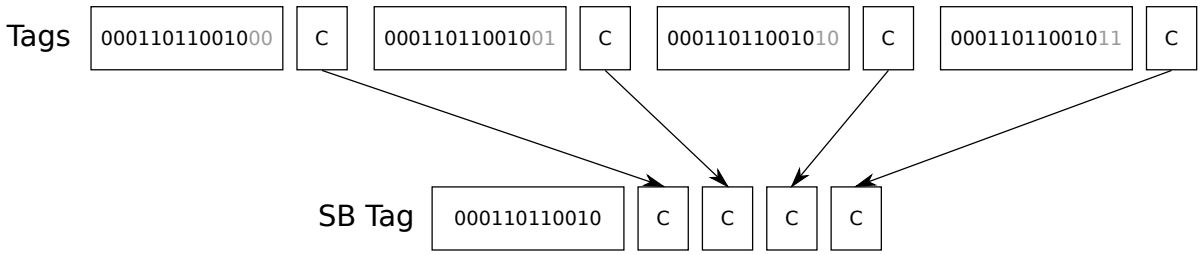


Figure 2.3 – Tag bits can be deduplicated if the tag field is reduced to accommodate only the Most-Significant Bits (MSB) of the original tags. Since in this example there are four blocks per superblock, and the tag offset is only two-bits long, the differing tag bits can be stored implicitly though the order of the coherence fields.

There is a major drawback of using superblocks, however: the number of co-allocation opportunities is drastically reduced. While previously any blocks could co-allocate, within a superblock only neighbor blocks can co-allocate. At a high level, superblocks are applying a base-delta compression algorithm (see Chapter 4) to tags; therefore, this concept can be expanded by changing the compressor being used [NW15] to find a good trade-off between size reduction and co-allocation opportunities. Alternatively, the maximum number of tags in a superblock does not necessarily need to match the amount of bits in the tag offset (number of neighbours) [YLK+04]. This allows sacrificing tag-size reduction and tag-locating complexity to achieve a similar compromise by explicitly storing the tag offset bits of each tag.

Making tags smaller reduces the burden of augmenting the tag array; yet, this overhead still exists. To effectively remove this extra compression cost, some techniques propose storing tags in the data entry [NW15, HAB+19]. For instance, Touché [HAB+19] stores the extra metadata in the data entry itself. When entries contain compressed data the tag array is populated with many short signatures (e.g., one 30-bit tag is replaced by 3 9-bit signatures, each referring to a different co-allocated block), instead of storing conventional tags. Since this process effectively reduces the tag size, it increases the number of tag conflicts, which generates false positives when performing lookups; thus, a copy of each original tag must be stored and accessed within the data itself in order to confirm matches. This reduces the available space to store the compressed data, and increases the average hit latency, but removes the need to increase the tag array's size.

2.3.4 Finding Segments

Most of these techniques make data smaller with the purpose of fitting more lines into the data entries. This means that, besides the conventional tag-data mapping, they must supply a way to locate the lines at a smaller granularity — *i.e.*, they must inform not only which data entry(ies) contains the data, but also where within such entry(ies) the line has been allocated. The simplest way to determine where a block is placed is to have a strictly **constrained** number of placement possibilities and enforcing a **contiguous** storage — the block's contents must not be spread over the data entry. Then, a compressed block's location can be dictated by its compressed size or other location metadata [LHK99, SSW14, SSW16]. Figure 2.4 displays placement under different constraint and contiguity configurations.

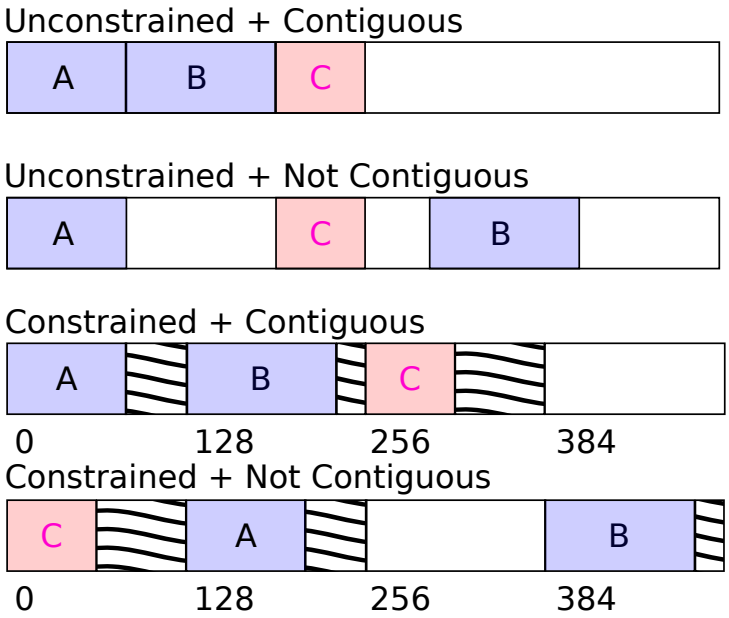


Figure 2.4 – Placement of compressed block C in a cache entry containing compressed blocks A and B, under different configurations. Stripes are wasted space.

Although the *constrained and contiguous* approach greatly cuts metadata overhead, it restricts the co-allocation opportunities, since limited placement locations translates to limited size choices (*e.g.*, 8, 16, 32 and 64 bytes). Blocks that are able to compress to sizes smaller than the possible sizes will waste the remaining space with padding bits; furthermore, requiring blocks to be stored contiguously means that if an overwrite to a block happens, and their compressed size changes, blocks may need to be rearranged — a process called *recompaction*.

To make the most out of compression, compressed systems must explicitly keep track of the compressed block's location. This can be achieved through the addition of a field containing the integral (*i.e.*, **unconstrained**) size [AW04a, CYD+10] — *e.g.*, in a cache with 64-byte cache

lines this adds up to 7 bits per tag — or pointers to indicate where the block is placed. Pointers can either associate the segments to their tag [SW13, GNL20], or the tag to its segments [HR05, GNL20]. Additionally, the use of pointers allows relaxing the contiguous property: multiple pointers can be added to specify the location of the dispersed segments of the compressed blocks, removing the need to recompact data [HR05, SW13]. Figure 2.5 shows the organization of the Decoupled Compressed Cache (DCC) [SW13], which is an example of how pointers can be used to map segments to tags.

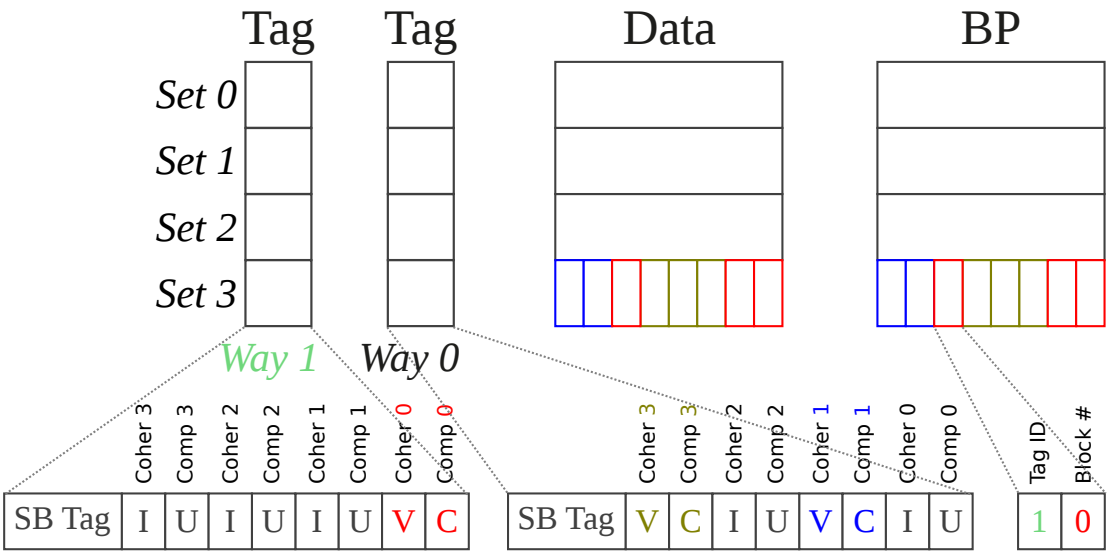


Figure 2.5 – Locating segments in DCC. The tag in set 3, way 1 contains a single valid compressed block at sub-block 0 (its respective coherence state is valid, and compression bit is set). The back pointer (BP) array contains 3 entries related to this sub-block: 0, 1 and 5. The tag ID of each of these BP entries is set to 1 to match the way at which its respective sub-block can be found, and the block number is set to the number of the sub-block. The corresponding data entries are highlighted with matching colors.

2.3.5 Specialized Caches

Many authors suggest adding extra smaller caches to store compressed or complement data. These special caches build upon different concepts and characteristics of compressed data; therefore, they are usually orthogonal approaches, and can be simultaneously applied to further enhance the system. The most common idea behind these specialized caches is to provide smaller data representation for frequent values: as shown by Zhang *et al.* [ZYG00], these values account for a significant percentage of accesses, while occurring in nearly half of the different memory locations, and generating a large proportion of caches' misses. In particular, zero values present high predominance over other values [KGJ98, ES05].

In a frequent-value-based specialized cache some values are defined as frequent (*e.g.*, by static or dynamic profiling) [ZYG00, DPS09]. Lines containing these values can then substitute their occurrences by their respective indices in the frequent-value reference table. In such a system a block's location depends on its contents; thus, special care must be taken to keep coherence, and accesses must perform simultaneous lookups on both conventional and specialized caches.

As in the compression algorithms themselves, one can change the granularity at which frequent values are dealt with. For instance, one can explore the lower variance of MSBs to reduce the width of a data entry. The main cache becomes then a storage of the LSB of cache lines, and each entry is paired with a compressed (*e.g.*, frequent value or sign extension) representation of its MSB. The specialized cache can either store the MSB of all lines in the main cache — *e.g.*, as in the Frequent Value Cache (FVC) [YG02] — in which case compression is used to avoid accessing the secondary storage; or part of the lines — *e.g.*, as in the residue cache [KLKH11] — in which case hits to partial lines need to request the missing data from the lower cache levels.

2.3.6 Other Designs

The proposals presented so far consider the use of caches with a somewhat conventional tag and data array design. Compression is not limited to this layout, though — other cache designs which significantly differ on the way data is found and accessed have been proposed over the years [QL12, NW15, JWN10, TGS18], and can be adapted for compression support.

For instance, alloy caches [QL12] have been designed for DRAM caches. They embed the tag information along its data counterpart to make accesses concise (tag and data are accessed simultaneously, in a Tag and Data (TAD) entry). This idea is expanded upon to accommodate compression in Dynamic-Indexing Cache Compression (DICE) [YNQ17], where each TAD entry supports a variable number of lines: as long as there is space available, blocks can co-allocate. The number of tags is informed by the addition of a bit to each tag informing whether there is a next tag, or if the data portion starts. A similar approach fixes the number of tags at two per compressed TAD to simplify indexing [AA18].

Most of the organization schemes described earlier focus on improving cache performance for single stream applications. Although they can be used in multi-core scenarios, it is not part of the method's design. The Manycore-Oriented Compressed Cache (MORC) [NW15], however, focuses on multi-core architectures, by improving throughput. In MORC sets are substituted by logs, forming a log-based cache: data is appended to entries based on data commonality patterns, not data address. Writing data in MORC is as simple as adding to the end of the active log and invalidating the old version, if it exists. This means that locating data is not trivial, and requires the use of some level of indirection.

Some local memories do not need to store tags — the *scratchpads* [JWN10]. Scratchpads are an alternative to conventional caches, mainly used in embedded systems. Contrary to transparent

caches — whose address space is a subset of the main storage's — scratchpads use a different address space; thus, any requests to a given address belonging to the scratchpad space are guaranteed to hit. Data moved from the main storage to the scratchpads is stored as independent temporary copies, which can — but does not need to — be written back when evicted.

This tag-less kind of cache can be used to reduce the tag footprint of compressed caches. For example, Zippads [TS19] applies compression to a software-managed object-oriented cache hierarchy based on Hotpads [TGS18]. In Zippads the data array is composed of objects, not lines; as a result, deduplication can be done between the same fields within objects, which yields better results than blindly comparing lines in object-oriented applications. New objects are always appended at the end of the data array; and, if a data expansion happens, the old entry becomes a forwarding pointer to the newly appended object. When a pad is filled, a process similar to garbage collection is triggered to free unused objects and recompact data.

2.4 Compression Usefulness

Compression has an inherent drawback: it adds a decompression step to cache hits that contain compressed data. Since one of the goals of increasing the effective cache capacity is to reduce the average access time, speedups are generally achieved when the total miss latency reduction overcompensates the hit latency increase. Consequently, compression can be a burden if an increase in cache capacity is not the workload's primary need. For instance, compression is a waste of energy and a performance limiter whenever: a block is compressed, but does not manage to co-allocate with any other blocks; the extra access latency due to decompression surpasses the latency of a miss; a block co-allocates, but is never re-referenced before eviction.

Since the impact of the decompression latency is proportionally smaller on higher-latency caches, compression is typically proposed for the LLC; nonetheless, some proposals tackle compression on caches closer to the core too [YZG00, TZ07]. It is also worth noticing that compression can be co-applied with other miss ratio reduction techniques such as prefetching and optimized replacement policies [AW07, PHM15] to further enhance the system's performance (more on that in Section 2.5).

2.4.1 Adaptive Compression

A simple approach to reduce the likelihood of having compressed blocks stored without companions is to apply a threshold to the compressed size, such that a block is only stored in compressed format if it is reduced to at most a certain number of bits [LHK99]. This is implicitly used by superblock-based compaction schemes with fixed size sub-blocks — such as Skewed Compressed Cache (SCC) [SSW14] and Yet Another Compressed Cache (YACC) [SSW16] — since they enforce that compressed blocks must fit in either 16 or 32 bytes.

Preemptively deciding whether compression is useful based on a fixed — *i.e.*, static — factor can be quite limiting considering that workloads tend to exhibit dynamic behavior. Besides, it limits compression: a block that has been compressed to a size $A > 50\%$ would still be able to co-allocate with a block whose compressed size is $B \leq 100 - A\%$; therefore, some techniques try to minimize this overhead by adaptively enabling and disabling compression, or switching between compressors based on the overall compressibility of the workload and the effects of compression on performance [ADS15, PS16, ALSW18]. Others focus on determining how much of the storage should be dedicated to compressed data [TG05, TFR+01].

Enabling and disabling compression can be done indirectly: the Indirect Index Cache with Compression (IIC-C) [HR05], for example, modifies its replacement policy based on the observation that the fewer misses there are, the lower is the usefulness of compression. In IIC-C the need to move blocks between the different priority queues of its specialized replacement policy is proportional to the number of misses; thus, blocks are only compressed when they move. In any case, the majority of the adaptive compressed systems take an active role on compression control. For instance, one can use a counter to track whether compression is being helpful by training it positively on avoided or avoidable misses — accesses to blocks that would only be present in the cache due to the existence of compressible entries — and training it negatively on penalized hits — accesses that would be hits regardless of the use of compression [AW04a]. Since the usefulness of compression is workload-dependent, a fine-grained approach (*e.g.*, a counter per core) can improve enablement-prediction accuracy [XL11].

Notwithstanding the potential benefits, switching between enabling or disabling compression may introduce overhead. For example, when turning a compressed cache into an uncompressed one, there may be not enough space for all currently allocated blocks; in this case, some of them will need to be evicted. For this reason, Xie *et al.* [XL11] propose using a Decision Switching Filter (DSF) with three parameters. One of the parameters controls the lowest access time improvement that would make it worthwhile switching compression state. The second parameter is the minimum access time difference needed to make switching beneficial. Finally, the previous parameters establish a range of access times in which it is uncertain whether switching is advantageous. The third parameter decides how long the system should stay in this intermediate state before a decision to switch the compression activation state is taken.

2.4.2 Avoiding Frequent Decompressions

When a compressed block is read too frequently, the benefits of avoiding accesses to the lower-level memory may be subdued by the accumulated decompression latency. This drawback can be lessened by preemptively decompressing such blocks, so that future reads skip the decompression step. By definition, this is done to a certain extent by the caches in between the core and the compressed memory [HR05]. Anyhow, one can add a decompression buffer to the compressed

cache itself, which must be kept coherent with the cache [LHK00, TFR+01]. The coherence handling can be avoided by expanding the block in the cache itself [PBL+17]; however, this has the downside of reducing the effective cache capacity due to extra evictions. Deciding which blocks to decompress follows guidelines similar to conventional replacement policies — *e.g.*, access frequency or recency [LHK00] — possibly incorporating the compressed size into this selection process [PBL+17].

2.5 Interactions and Summary

There are multiple components and policies in a cache, each of which is designed under specific assumptions. Frequently, one of these assumptions is that each single line occupies a whole data entry; however, with cache compression, this is not the case any longer. In this section we list some of the side effects and interactions that have been observed in compressed caches, and how some proposals modify such policies and components to attain a cohesive design. Finally, we conclude this section with a summary of the compressed-layout literature listed in this paper.

2.5.1 Interaction with Indexing Policies

Changes to the cache design to accommodate compression do not need to be a complete makeover; some techniques simply change the way blocks are indexed to take advantage of the characteristics of compression. For example, under traditional (uncompressed) indexing adjacent lines map to adjacent sets. This means that compressed caches co-allocate lines that are spatially distant from each other when using this scheme. As mentioned previously, spatially close lines present similar compressibility, and are more likely to be used within the same timeframe; thus, compressed systems that modify indexing to maximize co-allocation can improve performance and bandwidth consumption [KPM15, YNQ17].

For example, Young *et al.* [YNQ17] suggest using traditional indexing when data is not compressible, and a special spatial indexing otherwise (Bandwidth-Aware Indexing (BAI)). Cache lines under BAI are either on the same set or the neighboring set as they would be under traditional indexing, so that consecutive cache lines map to the same set, instead of several blocks apart. This allows successive data to be compressed together in the same set; therefore, accesses to a compressed line can provide multiple spatially neighboring lines, while still keeping some similarity with the traditional approach. Since this scheme relies on the compressibility of the data set, special care must be taken to determine when to prioritize each indexing policy.

The indexing policy's hashing functions can also be modified to better support compression. For instance, cache skewing [Sez93] applies a different hash per way. This may hinder co-allocation, since lines that co-allocate well may be placed apart from each other. A com-

pressed cache that uses skewing — *e.g.,* SCC [SSW14] — can cope with that issue by mapping blocks based on a function of the way, address, and their compression factors. This reduces the number of conflict misses without changing the cache's associativity.

2.5.2 Interaction with Replacement Policies

Although conventional replacement policies can be used with most of the compression techniques previously described, compression and replacement policies can interact negatively [BLN+13, PHC+15, GAS16, PBL+17, PS18]. For example, assuming the use of a LRU replacement policy, when a MRU cache line is co-allocated with the LRU block, either the MRU line will be co-evicted, or the policy must be given flexibility to search for another less controversial entry. In addition, inserting a block may require the eviction of multiple lines due to the removal of a shared tag, or evictions not freeing enough data-entry space for the new block. This means that a compression-aware policy might result in a more efficient cache space utilization. For example, a compressed data's size is important to determine how many other blocks can be co-allocated with it; therefore, a replacement policy that takes the co-allocatability of a block into account could improve the effective compression ratio. Nonetheless, co-allocatability should not be considered the paramount factor; otherwise, a small cache line with low temporal locality would occupy space indefinitely [BLN+13].

Multiple compression-aware replacement policies (*e.g.*, ECM [BLN+13] CAMP [PHC+15] and HoPE [PBL+17]) derive from Re-Reference Interval Prediction (RRIP) [JTSE10], a simple — yet efficient — replacement policy. RRIP assigns a value to each entry, indicating its distance from being re-referenced. If this value is 0, it's likely to be re-referenced in the near future, otherwise its next reference is distant. On hits the value is reset to 0, and on misses all values are incremented until a victim (block with the highest possible value) is found. RRIP can become size aware by, for instance, assigning different initial values for newly inserted entries based on their compressed sizes. For example, Effective Cache Maximizer (ECM) [BLN+13] was proposed for segment-based compressed layouts. It decides the size thresholds to assign different values dynamically, based on the physical memory usage ratio and the average segment usage per set. When choosing from the pool of eviction candidates, ECM uses the size information to select the entry that frees the most data space.

Panda *et al.* [PS18] notice that more than 55% of compressed and uncompressed blocks are used only once before being evicted; therefore, it would not be beneficial to waste tag storage with these blocks. To deal with this situation, they propose the Synergistic cache layout for Reuse and Compression (SRC), which applies the principle of the reuse cache [AInVnL13] to caches using superblock compression, such as YACC [SSW16] and SCC [SSW14]. SRC expands superblocks with a "first use" field to defer data allocation. Whenever an access to an invalid block whose superblock was already present happens, instead of allocating the data, the bit is

set. This way, an effective write of the data contents only happens when blocks are reused.

Alternatively, one can take an opportunistic approach and not include replacement data for the co-allocated lines. For instance, the Base-Victim cache [GAS16] *logically* divides cache entries into a *Baseline* and a *Victim* Cache, associating two tags per physical data entry. Whenever a Baseline entry is evicted, a special step is added to the eviction: if the evicted entry can co-allocate with another Baseline block, then it is saved as a clean entry in that Baseline block's respective Victim area; therefore, future read references can still be served by this cache level. As the Victim Cache consists of clean lines, evicting them is free of data traffic. The replacement policy only keeps track of the main line (Baseline Cache) in every entry; therefore, by design, the Base-Victim cache cannot have a miss rate higher than an uncompressed cache. Read and write hits in the Baseline cache are trivial, and write hits to the Victim Cache cannot happen due to the enforced inclusive property. When a read hit happens in the Victim Cache, the block is promoted to the Baseline in a fashion similar to a regular miss, with the advantage of not needing to access the lower level memory.

2.5.3 Interactions with Prefetching

Other works address the interactions with prefetching [AW07, ATGK07, PSK+13, AS14b]. Prefetching and compression can have an interesting synergy: bad prefetches can bring blocks that end up polluting the cache; however, in a compressed cache these blocks may co-allocate with useful blocks, reducing the drawbacks of those bad prefetches. In addition, good prefetches may take further advantage of the increase in the cache's effective capacity.

For instance, Prefetched Blocks Compaction (PBC) [KPM15] explores the similarity of prefetched lines to co-allocate them through inter-line compression and maximize the use of the effective cache capacity. It splits the cache into two different-sized caches: a compacted and a conventional part. To avoid recompactions and data expansions, the compacted cache only stores prefetched (non-dirty) entries; therefore, the block must be moved to the conventional cache when a writeback happens from a higher cache. Blocks are also moved when there are no available entries in the compactable cache. To control underutilization when not many prefetches are generated, or the dataset is not sufficiently compactable, PBC can change the size of the compactable cache dynamically.

2.5.4 Interactions with Security

Besides the direct drawback on the hit latency, compressed systems can be hazardous at a security level: the compressibility of data leaks information that can be explored to retrieve the data itself [Kel02]. The key idea is that an S-bits data block whose compressed size is S' bits reduces the exploration space of attacks, since only a subset of the possible 2^S values can compress to S' through a given compression algorithm.

In the context of hardware data compression, Tsai *et al.* [TSFS20] have applied this exploit to BDI [PSM+12]. Assuming that an attacker controls the contents of a line, except for the x-bit secret it wants to unveil, their proposal combines brute force with exploration space reduction. An attacking line is generated such that it is only compressible if the secret's $b-d$ most significant bits — where b and d are the base and delta sizes, respectively — match the respective bits in the base being forced. All possible variations of the $b-d$ bits are tested until compression is successful — *i.e.*, such $b-d$ bits of the secret become known. Then, the exploration space can be halved, and d is reduced to d' to discover the next $\frac{b-d}{2} - d'$ bits. This process is repeated until all bits are found. Although conceived for BDI, this concept can be applied to any dictionary-based compressor by selectively filling the dictionary and identifying which pattern matches the secret. While this paper did not show the possibility of a realistic attack on a compressed cache, it pointed out that compressed caches can leak information to an attacker.

2.5.5 Summary

Table 2.1 presents a summary of multiple state-of-the-art cache compaction methods. *Granularity* expresses whether deduplication occurs between values in a line, or between lines. The *Expansion* field informs the action adopted when a compressed block goes through a fat write. *Mapping* is the mapping between the tag and data storages. *Segment* contains the size of the smallest possible sub-division of a data entry, in bits. *Tag Optimization* informs if the tag representation is somehow optimized. Finally, the *Details* field adds some extra relevant information regarding the proposal.

Technique	Granularity	Expansion	Mapping	Segment	Tag Optimization	Details
Base-Victim [GAS16]	Intra	Evict Victim	M:1	32	-	Logical division of lines. Victim line is stored clean.
Bunker Cache [SMAJJ16]	Inter	Not applicable	M:1	512	Address aliasing	Not precise. Multiple addresses are remapped to a single one.
CC [YZG00]	Intra	Re-insert	M:1	256	-	Increases line length to store compression metadata.
DCC [SW13]	Intra	Make room	M:N	128	Superblock	Each segment has a pointer to its tag.
DICE [YNQ17]	Intra	Make room	M:M	32 [a]	Shared Tag bit	Non-contiguous segments. Is an Alloy Cache [QL12].
Dedup [TKJL14]	Inter	Re-insertion	M:1	512	-	Doubly linked list.
Doppelgänger [MAMJ15]	Inter	Re-insertion	M:1	512	-	Not precise. Doubly linked list. Extra cache.
DVSC [AW04a]	Intra	Make room + Recompaction	M:N	8	-	Adaptive.
FCMS [YLK+04]	Intra	Not informed	M:1	64	Superblock	Has a small decompression buffer.
IIC-C [HR05]	Intra	Make room	M:N	256 [b]	-	Non-contiguous segments. Adaptive.
MORC [NW15]	Intra,Inter	Re-insertion	M:N	N/A	Tag compression, Tags in data entry	Log-based storage.
O2W [AA18]	Intra	Make room	2:2	16	-	Two fixed line locations. Is an Alloy Cache [QL12].
Pair-Matching [CYD+10]	Intra	Make room	M:1	256	-	-
SC² [AS14b]	Intra	Make room	M:N	8	-	-
SCC [SSW14]	Intra	Re-insertion	M:1	128	Superblock	Uses skewed indexing.
SCMS [LHK00]	Intra	Make room	M:1	256 [c]	-	Has a small decompression buffer.
Thesaurus [GNL20]	Intra,Inter	Re-insertion + Recompaction	M:N	64 [d]	-	Doubly linked list. Extra cache.
YACC [SSW16]	Intra	Make room	M:1	128	Superblock	-

a. Not informed. Inferred from the compression-metadata field size.
b. In a 128-byte line configuration.
c. The maximum compressed size is equal to the minimum compressed size.
d. The minimum number of segments is two.

Chapter 3

Organization of Compressed Memories

Compression is not an idea exclusive to caches. It can be further applied to memories, and the links in between memory levels. In this chapter we provide an overview of the link and memory compression techniques that have been proposed over the years, as well as the problems and solutions faced when designing such systems. We conclude with a section discussing what are the challenges faced when all these techniques are combined to form a compressed memory hierarchy.

3.1 Memory Compression

The cost of DRAMs is exceedingly high compared to alternatives such as disk or flash memory [BH09], and designers of servers and high-end computers have to find the best compromise to come up with a cost-effective system. This is a situation in which compression can be helpful; it can increase DRAM's effective capacity at a fraction of the costs. Although still rare, an increasing number of companies have been adopting memory compression [TSW+01, QT17, AB19].

Nonetheless, compressing the memory has extra challenges, when compared to cache compression. For example, page allocation is typically handled by the OS, which uses a fixed memory capacity to infer where to put data, and which pages should be swapped [SGGS98, BL17]. When compression is being used, the conventional notion of fixed spaces to allocate data is broken, and page allocation must take into account how much free space exists, and how much would be generated with a page's eviction [AF00].

Another challenge occurs in systems where the cache tags are derived from physical addresses. Although straightforward in conventional caches, this approach needs to be adapted when the memory is compressed. This is the case because a compressed line's address typically does not correspond to the location it has been physically allocated to; therefore, a mapping mechanism must be added in such systems [PSK+13, ZLC+15]. Therefore, when implementing a compressed memory, the OS, the Translation Look-aside Buffers (TLBs) and memory controllers should be

aware of the compression, and possibly even take an active role on it [FHW99, AF00, RKP01, PSK+13, ZLC+15].

Although all the techniques presented in this section are applied to DRAM, there are works developed for Phase-Change Memories (PCM) hybrid memories and other multi-level Non-Volatile Memorys (NVMs) [DZC+13, BLNK14, PM17]. They use algorithms derived from the ones presented in Chapter 4 to compress and decompress stored data. We do not address these methods, as they are typically used as solutions to help make these technologies viable (*e.g.*, reducing the number of writes in PCM), and not as a direct system performance improvement. Finally, compression can also be applied through software [RKP01, TG05, IBM], aided by the compiler [LHK00], or to the Graphics Processing Unit (GPU) [CSO+19, SSK12, ALSW18, VPJ+15, PBV+16].

3.1.1 Mapping

Memories contain pages, which contain lines. In conventional caches each line has a fixed offset within the page, which makes locating them trivial and does not require metadata; however, in compressed memories the line's compressibility accumulate, and determining where a line is located relies on the accumulated compressibility state of its surrounding lines. To simplify location hardware, the line's compressibilities are typically limited (*e.g.*,, lines can only be compressed to four different possible sizes) [ES05, CEA18]. Analogously, this size-variability problem and solution applies to pages. An alternative solution is to add explicit pointers to the data [PSK+13].

Bandwidth-focused memory compression proposals, on the other hand, have simpler mapping. Their goal is to provide more data with a single access, so they typically map a few neighbour lines to a fixed data entry, based on the compressibility of each line. For example, assuming that the maximum compressed throughput is of four neighbours, a bandwidth-focused memory compressor could map lines as in Figure 3.1. In this case, block B is found in A's data entry if compressed, and in its original location if uncompressed.

The extra metadata required by these solutions must be stored somewhere, and will occupy a large amount of space. For example, even if every 512-bit line requires only a single extra metadata bit, the total metadata overhead for a 4-GB memory is 8MB. Because of this, it is a standard approach to store such metadata in the memory itself.

This has a major drawback: now every memory access must perform two accesses - one to retrieve the metadata, and other for the access itself. This is a severe penalty, and most likely would render compression disadvantageous. A solution adopted to cope with this is to add a metadata cache [PSK+13, ZLC+15]. This cache has a significantly lower access latency than the memory itself, and relieves part of this double-access pressure. As in regular caches, miss-mitigation schemes, such as prefetching [ZLC+15], can be used to improve performance. Finally,

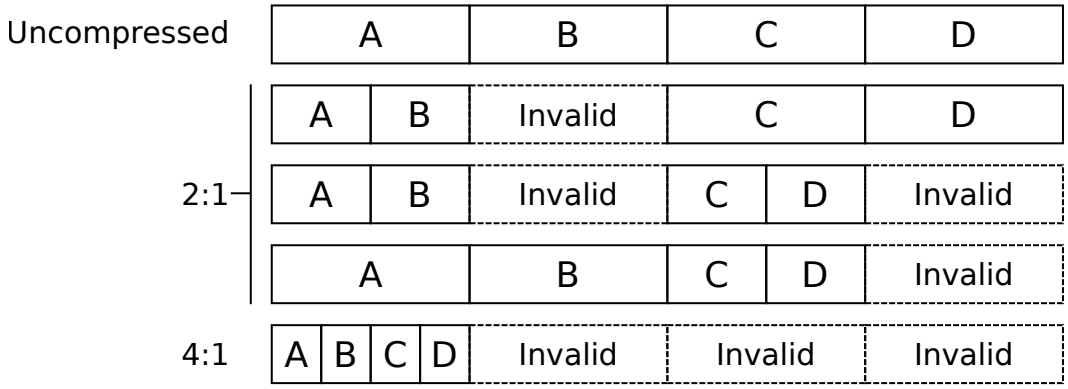


Figure 3.1 – Possible block locations for adjacent blocks A, B, C and D given different compression factors.

predictive methods can be added to access the data entry speculatively [PSK+13, ZLC+15, YKQ19].

3.1.2 OS support

The OS relies on the knowledge of the memory size for its memory allocation routines. In a system with a compressed memory focused on increasing its effective capacity, the available size becomes variable; yet higher than what is physically supported. This is a complicated situation: if the OS is not made aware of this possible size increase, compression is wasted; however, if it believes it has more memory than compression is currently being able to provide (*e.g.*, the workload's compressibility is sub-optimal), the OS may overcommit, and request more memory than available.

One possible solution to avoid overcommitting is to make the threshold at which a kernel triggers page swaps listen to controller-raised interrupts/flags [AF00, ZLC+15], instead of defining it statically. Overcommitting is not the sole OS-related issue; some pages may contain shared information regarding compressed pages, or how to perform high-priority operations (*e.g.*, page fault handling), and compressing them may generate recursive errors. A compressed memory must be able to handle these special cases to avoid incoherent or faulty behavior [PSK+13]. A simple solution would be to disable compression for such pages.

Finally, the OS can take an active role on compression. This includes, for example, compressing data coming from disk, or actively keeping track of pages and their sizes to reduce the probability of swaps [FHPR01].

3.1.3 DRAM-based Proposals

In this sub-section we will briefly describe how some of the previous work use the previously described solutions to generate a compressed memory system. A summary of these proposals is given in Table 3.1.

SCMS

One of the earliest memory compression proposals, Selective Compressed Memory System (SCMS) [LHK00], takes a fairly restrictive approach to minimize the negative effects of variable-sized pages. It allows two page sizes: small, in which all lines are compressed to half their size, and large (uncompressed). Moreover, pages cannot cross the normal page boundaries; therefore, small pages must be allocated in pairs. This minimizes the compression-related upkeep and eases locating pages, at a high negative impact on the system's compressibility.

Memory Expansion Technology (MXT)

IBM's MXT [TFR+01, TSW+01, FR01, FHPR01, BFR01, AFS+01] combines several approaches to create a compressed main memory that doubles the main memory's effective size at a low cost. It uses a Sector Translation Table (STT), which is an array of attributes for each compressed cache line, and a sectored memory, that is, memory is divided into 256-byte chunks. The sectors are referenced and used by storing a linked list of unused sector addresses, which is stored in the free sectors. As the nodes are used the space they were using can be used for data storage, therefore requiring no additional storage. If the compressed data is smaller than the STT's entry size, it can be stored in it, preserving sector entries. If not, one to four sectors are used, and pointers are stored in the STT. As there may be internal fragmentation within a compressed page, 1KB blocks can share a page if there is enough space, but only two at a time.

Because the cache is composed of relatively long cache lines (1KB), accesses are lengthy. In order to minimize stalls, the cache controller logically allows two cache lines to coexist in a physical line, allowing data to be written and read simultaneously. Besides, it divides the cache lines into 4 segments, so that only modified segments must be invalidated on eviction. This, however, can still generate a high memory traffic on read misses and evictions. MXT avoids overcommitting memory by listening to a memory-almost-full interrupt, zero-filling freed pages. Finally, in case a fast peak of memory utilization happens, a stalling process is created to hinder the other processes from overcommitting before the required swaps happen.

Robust

The robust main-memory compression scheme [ES05] tries to make the most of compression opportunities by increasing the flexibility of compressed lines and pages. In order to reduce the

complexity of recompaction, pages are divided into S sub-pages, each containing L lines. Both lines, sub-pages and pages can have multiple possible sizes. These sizes are cached along the TLB. The sub-divisions slightly lengthen the address translation logic. Furthermore, overflows and underflows must be handled in multiple granularities.

Zero Compression

As mentioned previously, zero blocks are frequent in the memory, so having a specific method to compress them in the main memory would be effective to reduce page faults and increase system performance. This is proposed by Dusser *et. al* [DS11] with their Decoupled Zero-Compressed (DZC) memory. The DZC memory is based on the Decoupled Sectored Cache [Sez94], treating pages as sets of sectors, and the Zero Content (ZC) Cache [DPS09]. It virtually splits the memory in two spaces: the uncompressed memory space (UP) and compressed memory space (CP), while using a compression controller to translate the UP addresses into CP addresses. Furthermore, it divides the memory in equal sized regions, called C-Spaces, each able to hold 64 to 512 pages.

Each page entry in the C-Space contains a null bit and a way-pointer for every memory block. Whenever a page is allocated in the C-Space, each memory block is checked, so that if it contains a null block, the corresponding entry's null bit is set, and there is no need to physically store the block. Otherwise, for non-null blocks, a way-pointer is stored, containing the allocated UP address of the page within the possible line position in the C-Space. Therefore, writes are trivial, except for the case where a null block is replaced by a non-null block. To deal with this case, each C-Space contains a set of validity bits to inform which entries are taken, so whenever this occurs the valid bits of possible lines are scanned to find a suitable position, and, if there is no free line, the whole physical page is moved to another available C-Space. If there is no available free space, an eviction process starts.

For easier management, the DZC memory stores redundant information, the number of free lines per set of lines, and the minimum of the free lines counters on each C-Space. Besides, as UP to CP address translation is required on every access, the authors propose using a translation cache (UCT) within the memory compression controller. By doing so, read accesses to null blocks can be served by the UCT, removing the need to access the slower memory chip. They also stated that there are more null blocks at the end of the pages than at the beginning, therefore they apply an exclusive-or to calculate set index and randomize distribution of null blocks across the sets.

LCP

Pekhimenko *et al.* [PSK+13] propose Linearly Compressed Pages (LCP), a memory compressor that enforces that every cache line within a given page fits in the same maximum compressed

size. LCP uses a small selection of possible maximum sizes to take the variability of the workloads' compressibility into account. Since often not all lines would be able to abide to this restriction, lines that compressed to sizes bigger than the page's are stored uncompressed in an exception area. A third area of the page is reserved for the page's lines' metadata, which stores information regarding the compressibility of a line, and, in the uncompressed case, where the line is located within the exception area.

It uses a metadata cache, and on misses to this cache lines are speculatively assumed to be compressed. To deal with systems using physically-tagged caches, the blocks' tags are extended with their offset within their corresponding page.Finally, the page table is adapted for compression with extra bits to inform the compressibility of pages and its adopted maximum line size, as well as bits to allow page addresses to be non-aligned with the virtual page's size.

Buri

Caches and TLBs are managed through an intermediate compression-enabled address space. The memory is divided into 16-MB chunks, each of which consisting of 1KB-blocks of data, as well as a linked list to locate free blocks. It uses a metadata cache. Pages have variable size, akin to LCP.

The OS is made aware of the physical space consumption through either an interrupt sent from the memory controller when a given threshold is reached, or by sporadically reading from a hardware flag that informs when memory is almost fully exhausted. The OS avoids overcomitting memory by preemptively performing page swaps in these situations.

MBZip

Kanakagiri *et al.* [KPM17] propose MBZip, which applies base-delta compression to multiple cache lines to avoid re-storing a base entry in order to compress multiple blocks into single entries. They suggest using different versions for the cache (MBZip-C) and main memory (MBZip-M). While the first adaptively uses conventional and SB tags to uniquely store data, MBZip-M is a bandwidth-focused memory compression scheme that duplicates blocks across multiple locations. As every block is also stored in its original location (*i.e.*, using conventional address computation logic), it keeps a 1 to 1 mapping of column addresses between conventional and the proposed memory pages, removing the need for special OS support. It uses metadata caches.

Figure 3.2 shows an example of data compressed using MBZip-M. Blocks b0, b1, b3, b4 and b5 can be co-compressed, and block b2 is compressible, but does not use the same encoding or base as the others. The valid bits of each entry are set accordingly, and data is duplicated between columns. If, for example, b5 is modified and cannot be compressed anymore, its valid bit in the metadata from columns b0, b1, b3, and b4 are invalidated, and the data is stored uncompressed in column c5.

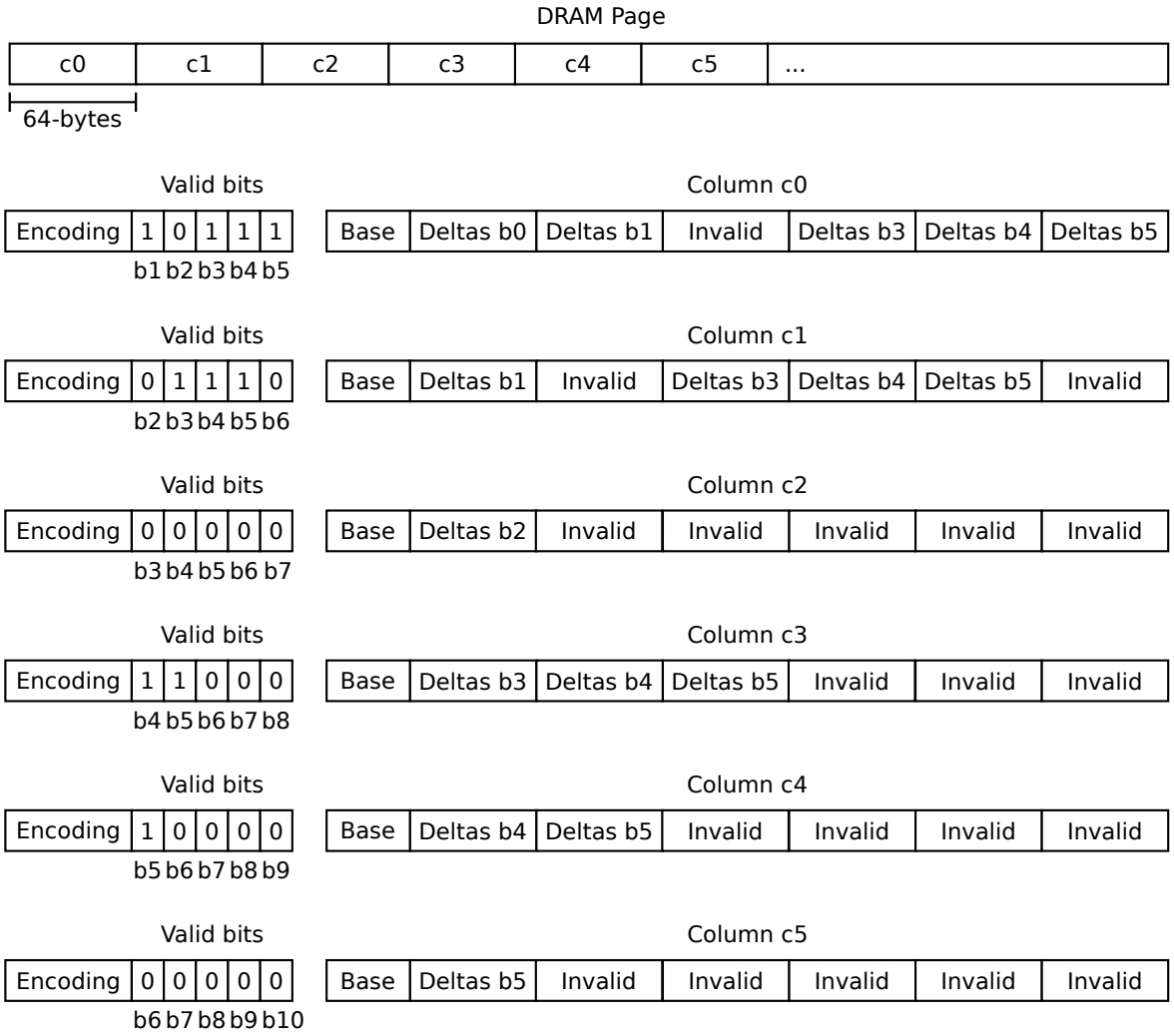


Figure 3.2 – Example of data compression using MBZip-M. Blocks b0 through b5 are compressible, but b2 uses a different base. The valid bits of each entry are set accordingly, and data is duplicated between columns.

Compresso

Expanding on the idea of extra accesses, Choukse *et. al* [CEA18] study the access increase due to metadata accesses, data expansions, and compressed entries that are stored across cache line boundaries. They propose Compresso, a series of optimizations to reduce data movement and access of capacity-focused memory compression layouts, while keeping the OS oblivious of compression.

In their system, compressed cache line sizes are made alignment-friendly, saturating counters are added to allow page overflow prediction (predicted are provided the maximum size), and

pages can be expanded by small chunks in case of insufficient space. They also keep track of the page sizes, and repack them when evicted from the metadata cache, if the advantages of doing so are relevant. This is needed due to the page overflow prediction mechanism, which may undermine compression potential in case of false positives.

Compresso manages to be transparent to the OS by applying the concept of balooning [Wal02], so that if the Operating System is unwittingly running out of memory, the Compresso driver will inflate and the OS will free its unused pages.

Attaché

Compression metadata requires a significant space, and even if it was cached in a metadata cache, this metadata would need to be stored and accessed, and thus extra memory accesses still need to occur on cache misses. In order to reduce this access overhead, Hong *et. al* propose Attaché [HNA+18], which stores the metadata in its respective block. By leveraging sub-ranked memory they manage to increase bandwidth without increasing latency.

Attaché is bandwidth-focused, so blocks always have a simple mapping, similar to an uncompressed memory. Consequently, OS support is not needed, since there is no special handling due to data expansion. The framework is composed of the Blended Metadata Engine (BLEM) and the Compression Predictor Unit (COPR).

BLEM transfers metadata to the block by making sure compressed blocks fit in a sub-rank, while reserving some of its space to prepend a Compression ID (CID) to inform the entry is compressed. If not compressed, however, both sub-ranks are used, and no CID nor compression bit is added. To avoid false positives, when the first X bits of an uncompressed entry match the compression ID, the X+1th bit, called Exclusive ID (XID), is set, and its original value is stored in a Replacement Area. An extra access is needed only in this low-probability case (A scrambling-descrambling mechanism guarantees data distribution uniformity).

In order not to reduce bandwidth, only one sub-rank is accessed when compressed, but both are needed for an uncompressed entry; therefore, COPR is used to avoid doubling cycles in the second case. It uses workload, per-page, and per-line granularity saturating counters, achieving 88% accuracy, on average.

Practical and Transparent Memory Compression

Developed concurrently with Attaché, Practical and Transparent Memory Compression (PTMC) [YKQ18, YKQ19] focuses on using location prediction and inline metadata to improve bandwidth while still using commodity memory modules and little to no OS support and metadata access overhead. PTMC, instead of keeping a per-block marker (CID), reserves 4 bytes per SB, and keeps a table for the unlikely case of uncompressed lines that conflict with the marker value.

Each block is either stored in its original location, or in one of its adjacent blocks. Figure 3.1 shows the possible block mappings for 4 consecutive blocks. Since the blocks have different possible locations based on their compressibility, a prediction mechanism, the Line Location Predictor (LLP), is used to determine where the block is likely located based on a cache of their previously known compressibility, with an accuracy of 98% at a tiny extra cost.

Inserting the blocks in their adjacent blocks, however, generates extra accesses due to data expansions and new co-allocations: if either a cache line does not co-allocate anymore, or starts being co-allocatable, it must be stored in a different location, and the previous one must be invalidated to avert using stall data when the LLP mispredicts. To further ease handling such events block writebacks are ganged with their adjacent blocks (*i.e.*, the whole SB is evicted at once), even if they are not marked for eviction, based on the assumption that they are likely old and would be soon evicted too, which may increase bandwidth consumption if the block was still needed, or it was clean.

To avoid having a huge impact when compression is not beneficial, they add a dynamic compression enabler, based on the number of extra evictions and invalidations caused by useless prefetches of co-compressible blocks, and accesses avoided due to useful ones.

3.2 Link Compression

The techniques discussed so far focus on increasing cache or memory capacity, reduce miss rates, and consequently lower bandwidth usage; however, compression techniques can also be applied to directly improve effective bandwidth, reduce transfers' power requirements, or narrow the width of buses. Such techniques are called *link compression* (or *bus compression*), and they work by reducing the amount of bits to be sent between memory levels. Notice that although the techniques presented here focus on data-bus transfers, similar ideas can be applied to both the address and instruction buses too [FP91, STD94, BDMM+98, XWL02]. A depiction of how some of these methods work will be given in the following paragraphs, and more details on specific energy-focused algorithms can be found in Mittal *et al.*'s survey [MN19]. By the end of this section a summary of link-compression algorithms is presented (Table 3.2).

A simple approach to reduce transfer size is to apply Significance-Width Compression (SWC): inform the number of relevant bits (*e.g.*, LSB) along with the relevant data bits themselves in order to take advantage of transfers of small values [TSS08]. It does not have high extra-hardware requirements; however, it highly relies on small-value locality. Ignoring LSB can also be used to save power in computations, reducing system dynamic power consumption by avoiding doing operations on insignificant bytes [CGS00].

Citron *et al.* [CR95] propose *Bus-Expander*, a component that compresses and decompresses data words between the ends of the buses (Figure 3.3). It works based on the principle of

Technique	Line Mapping	Page Size	Metadata Optimization	OS/Memory Controller Support
SCMS [LHK00]	A compression bit per pair of lines	2 compressed or 1 uncompressed	-	Not informed
MXT [TSW+01]	[1-4]:1 (Up to 64:1 for zero-filled pages)	Single	-	Controller sends interrupt → swap pages, zero-fill free pages, stalling processes
Robust [ES05]	L lines per sub-page, S sub-pages per page	Multiple	Cache (line, sub-page and page)	One pool of free pages per page size
DZC [DS11]	M:N	Single	Cache	Not informed
LCP [PSK+13]	N:1 (N depends on page size) + pointers for uncompressed lines	Multiple	Cache (line) that predicts compressed on misses	Page-compression-disable bit, scratchpad for page overflows
MemZip [STBD14]	1:1	Single	Cache (line)	Last DRAM row per page is re-mapped to an "overflow" page
Buri [ZLC+15]	Same as LCP	Multiple	Cache (line) with prefetching, adapted scheduling policy	Controller sends interrupt/raises flag → OS swaps pages
MBZip [KPM17]	[1-6]:1	Single	Cache (line and page)	Not required
Compresso [CEA18]	4 possible line sizes	Multiple	Cache	Not required (Uses virtual memory's ballooning)
Attaché [HNA+18]	1:1, 2:1	Single	In-data metadata, location predictor (line and page)	Not required
PTMC [YKQ19]	[1,2,4]:1	Single	In-data metadata, location predictor (line)	Not required

Table 3.1 – Summary of the DRAM compression techniques presented in this section. *Line Mapping* shows how lines are located within a page. The *Page Size* field informs the possible page sizes. *Metadata* contains where compression metadata is placed. *OS/Memory Controller Support* include the required modifications to support compression.

temporal locality by using a lookup table (LUT) (dictionary), and adding a compression bit to inform that only the low order bits of the uncompressed word are being sent. For every data transfer, the LUT is checked to see if there is an entry that is equal to the high order bits of the data. If it happens to be the case, its position is sent along with the low order bits, and the success bit is set; otherwise, the data must be transferred along multiple cycles. Although more than one cycle may be needed during transfer, this method allows a bus to be logically widened without significant performance loss on the average case. Their work is based on [FP91], which

uses a register file to compresses addresses, and can be applied to multi-core systems [KI03].

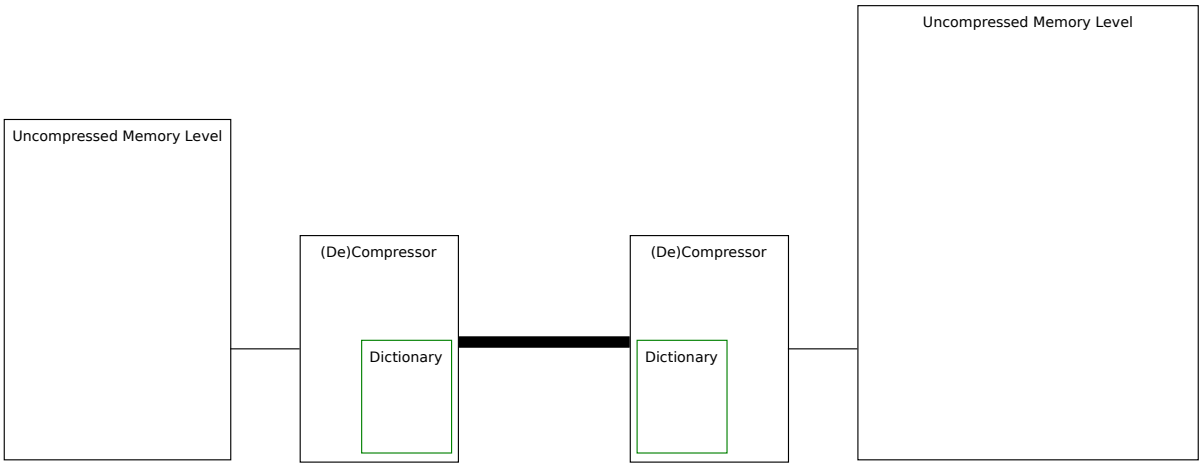


Figure 3.3 – Dictionary-based compression applied to link compression. The dictionaries of both sides of the bus contain the same entries at the same positions.

A similar approach is adopted by Yang *et al.* [YGZ04], where the whole word or bytes are stored in a table that keeps the most frequent values, and only the index bit (one-hot-code) is sent over the bus to save dynamic energy from bit flipping. This means that the table can have at most b entries, where b is the number of wires in the bus. They also propose extra techniques to reduce bit flipping, such as XOR-ing values from sequential different value transfers, and not allowing values that have low hamming distance from their one hot encoding to be stored in the table. Since multiple data types can be transferred, one can keep a different table per type to reduce negative interaction [ALYK12]. Partial dictionary matches can also be performed on the tables by embedding extra pattern information within the method's metadata [SAYN09]. Finally, Burtscher's FPC (BFPC) [BR10] mixes these approaches by matching the MSB of the table and sending the number of differing LSB. To reduces metadata overhead, instead of sending index bits, they predict which table entry to use.

Link compression schemes can also take advantage of coherence information. For instance, analogously to Bus-Expander, CAche-Based Link Encoder (CABLE) [NFW18] uses entire memories as LUTs: coherence bits determine which entries are present in both ends of the link and can be used to encode duplicated data. Since a memory-sized dictionary would greatly increase pointer overhead, they selectively pick a few entries from the memory that are likely to contain similar values to the line being transferred, in order to populate the dictionary.

One could also effectively store data in memories in compressed format, but not co-allocate them, in order to reduce the bandwidth usage. This can be done in memories whose accesses happen in small bursts: the less bursts need to be issued to fetch the whole data, the faster requests are serviced. Consequently, while in capacity-focused compression metadata is added

to locate blocks in the compressed memory, in bandwidth-focused compression its main purpose is to inform how many bursts are needed. This approach has been applied to both GPU [SSK12, VPJ+15] and CPU [STBD14] memories. In order not to waste the storage capacity saved by the compressed data, this idea can be combined with other techniques that demand extra storage overhead to reduce the negative impact of their footprint [STBD14]. For example, besides the data, each compressed cache line can contain metadata informing the compression algorithm used, Error-Correcting Code (ECC) bits, or Data Bus Inversion (DBI) [SB95] information.

Transferring compressed data has a higher per-bit entropy; furthermore, since values are compressed to different sizes, they lose their alignment, increasing the number of bit toggles (a bit switching from 0 to 1, or vice-versa). Toggling bit values has a direct impact on the energy spent on transfers; therefore, one could use metrics such as compression ratio, number of bit toggles, and bandwidth utilization to decide when to enable and disable link compression [PBV+16].

Technique	**Algorithm**	**Focus**
BFPC [AGN+15]	Prediction + Dictionary + XOR	T
Bus-Expander [CR95]	Dictionary	B
CABLE [NFW18]	Memory-sized Dictionary	T
Frequent Value Encoding [YGZ04]	Dictionary + XOR	E
Memory Compression [SSK12, STBD14, VPJ+15, QT17]	Compression scheme	T
SWC [TSS08]	Significance Width	B
TUBE [SAYN09]	Partial dictionary	E
VALVE [SAYN09]	Partial dictionary	E

Table 3.2 – Summary of the link compression techniques presented in this section. *Algorithm* is the data compression method used. The *Focus* field informs if the compression is focused on reducing energy/bit flipping (E), increasing throughput (T), or reducing bus width (B).

3.3 Memory-Hierarchy Compression

So far we have discussed techniques that isolatedly handle cache, and memory compression; however, these ideas can be applied simultaneously to create a fully compressed memory hierarchy. This allows greater performance improvement, energy savings or throughput increase than the isolated approaches. Such advantages can be of great significance to facilitate scaling of current chip multi-processors [RKB+09]. Although we will focus on the CPU's memory hierarchy, further expanding this concept to include device memories (*e.g.*, GPU) could also be beneficial.

There are, however, further challenges that must be tackled in a fully compressed memory hierarchy. For instance, each memory level has its own size and latency; thus, the impact of the decompression step is different on each level. Ideally, low-latency compressors would be the optimal choice on all memory levels; however, since there is correlation between compression

algorithm efficiency and latency, this would lead to compression opportunities being lost on memory levels at which higher decompression latencies are tolerated. One might be tempted to use a distinct per-level-cycle-efficient compressor, but this would mean that data being transferred from one level to another would not be fully compatible any longer.

Analogously, the granularity of each level's compressor's input data may generate a compatibility issue: one could envision a compression algorithm that works at a word granularity in the first-level cache, another that works at a cache-line granularity in the other cache levels, and one that compresses whole pages at the main-memory level. Moreover, besides the aforementioned effects on the compressor's latency and complexity (see Chapter 4), the input granularity further correlates with bus traffic and cache pollution: larger blocks bring more unrequested data to the caches, and increase bus traffic — in a manner akin to prefetching.

When connected memories have entirely compatible compressors, compressed data can be sent without the need to decompress, and received without the need to re-compress. Unfortunately, when the compatibility is reduced, extra decompression and compression steps must be taken to translate the transferred data. Furthermore, even if compressors are compatible, there must be a way to decompress data coming from direct memory accesses or bypassing responses.

Regarding compressed data being sent, the transfers could focus on bandwidth reduction, in which case only the requested data is sent; or compressed responses can include companion lines. In the latter case, compression would behave as a prefetcher; and, as such, would be subject to the advantages and disadvantages of prefetching. Decisions such as which lines to send, and when to send them become essential not to pollute the receiving cache. It is important to notice that while link compression applies compression to the data transfer, a compressed memory hierarchy transfers data that is already in compressed format from one compressed memory level to another. In addition, although link compression techniques can be used orthogonally, their usefulness is partly lessened because the compression opportunities are reduced due to the increased data entropy.

To summarize, some of the individual disadvantages of compression at each level are combined, and others are no longer applicable. Furthermore, new interactions apply, and a fully compressed memory hierarchy must be able to handle them seamlessly. Few works have studied memory-hierarchy compression, and they typically only compress the LLC, and main memory, transferring compressed data in between them [LHK00, HR05, DS11].

For instance, Hallnor et al. [HR05] make IIC-C [HR05] and MXT [TSW+01] compatible by having the same compression algorithm on both compressed levels, and matching the size of the cache line with the memory block's. Dusser *et al.* [DS11] integrated both the DZC memory [DS11] and the ZC Cache [DPS09] to create a zero-compressed memory hierarchy. Whenever a read request for a null block arrives from the LLC, the memory provides the demanded entry along with the information of all null blocks in the block's page, since it is lightweight (*e.g.*, 128

bits can represent all the null block info of a 8KB page). This zero block prefetching reduces miss ratio and avoids future memory traffic.

Chapter 4

Compression Algorithms

Cache compression schemes are typically simplified versions of conventional data compression algorithms. This is due to hardware complexity constraints, and the need for speed: since decompression is done on the critical execution path, it must be fast enough not to severely degrade the average hit latency [FRT96, PSM+12, SASW15]; therefore, cache compressors usually try to find a suitable trade-off between compression efficiency and decompression latency.

Another key difference between hardware and data compressors is the input size: hardware compression algorithms are typically applied at a cache line granularity — *e.g.*, all compression information is acquired from a chunk of 64 bytes [CYD+10, AW04b, PSM+12] — while conventional algorithms do not abide to this restriction; thus, the latter are able to achieve much better compression ratios due to higher ratio between deduplication opportunities and input size.

In this chapter we describe the evolution of hardware data compression, as well as what is the goal of each kind of solution. The algorithms depicted here generate better results for integer datasets, as they are usually designed for that reason, but compression can also be applied to code [EEF+97, BFG+03, SM08] and floating point data [LI06, BR10, ADS15]. We have opted not to enter into detail on these non-integer-based compression methods to focus on general data compression proposals.

4.1 Dictionary-Based Compression

In 1977, Lempel and Ziv proposed *LZ77* [ZL77], a dictionary based coder that replaces repeated occurrences of data by length-distance tuples. It works by parsing the data in a sliding window, storing, for each entry, the relative distance from the current input to its longest last seen copy, its size, and the next unmatched value — which we will represent by the tuple (*distance*,*length*,*value*). It is clear that the compression efficiency depends on the width of the distance and length fields, as well as the input size.

For example, given the 8-bit data 10110011_2, when parsing from left to right, LZ77 would compress the first couple of bits — 1_2 and 0_2 — as unseen bits $(0,0,1_2)$, $(0,0,0_2)$, since they hadn't been seen beforehand. However, when the third bit is parsed, the **dictionary** of known values will have already been populated with an occurrence of 1_2, from the first bit's compression (*i.e.*, a distance of two bits). This is a dictionary match, so the comparison is extended to the

fourth bit — *i.e.*, the value 11_2 is searched for in the dictionary. Since it does not exist, the length of the match is 1, and these two bits are compressed as the tuple $(2,1,1_2)$. Then, the fifth and sixth bits are compressed analogously, as $(3,1,0_2)$. Finally, the two last bits are parsed as a repeated occurrence starting at the third bit. In this simple example compression is actually disadvantageous, since the data input is too small for the algorithm's asymptotic efficiency; but, as the input size increases, better compression ratios are noticeable.

Based on the original algorithm, many variations have been proposed, such as adding a flag bit to indicate if a compressed tuples contains a pointer (*i.e.*, $length \neq 0$) or unmatched values (*LZSS*) [SS82]; using backward dictionary pointers (*LZ78*) [ZL78]; pre-initializing the dictionary (*LZW*) [Wel84]; and performing parallel comparisons [FRT96, TSW+01], which can then be implemented in hardware to be used in systems with compressed memories [CPR03, HR05, TSS08]. In particular, when porting this compression algorithm to be used in caches, its compressibility must be sacrificed to attain viable decompression latency and area overheads.

A hardware Lempel-Ziv compressor starts parsing input at a tiny granularity — *e.g.*, byte [TSW+01] — but, as long as there are dictionary matches, the granularity grows. Although using this variable-sized input allows reaching high compression ratios, the execution speed is quite low, since the algorithm's parallel-processing capabilities are limited by the design's budget. This, combined with the need to keep decompression latency low, has led proposals to concede compressibility in order to attain better decompression latencies. A straightforward way to simplify compression is to limit the granularity of the input: cache lines are then parsed in fixed-size **chunks**, and each chunk is individually compared against the dictionary entries to generate its respective compressed representation. In general, the dictionary entry's size matches the chunk size, but that is not always the case [TS19].

For instance, the top values of Figure 4.1 (in blue) represent how a hardware implementation of Lempel-Ziv whose window size can grow in multiples of bytes (*i.e.*, the chunk size is dynamic) parses an example input. The first chunks are small — *e.g.*, *0x03* and *0x2B* — as they are bytes that have not been seen previously, so their *distance* and *length* fields are 0. Later chunks can be larger, and inform where the repeating bytes are located (*e.g.*, the 3-byte value *0x032B00* in *0x032B0002* is repeated four bytes before, generating the tuple (4, 3, *0x91*)). In statically sized chunks the position of the repeating bytes can be extracted using a pointer. For example, when using 2-byte chunks (bottom values of Figure 4.1, in pink), value *0x032B* is seen first at the first chunk, so its second occurrence, at the third chunk, can be represented by the index 0 (tuple (1, 0)).

With the use of fixed-size chunks the tuple representation needs to change. First, the length field is fixed at the chunk size; thus, it is no longer required. Second, the value field is only needed when there is no dictionary match, and the distance field is not used unless there is a match; thus, both fields become conditionally optional, relying on the value of a new field —

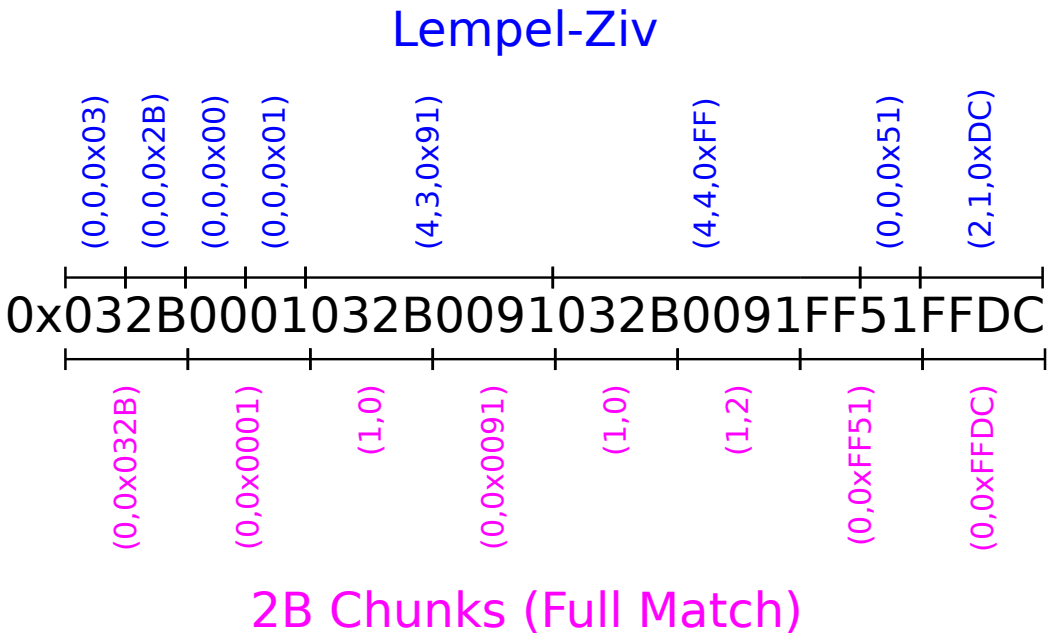


Figure 4.1 – Comparison of the generated tuples when using a Lempel-Ziv compressor and a compressor with a fixed granularity of 16 bits. Parsing is done from left to right.

the *pattern* field — which indicates if there is a match or not. Finally, the distance field can be renamed as the *pointer* field, since it contains the index of the dictionary entry to which the chunk refers. To summarize, the tuple becomes (*pattern*,*pointer*) for matches and (*pattern*,*value*) for non-matches. Figure 4.1 depicts how the tuple generation would differ when changing from a Lempel-Ziv compressor to a fixed-chunk-based compressor.

4.1.1 Adding Patterns

This change in granularity introduces an issue: in Lempel-Ziv the dictionary entries are compared seeking perfect matches, so even a single differing bit would make the matching fail. At a byte granularity this is a fair compromise; but, when the chunk size is larger, duplicating a whole chunk due to a couple of non-matching bits can severely restrain compressibility. This issue can be diminished by relaxing constraints to perform **partial matching**. Partial matching adds new patterns, which accept a few differing bits at predefined positions, duplicating only these said bits — *i.e.*, the tuple for matches becomes (*pattern*,*pointer*,*differingBits*). The number of bits is determined by the pattern granularity, and will be assumed to be 8 throughout this chapter, although other values can be used [WKS99, BBMM02].

For instance, assume that the symbol X means that the byte at that position of the chunk does not need to match the dictionary entry, and that M represents a match at that position.

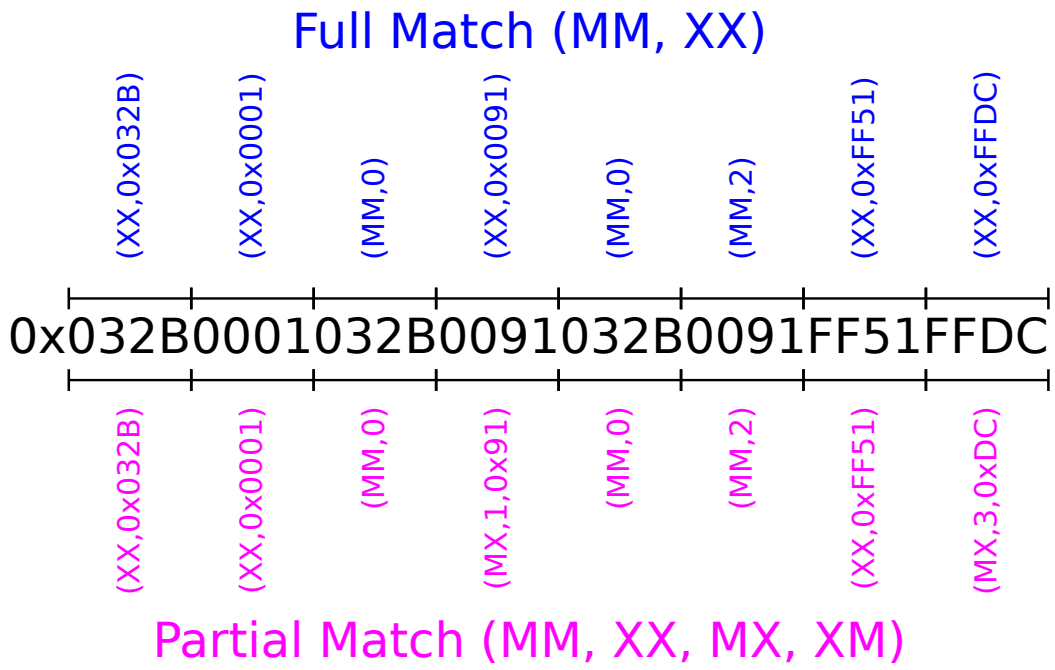


Figure 4.2 – Comparison of the generated tuples when using a compressor supporting only full 2-byte matches (patterns MM and XX), and a compressor with partial 2-byte matching (patterns MM, XX, and MX). Dictionary entries are not allocated for MM instances.

Then, a compressor that uses a 32-bit dictionary, and only allows perfect matches supports just the patterns MMMM and XXXX. One could, however, have different combinations of M and X bytes to perform partial matching and increase the likelihood of compression. In the previous example, the chunk *0x0091* would not be compressible, since it had not been previously seen; and thus it generates the tuple (XX, *0x0091*) under perfect-matching-only compression. When partial matching is supported — say, adding MX to the pattern list — that chunk would be compressible to (MX, 1, *0x91*) — *i.e.*, its last byte is different from the dictionary's entry at index 1, but the first byte matches (Figure 4.2).

With more patterns, the pattern field expands, incurring a larger per-chunk overhead. An alternative to cope with this extra cost is to use variable-length encoding to prioritize generating smaller pattern codes [KGJ96, NFBJ99, NFJB01, AYK01, CYD+10, NW15]. For example, X-Match and its derivations [KGJ96, NFBJ99, NFJB01, AYK01] predefine a set of patterns, based on the workloads' characteristics, using Huffman code [H+52] for the patterns' encodings. Since they also apply Phased-In Binary Codes (PBC) [Sal07] to the pointers, the dictionary entries are manipulated so that entries whose index generate smaller PBCs are more likely to occur. An analogous idea can be applied to reduce the pointer field size [KGJ96]. These solutions, however, make the compressor more complex, resulting in higher latencies.

It is worth noticing that not all possible patterns are useful; for example, a partial match on the most-significant bytes of a chunk (*e.g.*, patterns MMMX, MMXX, and MXXX) is significantly more likely to happen than other partial-matching patterns (Figure 4.3). Besides, when parsing a chunk, each dictionary entry is checked against the chunk using each of these patterns; thus, the hardware cost and complexity is higher, the bigger is the number of patterns. Consequently, it is desirable to limit the number of patterns to generate a cost-effective hardware implementation. This set of patterns is usually defined statically at design time, but compressors can also have a profiling stage to detect and select good workload-specific patterns [KGJ96, GHZ18].

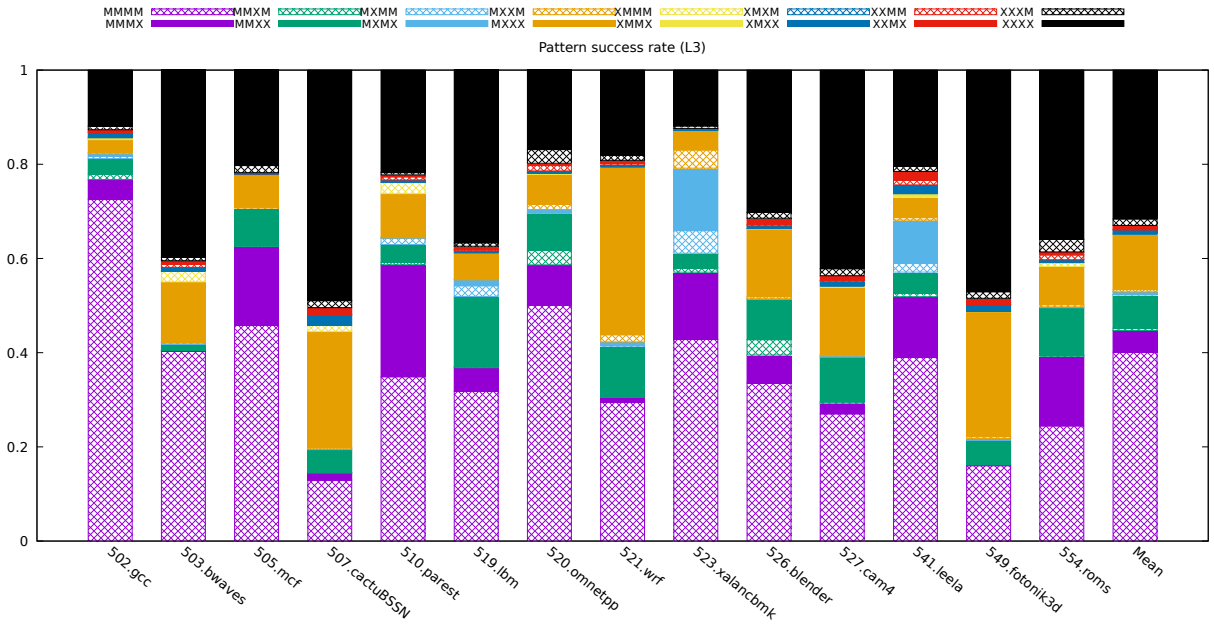


Figure 4.3 – Success rate of each of the possible patterns generated by the symbols M and X, at a 4-byte-chunk granularity, in a system with a compressed L3 cache. Results are shown for the SPEC 2017 benchmarks. Notice that most of the patterns are subsets of other patterns; thus, the success rate of such superset patterns is effectively higher than the values shown — *e.g.*, the effective success rate of MMMX is equal to the sum of the success rates of patterns MMMX and MMMM.

4.1.2 Adding Symbols

So far we have discussed comparing chunks strictly against dictionary entries; yet, there are patterns that are expected to be seen frequently and do not rely on the dictionary. For example, bytes containing only-zeros or only-ones are common, mainly among the most-significant bits [KGJ98, ES05, DPS09, TSS08]. Therefore, other symbols — such as a zeros byte (Z); a ones byte (F); a byte that is the sign extension of the bit that precedes it (S); or a byte that is repeated

throughout the whole chunk (R) — could be added to generate patterns with lower metadata footprint. This happens because a value that is implicitly defined by the use of a pattern code, does not need to use a dictionary-entry pointer. Furthermore, the first occurrence of a value does not necessarily imply in a full chunk duplication anymore (Figure 4.4).

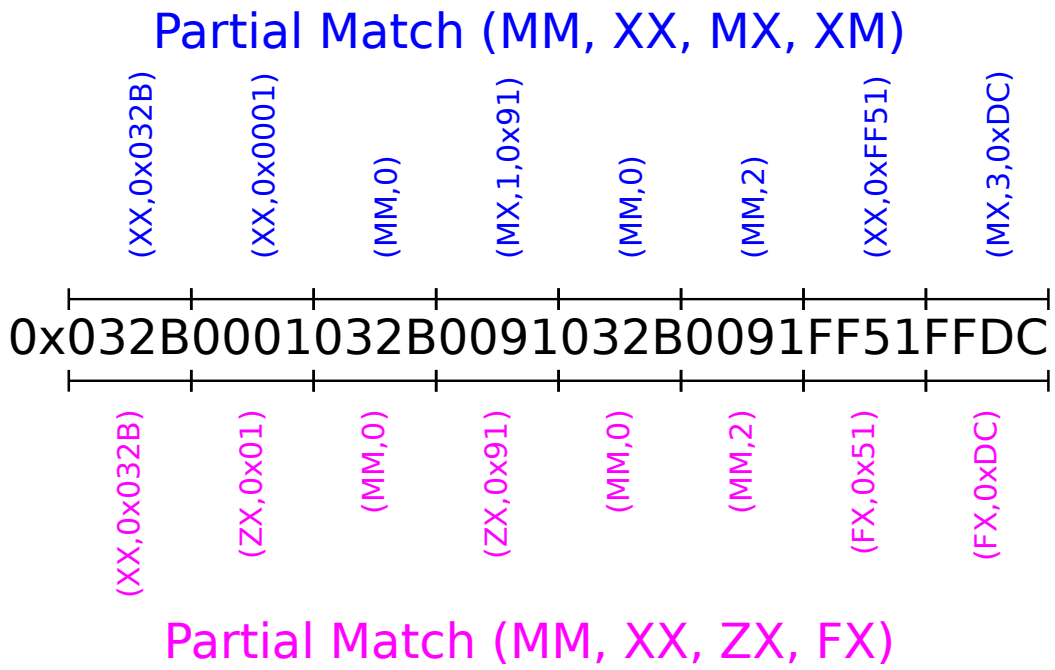


Figure 4.4 – Comparison of the tuples generated when using partial matching with symbols M and X (above, in blue), and partial matching with symbols M, X, Z and F (below, in pink). Each compressor has four patterns. Dictionary entries are not allocated for MM instances.

The symbols and patterns used in Frequent Pattern Compression (FPC) [AW04b], and Cache Packer (C-Pack) [CYD+10] are shown in Tables 4.1, and 4.2, respectively. *Code* indicates the code generated if the respective *Pattern* is found. *Data Size* is the block's size, in bits, after compression (without metadata). The last column presents the generated compressed output. The size of the position index is directly proportional to the logarithm of the dictionary size, *i.e.*, if the dictionary has 16 entries, the position index is 4 bits long.

4.1.3 Adjusting the Dictionary Size

As stated previously, a compressor's complexity is proportional to the number of patterns and dictionary entries; therefore, one could reduce the number of entries a chunk must be compared against to reduce its intricacy. For instance, Diff1 [BBMM02] selects the first word of a line as the sole dictionary entry, and calculates a single minimum delta d such that the $32 - d$ MSB of the dictionary entry matches all words' respective bits. Then, all other values are stored as d-bit

Code	Pattern	Data Size	Output (Code)Data
000	ZZZZ (xN)	3	$(000_2)N$
001	SSSsx	4	$(001_2)x$
010	SSSX	8	$(010_2)X$
011	SSXX	16	$(011_2)XX$
100	ZZXX	16	$(100_2)XX$
101	SXSX	16	$(101_2)XX$
110	RRRR	8	$(110_2)R$
111	XXXX	32	$(111_2)XXXX$

Table 4.1 – Pattern encoding for FPC. N is the number of consecutive zeroes, going up to 8. The lowercase variants of S and X indicate a 4-bit pattern match, instead of 8-bit.

Code	Pattern	Data Size	Output (Code)Data
00	ZZZZ	0	(00_2)
01	XXXX	32	$(01_2)XXXX$
10	MMMM	0	$(10_2)p$
1100	MMXX	16	$(1100_2)pXX$
1101	ZZZX	8	$(1101_2)X$
1110	MMMX	8	$(1110_2)pX$

Table 4.2 – Pattern encoding for C-Pack. p is the index of the position of the match.

differences relative to that dictionary entry. Since not all words will differ by the same number of bits, many difference bits will be duplicated; therefore, variants of this technique — Diff2 and Diff3 — assign a delta d_w to each word w to exchange metadata overhead for difference deduplication.

If both techniques are combined (limiting both the number of patterns and dictionary entries to a minimum), the compressor's complexity can be reduced to the point of attaining negligible decompression latencies. This approach is taken by the Base-Delta-Immediate (BDI) compressor [PSM+12], which enforces that the dictionary contains exactly one entry — the first chunk in the cache line — while only allowing two patterns — a match except last d bits, and an all-zeros value except last d bits. Since d is defined at design, the position of every chunk's data is fixed; therefore, the compressor becomes as simple as an adder tree, and a single-cycle decompression latency is achievable.

Although defining statically which chunk will be the dictionary significantly reduces hardware complexity, it also severely restrains compressibility. A solution that can achieve a good trade-off between reducing complexity and keeping a similar compression ratio is to reduce the dictionary size, yet make it dynamic. This is the case of Diff2 [BBMM02] and Frequent Pattern Compression with limited Dictionary support (FPC-D) [AA18], which limit comparison of a chunk, respectively, to its former or two previous chunks. This restrictive design works well due to the the spatial value locality of data: nearby chunks are likely to contain similar values [AS14a]. FPC-D also increases the number of symbols and embed the match location in the pattern code to increase efficacy and reduce the increase in complexity, respectively (Table 4.3).

4.1.4 Multiple Dictionaries

Sometimes data can repeat in patterns larger than the dictionary entry's size. This means that the dictionary is not able to capture the granularity of the data, and will likely not be able to

Code	Pattern	Data Size	Output (Code)Data
0000	ZZZZ	0	(0000_2)
0001	FFFF	0	(0001_2)
0010	MMMM (Prev.)	0	(0010_2)
0011	MMMM (Pen.)	0	(0011_2)
0100	ZZZX	8	$(0100_2)X$
0101	XZZZ	8	$(0101_2)X$
0110	RRRR	8	$(0110_2)R$
0111	MMMX (Prev.)	8	$(0111_2)X$
1000	MMMX (Pen.)	8	$(1000_2)X$
1001	ZZXX	16	$(1001_2)XX$
1010	ZXZX	16	$(1010_2)XX$
1011	FFXX	16	$(1011_2)XX$
1100	XXZZ	16	$(1100_2)XX$
1101	MMXX (Prev.)	16	$(1101_2)XX$
1110	MMXX (Pen.)	16	$(1110_2)XX$
1111	XXXX	32	$(1111_2)XXXX$

Table 4.3 – Pattern encoding for FPC-D. (Prev.) and (Pen.) indicate if it was a match with the previous or the penultimate dictionary entry, respectively (*i.e.*, there is no need to store a match pointer).

match (*i.e.*, generate more no-match patterns). Furthermore, smaller dictionaries parse the input in smaller chunks, which generate proportionally more metadata overhead than larger chunks. One could reduce this metadata overhead by simultaneously using multiple dictionaries, each with its own size (Figure 4.5). The compression efficiency is improved under this configuration, at the cost of a higher algorithmic complexity.

A variant of this idea is explored by Large-Block Encoding (LBE) [NW15], which has four dictionary entry sizes: 32, 64, 128 and 256 bits, covering the patterns shown in Table 4.4. To reduce the dictionary-related area, the entries of the larger dictionaries are represented as pointers to the smallest dictionary, adding an extra level of indirection to the decompression step. In the previous example, the 16-bit dictionary entry *0x0001* would be physically stored as two 4-bit pointers (to 8-bit dictionary's *0x00*, and *0x01*, respectively). Furthermore, to decrease (de)compression complexity and reduce the number of entries per dictionary, entries for the Y-bit dictionary are only added at Y-bit boundaries. That is, the entries *0x0102*, *0x0300*, and *0x0400* would not be generated, and the space required to store the maximum number of 16-bit dictionary entries needed would be halved, removing one bit from their pointer representation.

4.1.5 Sharing Dictionaries

So far the dictionary-based compressors presented apply compression to lines individually; however, just as values can be deduplicated within a line, different lines can contain similar values, so there is a potential advantage in sharing their dictionaries [NW15, PS16, ZJCP18,

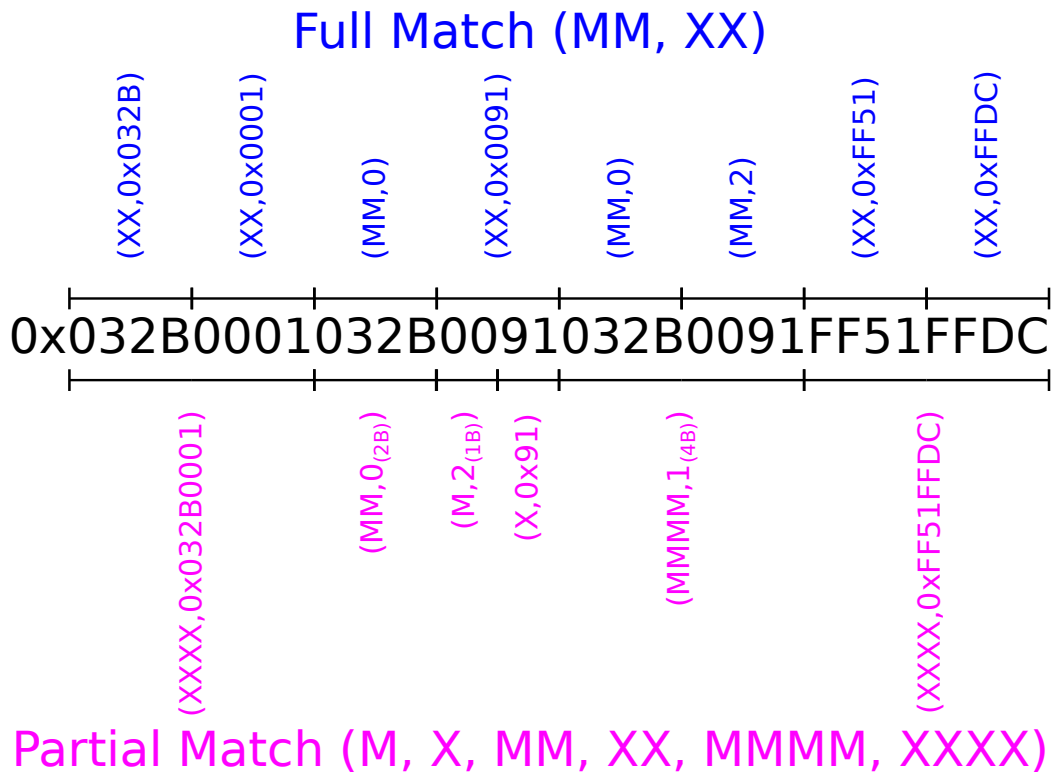


Figure 4.5 – Comparison of the tuples generated when using a single 2-byte dictionary (above, in blue), and three dictionaries (1, 2, and 4-byte) (below, in pink). The subscript of the index indicates which dictionary it refers to.

NFW18]. Whenever a dictionary is shared among multiple lines, the compressor's efficiency can be increased (more dictionary entries can be deduplicated). In addition, since the dictionaries are spanning multiple lines, they can be enlarged to contain more entries. This increases the probability of successfully compressing individual lines, at the cost of slightly larger dictionary pointers.

Compressors that share dictionary entries, however, increase the compressed cache's management complexity. In general, shared dictionaries are mostly static: updates should happen infrequently — *e.g.*, when the current dictionary does not abide to the workload's characteristics [AS14b] — since any modification to a dictionary entry may cause the update or eviction of multiple cache lines. This drawback can also be lessened by controlling the amount of lines that can share a dictionary — *e.g.*, a shared dictionary per data entry [PS16]. In this case, writes and overwrites, besides loading and comparing dictionary entries, also require the identification of the prospective dictionary; furthermore, accesses to a line are tightly coupled with other lines, which may increase the average hit latency, raise energy consumption, and overload read and write buffers.

Code	Pattern	Data Size	Output (Code)Data
00	XXXX	32	$(00_2)XXXX$
01	MMMM	0	$(01_2)p_{32}$
100	ZZXX	16	$(100_2)XX$
1010	ZZZZ	0	(1010_2)
1011	ZZZX	8	$(1011_2)X$
1100	$8{\cdot}M$	64	$(1100_2)p_{64}$
1101	$8{\cdot}Z$	0	(1101_2)
11100	$16{\cdot}M$	128	$(11100_2)p_{128}$
11101	$16{\cdot}Z$	0	(11101_2)
11110	$32{\cdot}M$	256	$(11110_2)p_{256}$
11111	$32{\cdot}Z$	0	(11111_2)

Table 4.4 – Pattern encoding for LBE. p_d is the index of the position of the match for the dictionary whose entries' size is d.

Shared dictionaries are generally used by profile-driven compressors [YZG00, BBMM02, AS14b]. These compressors fully separate the dictionary from the cache; the dictionary is stored in a global static table, and encompasses the majority of values expected to be seen on cache lines. To do so the compressor passes through a brief sampling stage, where the workloads' data is parsed to create a probabilistic model based on the frequency of values. This model then populates the table — which is frozen after sampling — and blocks are compressed by comparing the chunk's values to the table's entries. Since the pointers become larger due to the bigger dictionary, one can apply variable-length prefix coding methods — such as Huffman coding [H+52] — to reduce the metadata overhead of the most frequent values. Figure 4.6 exemplifies the difference between local dictionaries, 2-line dictionary sharing, and global static dictionaries.

In order to find the n most frequent values, Yang *et al.* [YZG00], for example, propose having a Content-Addressable Memory (CAM) [SGSB99] table with $2 \cdot n$ entries, each consisting of a value and a counter. During the sampling phase the counters are updated for each repeated value, and if the counter saturates, its respective value is swapped with the next entry. If the value was not present in the table, one of the entries with the lowest counter is replaced by it. At the end of this stage, the n first entries are used in the encoding. This allows a hardware-efficient way to achieve approximate sorting. In case the compression ratio becomes low, another sampling stage can be started concurrently using a second decompressor; therefore, data inside the cache may have two encodings. Nonetheless, re-sampling does not need to be frequent, due to the little variation in value locality over time — even in case multiple applications are being simultaneously executed; thus, it could be software-managed to reduce compression latency [AS14b].

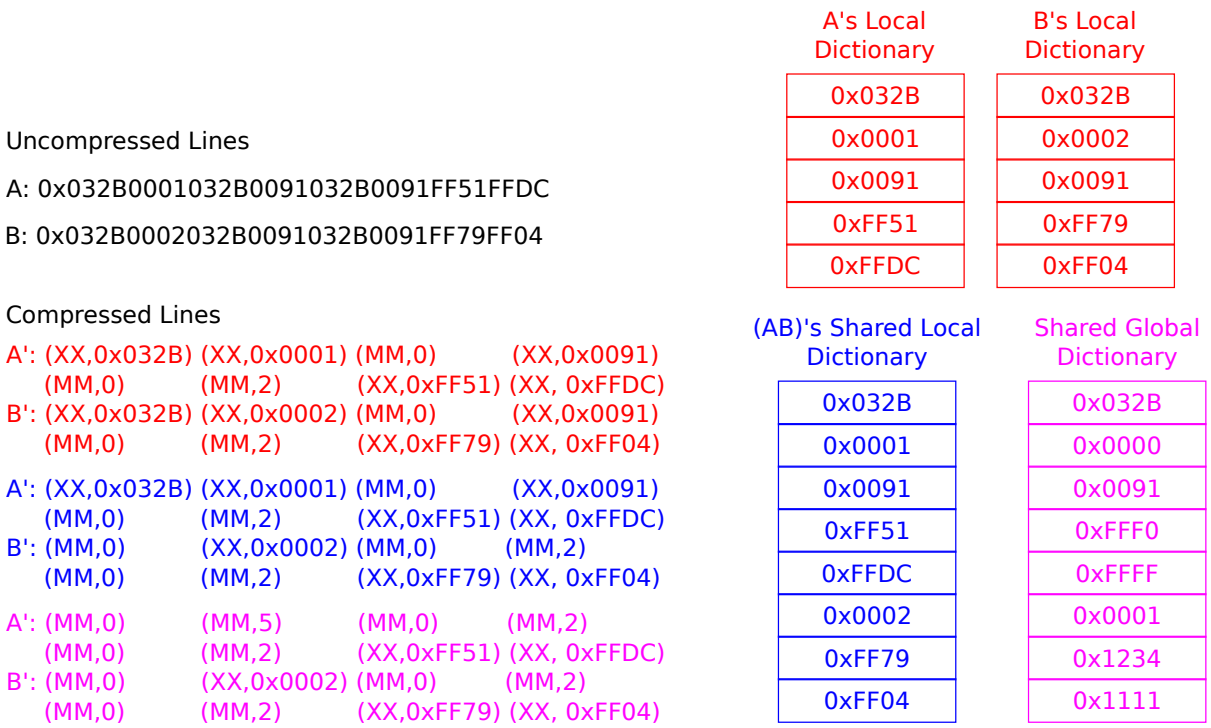


Figure 4.6 – Example of full match (MM and XX) compression using: 1) local dynamic dictionaries (top A'B' pair, in red); 2) shared local dynamic dictionaries (middle, blue); and 3) shared global static dictionaries (bottom, pink). The global dictionary was populated arbitrarily, simulating a previous sampling stage.

4.2 Other Techniques

Dictionary compression methods achieve a certain degree of compression, and are usually effective given their processing speeds; yet, they are not the only compression methods that exist — data compression has been a field of research for long [JTP21]. Compression can be as simple as bitwise opperations [YGZ04, AGN+15, KSCE16], or as complex as using mathematical transformations [JTP21]. Furthermore, a single technique does not necessarily achieve the minimum entropy on all workloads; thus, multiple layers of compression can be applied simultaneously to increase coverage. Needless to say, re-compressing the compressed data comes at a very high latency cost.

An example of multi-layer compression is Bit-Plane Compression (BPC) [KSCE16], which combines delta and pattern compression through sequential bitwise transformations to further increase the compression ratio. In BPC compression passes through 4 stages in order to achieve higher compression: Delta-BitPlane (DBP) transformation, which stores the first chunk as a base, and then calculates deltas between consecutive chunks (unlike BDI, which uses the difference between word and bases); bit plane rotation, which reassembles (isolates) the DBP bits based

on their position within the delta; then a XOR is applied between consecutive bit-planes; and lastly the bit-planes are scanned for pattern matching.

4.3 Multi-Compressors

Each compressor has its own merits and capabilities; thus, they can be combined to improve their individual shortcomings. These are **multi-compressors** — compressors composed of multiple sub-compressors — as depicted in Figure 4.7. Examples of multi-compressors in the literature include BDI [PSM+12], Hybrid Compression (HyComp) [ADS15], and Dictionary Sharing (DISH) [PS16].

Multi-compressors need extra bits to identify which of their sub-compressors is associated with some compressed data. These bits can be stored in either the tag or data entry. The former configuration requires a slight expansion of the tag array, but can be beneficial when performing sequential accesses, since decoding which compressor will decompress the data can be done in parallel with the tag and coherence checks. On the other hand, if the encoding metadata is stored in the data itself, or if the data access happens in parallel with the tag access, this decoding adds 1-2 cycles to the decompression step, which correspond to the time to decode and select the sub-compressor.

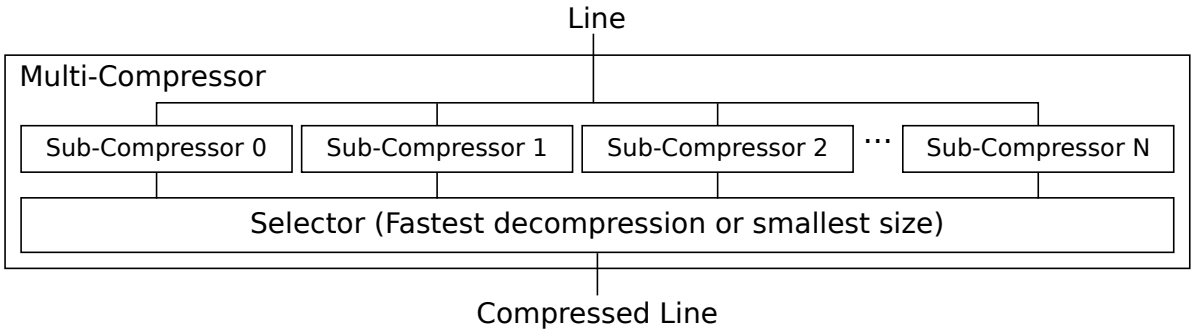


Figure 4.7 – Example of a multi-compressor consisting of N+1 sub-compressors.

4.4 Latency Trade-Offs

Caches typically display low hit latencies, and decompression adds to it; thus, the higher the decompression latency, the harder it is for the system to cope with it. The speed of a compression algorithm is highly correlated with its parallelizability; therefore, the higher the amount of chunks on which a given chunk's value can depend on, the higher is the ripple effect, and the costlier it is to parallelize its operations.

For instance, in C-Pack [CYD+10] a chunk can be represented as a reference to any of the

previously seen chunks (dictionary entries); in FPC-D [AA18] a chunk can reference one of the two most-recently seen chunks; and in base-delta compressors [PSM+12] any chunk can only refer to a single, statically determined (*i.e.*, the first), chunk of the line. The complexities of these algorithms are reflected in their speeds: by having the highest number of deduplication opportunities, C-Pack achieves high compressibility, yet its hardware is harder to parallelize; FPC-D exchanges some opportunities to simplify its implementation and increase its parallelizability; finally, a base-delta compressor's circuit can be as simple and fast as an adder tree.

4.5 Summary

Table 4.5 presents a summary of the state-of-the-art compression algorithms. All values assume a 64-byte cache line configuration, for uniformity. The latencies shown here are merely suggestive under the assumption of *a worst case scenario*, that the input data is available in its entirety when compression starts (*e.g.*, no delays due to bus width not matching line size), and the use of single data rate. Also, compressors are typically parallelizable, so the level of parallelization claimed in their original proposal is shown as a multiplier on the latency. In real implementations the levels of parallelization are possibly different, since they depend on the area budget, clock frequency of the (de)compression units, among other design decisions. Moreover, the latencies stated in bytes per cycle do not take into account the cycles needed to perform other pipeline operations on the compressed data, such as packing and shifting.

For multi-compressors, the values shown are for their respective sub-compressors. In general, it should be assumed that the average compression latency of a multi-compressor matches the latency of its slowest sub-compressor. In practice, some optimizations can be applied to stop execution early — *e.g.* if a sub-compressor that achieves the smallest possible size among the remaining sub-compressors is successful, the others can be halted. Finally, the decompression latency of multi-compressors only take into account the time to decompress the data of the sub-compressors. In reality, 1 or 2 extra cycles may be required to decode which sub-compressor to use.

Technique	**Dictionary Size**	**Entry Size (bits)**	**Num. Patterns**	**Compression**	**Decompression**
BDI [PSM+12]	0, 1, 2, 2, 2, 2, 2, 2	64, 64, 64, 64, 64, 32, 32, 16	1, 1, 2, 2, 2, 2, 2, 2	8, 8, 8, 8, 8, 4, 4, 2 bytes/cycle	$8 \cdot 8$, $8 \cdot 8$, $8 \cdot 8$, $8 \cdot 8$, $8 \cdot 8$, $16 \cdot 4$, $16 \cdot 4$, $32 \cdot 2$ bytes/cycle
BPC [KSCE16]	1	32	5	7 cycles	7 cycles
COCO [TS19]	At least 128 [a]	Variable [b]	Up to 2^{64}	8 bytes/cycle	8 bytes/cycle
C-Pack [CYD+10]	16	32	6	$2 \cdot 4$ bytes/cycle	$2 \cdot 4$ bytes/cycle
DFPC [GHZ18]	0	32	4 (+ 4 dynamic)	$16 \cdot 4$ bytes/cycle	$16 \cdot 4$ bytes/cycle
Diff1/2/3 [BBMM02]	1/1/8	32	32	$16 \cdot 4$ bytes/cycle	$16 \cdot 4$ bytes/cycle
DISH [PS16]	4, 8 (shared)	32	1	4, 4 bytes/cycle	$16 \cdot 4$, $16 \cdot 4$ bytes/cycle
FPC [AW04b]	0	32	8	3 cycles	5 cycles
FPC-D [AA18]	2	32	16	$4 \cdot 4$ bytes/cycle	$8 \cdot 4$ bytes/cycle
Huffman [AS14b]	512/1024 (single/multi-core) (shared)	32	2	6 cycles	14 cycles
LBE [NW15]	128 (shared)	32	5, 3, 3, 3	4, 8, 16, 32 bytes/cycle	4, 8, 16, 32 bytes/cycle
Lempel-Ziv [FRT96, TSW+01]	$4 \cdot 16$	32	2	$4 \cdot 1$ byte/cycle	$4 \cdot 2$ bytes/cycle
X-Match,X-RL [KGJ96]	16	32	10–16 [c]	4 bytes/cycle	4 bytes/cycle

a. Assuming the use of a 8KB dictionary, and a maximum object size of 64B.
b. COCO works on objects, not basic data types.
c. Depends on the probabilities of the patterns in the set that generates the Huffman codes.

Table 4.5 – Summary of data cache compressors.

Chapter 5

Efficiently Dealing with Compressed Blocks

Cache compressors work as black boxes: data enters in uncompressed format, it is processed, and the compressed data is output. However, compression by itself is not enough to improve either system performance, or cache capacity; a *compaction scheme* (or *cache organization*, or *compactor*) must be used to determine what to do with the compressed data. That is, compaction schemes expand the capabilities of conventional tag-data mapping methods to account for compressed blocks and their ability to share data entries.

Consequently, besides deciding on which data entry a block should be allocated, compactors have extra responsibilities (as described in detail in Chapter 2): to co-allocate blocks based on their compressed size; to determine whether the level of compression achieved by a given compressed block is satisfactory, or if the compression results should be discarded, and the block should be stored in uncompressed format; and to determine where a block should be placed within a data entry.

The main reason compression results would be discarded is that the probability of co-allocation follows a Gaussian function centered around 50% of the cache line's size: a block that has been compressed to 99% of the cache line's size can only co-allocate with blocks that have been compressed to 1% of their original size, but a block that has been compressed to 90% of its size can co-allocate with the whole range of sizes from 1 to 10%. If a line has low probability to co-allocate, it has a high chance to be stored alone, and the extra decompression step required to read it would be detrimental to the system's performance.

Another reason to dispose of compressed lines is a restriction on the metadata budget. The size of a block must be available, either explicitly stored or obtainable through some processing. Therefore, some metadata bits must be added with the relevant size information — the *size field*. If there are no limits on the size field representation, blocks can be compressed to any size [AW04a, CYD+10, SW13] — we will refer to this concept as **unconstrained** methods. Although not imposing restrictions on the size allows compression to reach its full potential, these methods significantly increase the metadata overhead due to the number of bits needed to represent the compressed size. Besides, locating cache lines becomes non-trivial: lines can be

placed anywhere in the data entry. This results in a few more cycles being added to the access path to find the exact block location.

To cope with those drawbacks, some compaction techniques limit compression to fixed sizes (*e.g.*, 25% and 50% of the line size) [SSW14, SSW16]. These **constrained** methods have low metadata overhead, at the cost of limiting co-allocation opportunities. This is due to the fact that although cache compressors may successfully compress some workload regions, there is still plenty of data that fails to attain favorable compressed sizes for compaction. For instance, The average compressed line size in SPEC 2017 for multiple state-of-the-art compressors [PSM+12, KSCE16, CYD+10, AW04b, AA18, AS14b, KGJ96], is still far above 50% of the cache line size, making it hard to effectively co-allocate blocks with the limitations of these overly restrictive techniques (Figure 5.1).

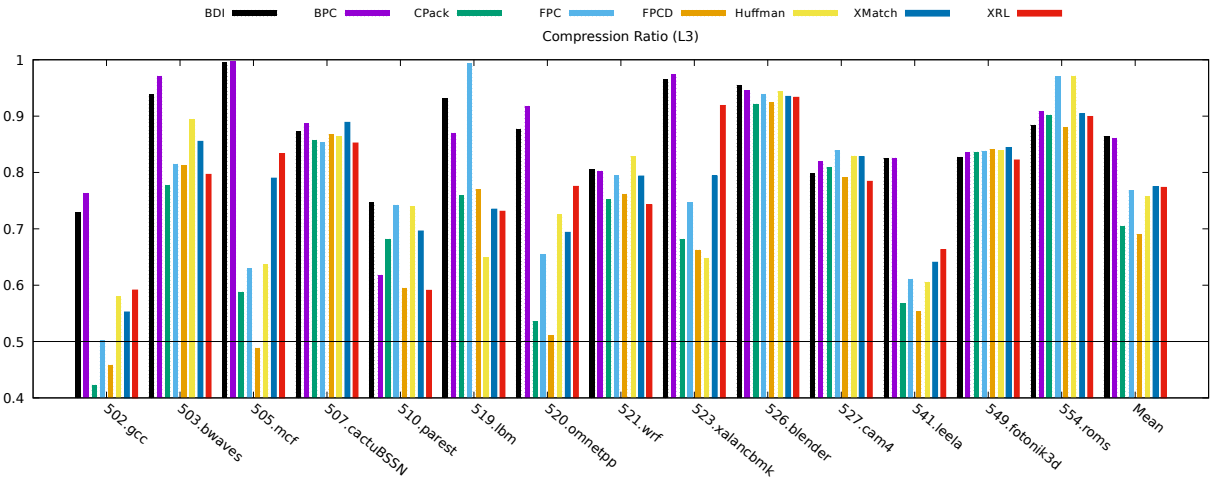


Figure 5.1 – Average compression ratio of SPEC 2017 workloads for multiple state-of-the-art cache compression methods applied to the LLC.

To achieve the best trade-off between limiting the number of possible sizes and having an unconstrained representation, we have come up with **Pairwise Space Sharing** (**PSS**). PSS stores sizes in a partially-constrained fashion: *blocks are grouped in pairs, and although each pair must fit in a fixed-size entry, restrictions on the sizes of the blocks within a pair are lessened.* Furthermore, PSS can be applied in conjunction with most state-of-the-art cache compaction proposals. This means that PSS makes the most out of co-allocation opportunities, and requires far fewer metadata bits than conventional unconstrained methods. Moreover, *due to the design decision of coupling blocks in pairs, line location within a data entry is made trivial*, and no extra cycles are required. The following sections describe how it operates in details, and the compaction improvements that can be achieved with it.

5.1 Pairwise Space Sharing

5.1.1 Block placement

One of the main issues of unconstrained cache compaction techniques solved by PSS is block placement within a data entry. In compressed caches, blocks are assigned to the next available location, if any; however, since each block can be compressed to any size, this location can be anywhere in the data entry. As a result, unconstrained methods typically use pointers or surrounding sub-blocks' sizes to inform where a block has been placed [AW04a, SW13, HR04]; therefore, under this configuration, locating any compressed block is not trivial, and likely increases the access latency in a realistic compressed cache layout. *Pairwise Space Sharing solves the placement issue by splitting the data entry into multiple constrained segments, and then applying an unconstrained layer on each.*

The idea is that smaller constrained entries restrict placement possibility. For example, a 512-bit data entry can be divided into four 128-bit segments. Each segment can then co-allocate blocks without constraints, as long as they fit in its 16-byte space, as depicted in Figure 5.2. If a rule is applied so that a given block B can only be assigned to segment S, then $\frac{3}{4}$ of the placement locations are removed from the possibilities. Nonetheless, this restriction is not enough to satisfy latency requirements, because B can still be stored anywhere within its segment, for a wide range of placement possibilities.

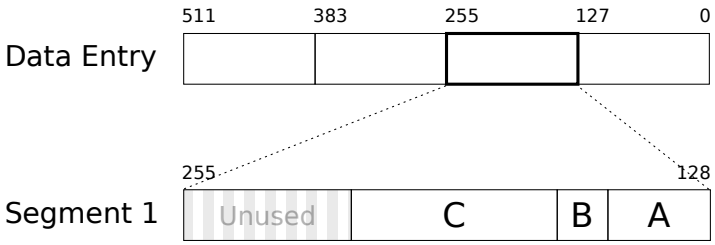
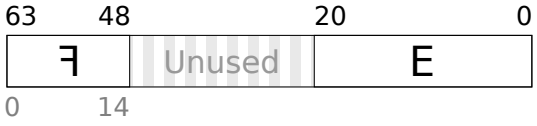


Figure 5.2 – A data entry can be split into multiple segments whose size is fixed. Then, within each segment, block co-allocation can be done in either a constrained or unconstrained fashion. In this example segment 1 is using unconstrained co-allocation, and each of the blocks it contains is compressed to a different size.

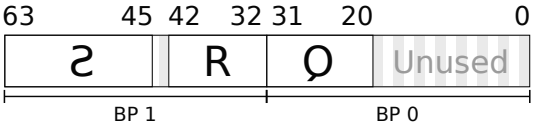
There are, however, two special cases that deserve distinct attention: when the number of blocks allowed per segment are 1 and 2. When only one block can be allocated per segment, there are two possibilities: either it is an uncompressed cache (the segment size is 64 bytes); or it is the general case of a constrained method — the segment size is smaller than 64 bytes, and compressed blocks must fit in fixed-sized entries.

The other case, when there are up to two blocks per segment, has a peculiarity that can be exploited to greatly simplify locating blocks. Within a constrained entry, no matter its size, there

are two invariable locations: its leftmost bit, and its rightmost bit (*i.e.*, the extremities). These can be used as markers that define the beginning of a sub-block, with one of the sub-blocks being stored in reverse order (the MSB becomes the LSB and vice-versa) (Figure 5.3a). Another advantage of these fixed extreme locations is that, since the bits in between the sub-blocks are unused, *data contractions and expansions that still fit in the pair do not need recompaction.* We will refer to segments that contain up to two blocks as a **block pair** (**BP**).



(a) Block placement in an entry containing a single BP. E is stored conventionally, and F is stored with its bits reversed.



(b) A data entry with two segments supports up to four sub-blocks — there are two BPs. R and S are paired up in a BP 1, and Q is in BP 0. Q's companion is not present. Q and S are stored reversed.

Figure 5.3 – Overview of block pair sub-block placement. Each sub-block is stored relative to an extremity of its block pair.

5.1.2 Size Representation

The size information of each block is an inherent metadata overhead — it is need to determine whether a block can co-allocate with others, so it must be available either explicitly or implicitly. As discussed previously, compactors represent sizes either in constrained or unconstrained format. Having unconstrained sizes means that the compressed size is represented in full [1], so compression can be used to its maximum, and possibly no data space will be wasted [AW04a, CYD+10, SW13]. On the other hand, constrained sizes (*e.g.*, half or quarter a cache-line size) require less metadata bits at a small compression efficiency penalty: only a couple of compressed sizes are representable, so the compressed data must be padded to fit in one of those possible sizes [SSW14, SSW16].

PSS, however, changes the size of the constrained section from a data entry's size to smaller containers (the block pairs). As a matter of fact, the size of a block pair is fixed, but dependent of the compression factor (CF), as shown in Equation 5.1; thus, as opposed to prior proposals [CYD+10], this idea can be applied to a compressed cache that allows more than two compressed blocks per data entry. For instance, in YACC [SSW16] a superblock's compression factor defines the minimum size to which a sub-block must be compressed in order to be able to co-allocate: a quarter of the data entry for a compression factor of 4, and half a data entry for a compression

1. There is a small caveat for unconstrained compression: since the benefits of representing sizes in bits is minimal compared to bytes, these methods typically employ a byte-granularity representation to avoid the cost of those extra 3-bits.

factor of 2. With PSS this limitation is lessened, and the ***pair**'s size must fit in a half or a whole data entry instead*, respectively.

$$BPSize_{CF} = 2 \cdot \frac{cacheLineSize}{CF} \tag{5.1}$$

Under this configuration, no further modifications are required, and sub-blocks are paired like in the PSS-less YACC: if the compression factor is 2, there is only one block pair, and any of its four sub-blocks can be paired in it; and if the compression factor is 4, there are at most two BP, and each sub-block has a determined position in the data block — sub-block 0 can only be paired with 1, and sub-block 2 only with 3. Considering that converting between compression factors would increase the placement complexity — due to the fact that one compression factor allows any sub-blocks to be paired up, and the other imposes strict positioning — our design enforces that a superblock must commit to a compression factor until all its sub-blocks are evicted. Figure 5.3b shows an example of a data entry containing more than one block pair.

Optimizing The Size

From Equation 5.1, and assuming a cache line of 64B, a superblock with a compression factor of 2 contains a single BP, whose size field is naively represented as $2 \cdot 6$ bits; and a superblock with a compression factor of 4 shelters two BPs ($4 \cdot 5$ bits). Since the size field must have enough space for the worst-case scenario, 20 bits would be need to be reserved per data entry. Under this scenario any size is valid, from zero bytes to the number of bytes of an uncompressed cache line.

However, the probability distribution of compressed sizes follow a non-uniform cumulative distribution function: barely compressing a block is significantly more frequent than compressing it to a tiny size (Figure 5.1). This means that the likelihood of co-allocating a block is inversely proportional to its compressed size; thus, a block that has been compressed to a large size will likely not co-allocate, imposing an unnecessary decompression latency fee on hits. This remark is particularly distinguishable for compaction methods that use a superblock tag representation, since neighbor blocks tend to present similar compressibilities [PSM+12, SW17]. Consequently, the neighbors of a block whose size is exceedingly large will probably have similar sizes, which means they would not be able to co-allocate.

Hence, seeing that having large compressed blocks is likely not beneficial, one can impose an upper ($maxSize_{CF}$) and lower ($minSize_{CF} = BPSize_{CF} - maxSize_{CF}$) limits on the compression size to reduce the number of bits needed to represent a valid size (Equation 5.2 if $2 \cdot maxSize_{CF} \neq BPSize_{CF}$; 0 otherwise). Consequently, all sizes are stored as a number relative to the lower limit on the compression size. In the previous example, if the maximum compressed size allowed is 62.5% of the uncompressed line ($maxSize_2 = 40B$, $minSize_2 = 24B$,

$maxSize_4 = 20B$, and $minSize_4 = 12B$), the number of bits needed per size entry changes to $sizeBits_2 = 4$, and $sizeBits_4 = 3$. Therefore, for instance, an absolute size of 30B would be stored as a relative size of 6B (0110_2).

$$sizeBits_{CF} = \log_2 (2 \cdot maxSize_{CF} - BPSize_{CF}) \tag{5.2}$$

Another optimization can be done to reduce the number of size field entries. Since both the segment's size and location are known, *only one of the sub-blocks' sizes needs to be stored in the tags, in the pair's respective size field entry*, and the other (e.g., the non-reversed sub-block's) is implicitly defined as its complement. If only the non-reversed block is present in the pair, the stored size represents the available space for the reversed sub-block.

Total Overhead

A compactor using Pairwise Space Sharing needs — besides the usual tag, replacement state and coherence fields — to dispose, per data entry, of $\log_2 (maxCF)$ bits to inform the number of block pairs in the data entry (substitutes the conventional compressibility state); and $\log_2 (\frac{maxCF}{2})$ size field entries to bear the size of the smallest possible block pair entry ($sizeBits_{maxCF}$). For instance, the case of PSS where the maximum compressed size allowed is 50% is equivalent to constrained methods allowing two possible sizes — 25% and 50% — such as YACC [SSW16] and SCC [SSW14]: $BPSize_2 = 32B$, $BPSize_4 = 16B$, $maxSize_2 = 32B$, $maxSize_4 = 16B$, and $sizeBits_2 = sizeBits_4 = 0$ bits.

5.2 Methodology

Our simulations have been performed using gem5 [BBB+11], a software capable of performing the required steps for instruction emulation, at the cost of higher simulation times than other popular yet less accurate simulators [CHE11, SK13]. Compression-related statistics are averaged across all (de)compression occurrences. Compaction-related statistics are calculated by averaging snapshots of the contents of the cache, which are taken every 100 thousand simulation ticks.

To test the behaviour of the analysed techniques we took multiple checkpoints per benchmark of the Standard Performance Evaluation Corporation (SPEC) 2017 benchmark suite using SimPoints [SPHC02]. The average of each benchmark's statistics has been calculated with the arithmetic mean of its checkpoints, and the total geometric mean of the benchmarks was normalized to a non-compressed baseline system. Benchmarks whose number of MPKI was lower than 1 were discarded from the analysis, since these barely benefit from having larger caches — in these cases, a compressed cache would not be useful.

Although appropriate for studying compression techniques, Sardashti and Wood claim that

using conventional SimPoints does not fully represent the compression properties of real workloads [SW17]. They propose using toolsets to study memory and cache data on real machines. These toolsets work by periodically stopping a workload to take snapshots and analyze the memory contents, and forcing TLB flushes and finding the address that causes the next TLB miss in the cache. They also notice that compression ratio significantly varies over time. To conform to this scenario we execute workloads for long periods of time to get more accurate results; therefore, after a warm-up of 100M instructions, the workloads are executed for 200M instructions.

The baseline model performs out-of-order (OOO) execution, and is detailed in Table 5.1. All compression and compaction algorithms are applied to the L3 on top of this common configuration.

Processor	1 core, OOO, 8-issue
Cache line size	64B
L1 I/D	32KB, 4-ways, 4 cycles, LRU
L2	256KB, 8-ways, 12 cycles, RRIP
Shared L3	1, 8-ways, 34 cycles, RRIP
MSHRs and write buffers	64
DRAM	DDR4 2400MHz 17-17-17, tRFC=350ns, 4GB
Architecture	ARM 64 bits
Clock	4GHz
Image	Ubuntu Trusty, Little Endian

Table 5.1 – Baseline system configuration.

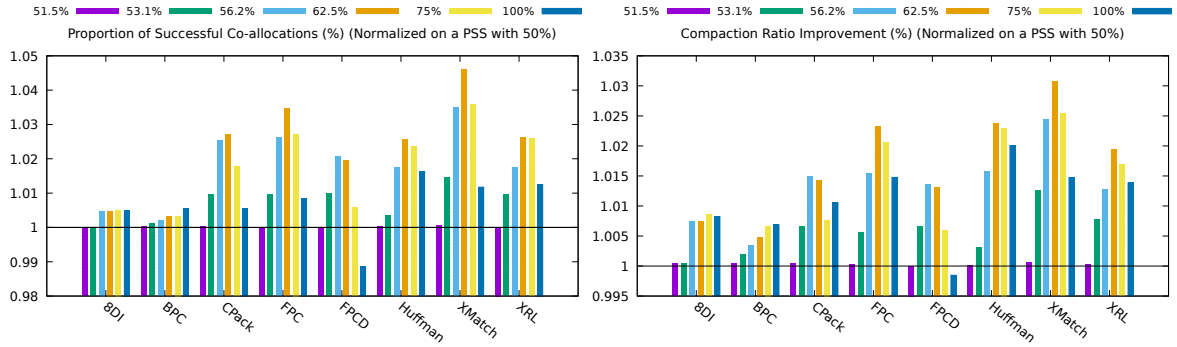
5.2.1 Results

As mentioned previously, the compressibility of neighbor blocks tends to be similar, due to the fact that they usually present similar data contents. Consequently, when using block pairs, the companion of a block that compresses to a size greater than 50% of the block pair's size has a high likelihood of compressing to a size greater than 50% too; and the probability of being able to co-allocate them is lessened. Thus, modifying the restrictions to discard less compressions may actually be detrimental to the system's performance.

We have conducted an analysis of the proportion of blocks that were able to co-allocate with another block at the moment it was compressed to determine the optimal number of bits to be used in the size field. As it can be seen in Figure 5.4a, the lowest ratio of unsuccessful co-allocations is achieved when block sizes are within the absolute range [37.5% : 62.5%] of the block pair's size. This behavior is reflected in compaction ratio improvements: Figure 5.4b measures by how much the compaction ratio is improved when compared to the respective baseline PSS

with a maximum compressed size of 50%.

Finally, Figure 5.5 shows the difference in compaction ratio when using multiple state-of-the-art compressors while coupling YACC [SSW16] with PSS using a range of [37.5%: 62.5%]. All configurations using PSS outdo their non-PSS counterpart. Note that although Pairwise Space Sharing has only been applied to a superblock-based compaction layout, it can be applied to non-superblock-based layouts too. This is due to the fact that it decides *where* and *how* blocks are allocated in a data entry, not *which* blocks.



(a) Normalized ratio of successful co-allocations ($\frac{numCoAllocations}{numCompressions}$).

(b) Normalized compaction ratio.

Figure 5.4 – Comparison of the best range choice for multiple state of the art compression methods under a YACC layout with PSS. Values for each compressor are normalized to the respective compressor using a PSS of 50%.

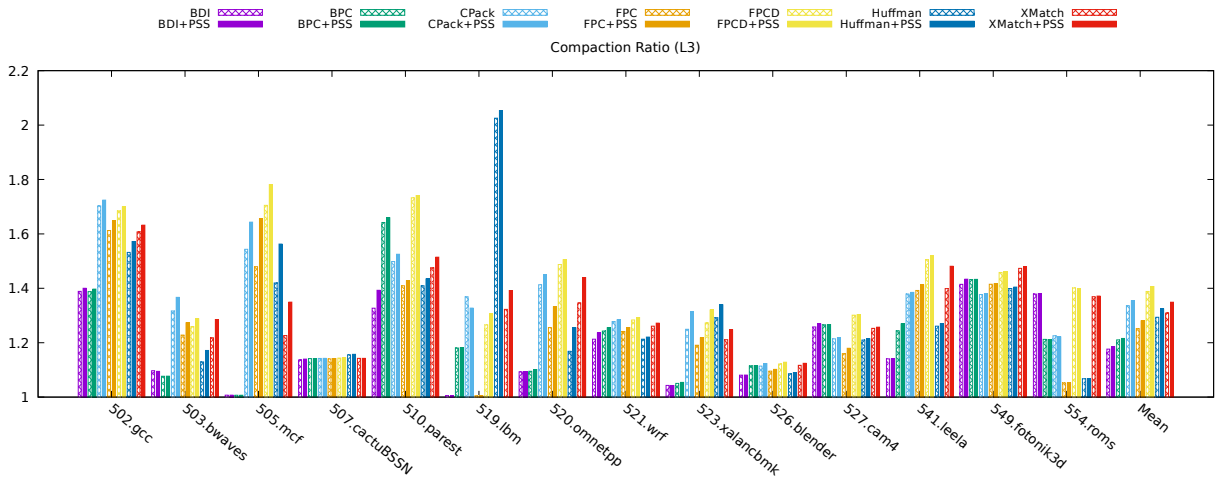


Figure 5.5 – Compaction ratio of multiple state of the art compression methods under a YACC layout without and with PSS applied to them.

5.2.2 Effects of a Single Size per Pair

As discussed earlier, one possible optimization is to represent only one of the pair's sub-block's sizes. This halves the number of bits needed for the size representation; however, it has a drawback in case of a data expansion of a block whose size is stored: since the exact compressed size of the non-reversed block is not known, its data must be read and compressed again to find out whether an eviction will be needed.

Nonetheless, this event is rare, occurs out of the critical path, and the re-compression step can be removed by adding a delimiter bit to the end of the non-reversed block's compressed data. To quantify this side effect, we have simulated a worst-case scenario where the latency of a read was added to every block overwrite. The differences in IPC and compaction ratio were far below 1%.

5.2.3 Comparison with Pair-Matching

Chen *et al.* introduced an idea with concepts similar to Pairwise Space Sharing, called *pair-matching* [CYD+10]. Although both techniques group blocks in pairs, there are multiple advantages of using PSS over pair-matching. For starters, PSS cleverly positions blocks at the extremities of the block pair, making locating blocks trivial. This also simplifies data expansion and contraction, since the unused bits are always located in between the blocks, so there is never a need to re-locate/re-compact data within a block pair.

Pair-matching pairs blocks such that the average of their compressed sizes must fit in half a cache line. As depicted in Figure 5.4a, this is sub-optimal, and is improved by PSS's probabilistic analysis of co-allocation. Moreover, this analysis, along with the removal of the partially redundant companion's size information, allows to greatly decrease metadata overhead: the number of bits needed to portray the compressibility state and size is reduced from $1 + 2 \cdot 7 = 15$ to $1+1 \cdot 4 = 5$, and from $2*(1+2 \cdot 7) = 30$ to $2+2 \cdot 3 = 8$, for systems with a maximum compression factor of, respectively, 2 and 4.

Chapter 6

Granularity Exploration

As seen in Chapter 2, cache compressors typically rely heavily on the temporal and spatial localities of data: they expect that values will be seen more than once, either with a few differing bits, or as perfect duplicates. Consequently, there is a predominance of dictionary-based pattern compressors — these use the first occurrence of values to populate a dictionary of reference values. When parsing cache lines, values are first compared against the dictionary entries using predefined patterns to check if they can be deduplicated by storing the parsed value as a full or partial reference to an entry (*value deduplication*) [KGJ96, CYD+10, AA18]. This means that the first occurrence of most values will not only be incompressible, but will also require more bits to be represented in compressed format due to encoding overhead.

To reach the full potential of value deduplication, workloads must restrain to merely using a single fixed basic data type — 8, 16, 32 or 64 bits. This is unlikely to happen; lines are composed of data structures and arrays, each of which can contain any composition of these data types, so multiple data types are expected to be found. Pattern-based compressors cope with that by tweaking their patterns to simulate an underlying assumption of smaller data types. For example, the pattern MMMX — where M is a byte match, and X is a byte mismatch, in a notation akin to previous work [CYD+10, AA18] — does not match the least-significant byte, matching only the three most-significant bytes of 32-bit values. To capture similar behavior for 16-bit data types, while still assuming that workloads consist mostly of 32-bit data types, the pattern MXMX can be added; this pattern does not match the least-significant byte of two consecutive 16-bit values, deduplicating the MSB of each.

This idea could be expanded to cover all possible combinations of basic data types; however, the more patterns are added, the more complex and slow becomes the compressor. The number of permutations becomes even more prohibitive with the rising use of 64-bit values. Usually, patterns are selected based on the observation that some bits will consistently have more matches than others; notably, that the MSBs of chunks tend to vary less than their LSB counterpart. Hence, compressors tend to focus on patterns that match the MSB and copy the LSB (*e.g.*, MMMX, and MMXX) [CYD+10, PSM+12, AA18].

Nonetheless, even when the data type of values matches the expectation, similar values do not necessarily generate similar patterns. For instance, the values *0x00000000FFFFFFFF* and *0x0000000100000000* differ by a single unit, yet only their 31 most-significant bits match.

Compressors would typically not cover the pattern "33 non-matching bits followed by 31 matching bits", so this seemingly simple deduplication would generate a XXXXXXXX pattern. Even though there was room to remove duped bits, the compressor's limited number of patterns blocked it from happening, and in fact the extra metadata added to inform the pattern worsened the value's representation.

It is clear that there is a correlation between number of patterns, compressibility and decompression latency; having more patterns increases the compression effectiveness, but also complicates compression hardware, slowing the decompression down. This is seen in Figure 6.1, which shows the average compression ratio of multiple state-of-the-art compressors [PSM+12, KSCE16, CYD+10, AW04b, AA18, AS14b, KGJ96] for the SPEC 2017 benchmarks [Cor17]. BDI [PSM+12], for example, provides a single cycle decompression, which was made possible by only allowing two patterns. Because of that its compression ratio is as high as 86.3%. Other proposals with more patterns, such as C-Pack [CYD+10], and FPC-D [AA18], reach lower ratios, but their decompression processing speed can be as slow as a word per cycle.

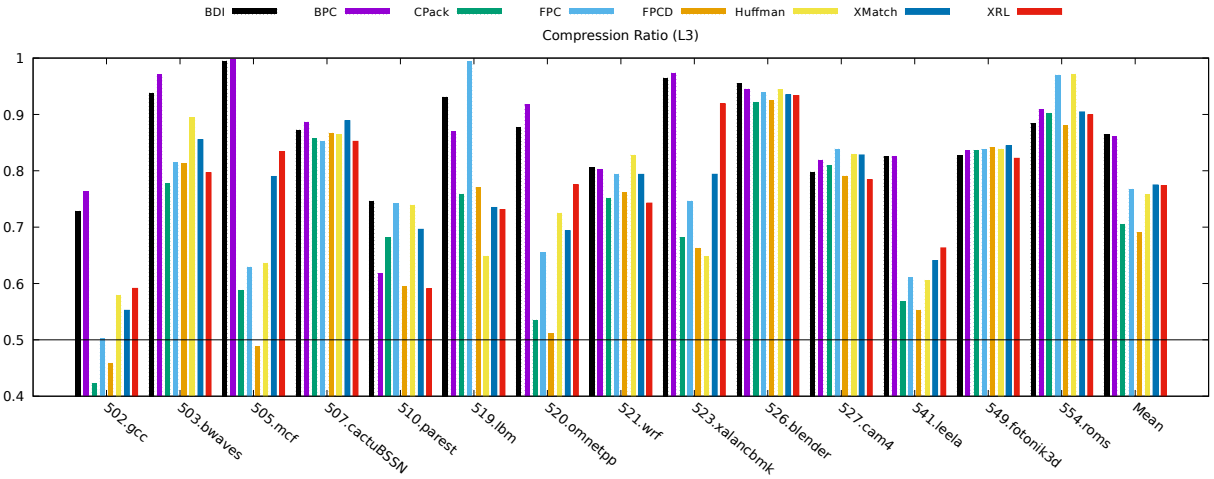


Figure 6.1 – Average compression ratio of SPEC 2017 workloads for multiple state-of-the-art cache compression methods applied to the last-level cache (L3). The lower, the better.

In the next sections we analyze how dictionary-based pattern compression works to try to better understand and ease the problems of pattern matching. This exploration allows us to propose a compressor that loosens the ties between compressibility and decompression latency to achieve low compression ratios **and** low decompression latency.

6.1 Divide and Conquer

As pointed out in the introduction, the selection of a single data type as the representative of a workload plays a big role on the ability to compress lines. This is the case because compressors

divide cache lines into parsing **chunks** and try to capture any regularity between them (Figure 6.2). The chunk size is directly associated to which data types, and how well it can expect to compress; while small chunks cannot capture correlation of bigger data types, big chunks are harder to compress, since their data has higher entropy. As a compromise between both large and small types, cache compressors typically use 32-bit chunks [SASW15].

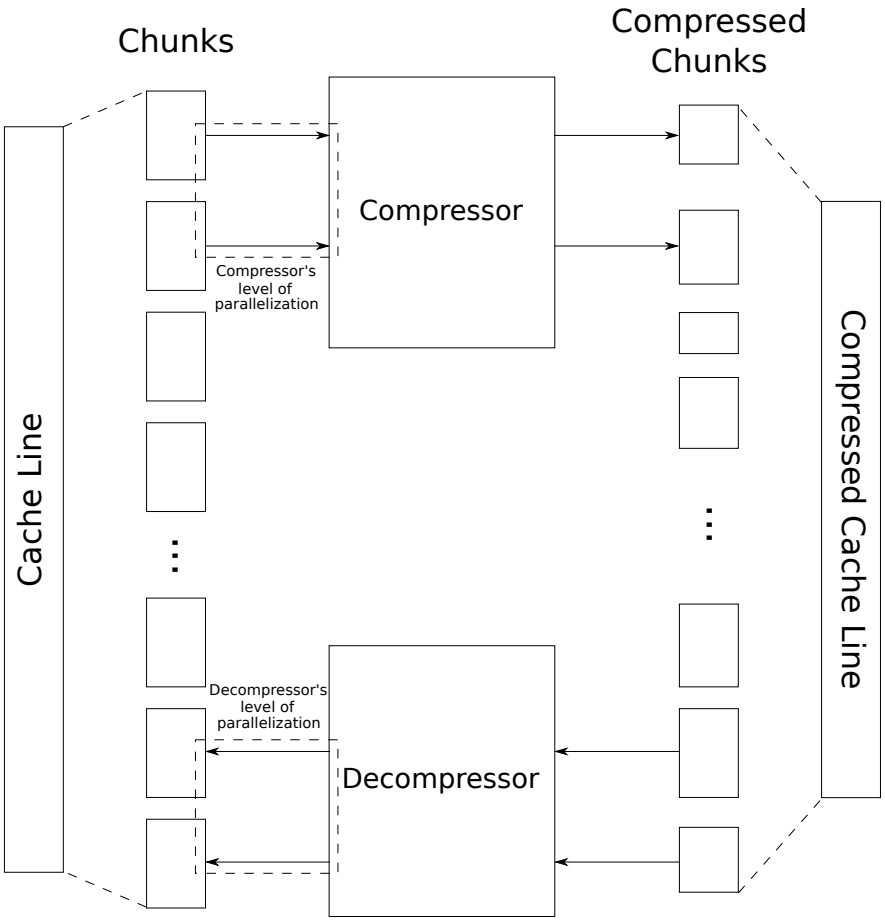


Figure 6.2 – Cache lines are divided into chunks. N chunks can be parsed per cycle (level of parallelization). The level of parallelization of the compression and decompression steps do not need to match.

In general, it has been observed that the contents of the MSB portion of a value present less variability than its LSB counterpart, and thus compressors tend to compress the former better [CYD+10, PSM+12, PS16]. Therefore, it might be advantageous to further divide chunks into different portions, which are compressed differently. The intuition is that the probability of seeing equal values is proportional to the chunk size ($\frac{1}{chunkSize}$), so the probability of referring to previous dictionary entries is higher, increasing compression efficiency. Moreover, since each portion is still expected to represent the same respective bits of the chunk, the overall assumption

of the workload's representative data type is kept.

We hereby propose the concept of **Region-Chunk compression**. A compressor under Region-Chunk compression further divides chunks into equally-sized **regions**, each of which is compressed independently (Figure 6.3). A region is then a smaller portion of the chunk which exposes the correlation between different chunks at a finer granularity. We will refer to compressors that divide cache lines into w-bit chunks, each with x-bit regions as a $\mathbf{R}_x\mathbf{C}_w$ **compressor**. It is worth noticing that conventional compression fits the case where the chunk size matches the region size; therefore, it is a subset of Region-Chunk compression — *i.e.*, each chunk contains a single region, and thus uses a single compressor.

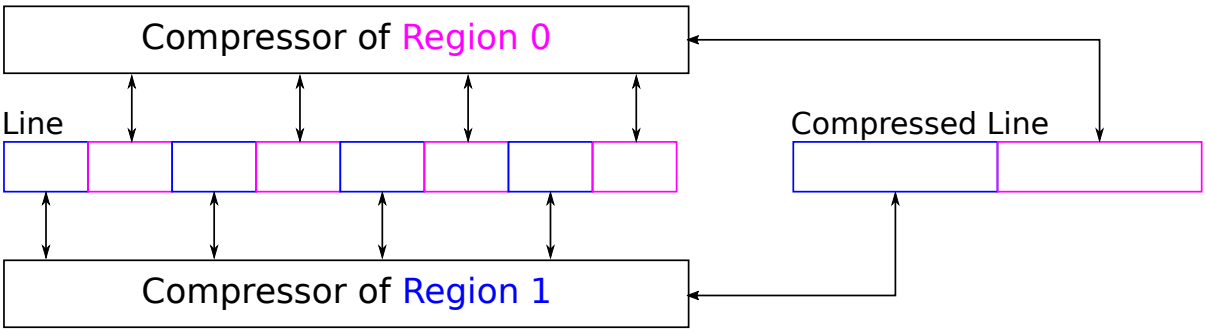


Figure 6.3 – Region-Chunk compression, with two regions per chunk. Each blue-pink pair in the original line corresponds to an original line's chunk. Compressing and decompressing regions works by using only the regions of interest of the cache line as input to their respective compressors.

The main advantage of adding sub-divisions is that the chunk size still matches the overall workload's expected predominant data type, yet occurrences of equal or smaller data types are compressed more efficiently — the finer granularity at which dictionaries are built reduces duplication. Furthermore, the number of patterns covered is implicitly increased, since the combination of the patterns of each region's compressors generate a larger spectrum. For example, for 64-bit chunks the pattern MMMMMMXMM would usually not be part of the selected patterns; however, it would be assured as a possible combination of the four regions in a $\mathrm{R}_{16}\mathrm{C}_{64}$ compressor that contains the patterns MM and MX (MM + MM + MX + MM).

Another advantage of refining the granularity is that each region's compressor can be modified to cover its necessities. For instance, by expecting that MSB regions have less variability than LSB regions, it would be reasonable to reduce the maximum number of different dictionary entries allowed or the number of patterns covered, which in turn would reduce the number of metadata bits needed. Another way to tailor these compressors would be to modify the patterns themselves — *e.g.*, by increasing or decreasing the number of non-matching bits one could increase the likelihood of deduplicating entries, or reduce the size of the compressed data, respectively.

Listing 6.1 – An example of a structure containing mixed types.

```
struct {
  int value;
  short descriptors[2];
} Structure;
```

For instance, Table 6.1 shows how a conventional (32-bit chunks, 32-bit regions) generic dictionary-based pattern compressor would parse an example 64-byte cache line which represents a vector of the mixed data structure described in Listing 6.1. Table 6.2 depicts how it would be done with a generic dictionary-based pattern compressor parsing lines as 32-bit chunks and 16-bit regions ($R_{16}C_{32}$).

Knowing that each region has less variability — and thus is likely to require less dictionary entries — allows for a reduction of the number of available dictionary entries, generating a direct cut on the size of the pointer fields. Besides, although the number of patterns is kept the same — and therefore the number of pattern bits is doubled — the finer granularity of the patterns results in deduplication within the original X bits. Other configurations could be used as well: in Table 6.3, even though the chunk size does not match any of the types in the structure, a R_8C_{64} can still exploit the predominant characteristics of each region.

6.1.1 Applying Region-Chunk to state-of-the-art compressors

Figure 6.4 shows the average compression ratio achieved by multiple state-of-the-art compressors when using the Region-Chunk concept at a 32-bit region granularity. Each chunk is split into two regions, and each region is parsed by its respective region's compressor. Some of the compressors benefit from the use of the Region-Chunk concept. The benefits are more noticeable when coupling it with Pairwise Space Sharing: as it can be seen in Figure 6.5, all compressors except X-Match [KGJ96] improved their average compression ratio. *Therefore, further dividing chunks into regions is a concept that is beneficial to dictionary-based cache compressors in general.* Furthermore, this experiment did not take into account the fact that making lines smaller decreases the required compressor complexity, so it is conceivable that some of these compressors may even achieve a reduction of their decompression latency.

Notice that both figures do not present the results of using the Region-Chunk concept with BDI. This is due to the fact that base-delta compressors have fixed sizes, formulated for a specific cache line size configuration. Since $R_{32}C_{64}$ divides the cache line into two, BDI becomes not applicable as proposed. Nonetheless, alternatives will be presented shortly.

chunk	B_3	B_2	B_1	B_0	Pattern	Pointer	X bits
0	*0x00*	*0x00*	*0x00*	*0x00*	XXXX	-	*0x00000000*
1	*0xAC*	*0x00*	*0xAC*	*0x01*	XXXX	-	*0xAC00AC01*
2	*0x00*	*0x00*	*0x00*	*0x01*	MMMX	0000_2	*0x01*
3	*0xAC*	*0x02*	*0xAC*	*0x03*	MXMX	0001_2	*0x02,0x03*
4	*0x00*	*0x00*	*0x00*	*0x02*	MMMX	0000_2	*0x02*
5	*0xAC*	*0x04*	*0xAC*	*0x05*	MXMX	0001_2	*0x04,0x05*
6	*0x00*	*0x00*	*0x00*	*0x03*	MMMX	0000_2	*0x03*
7	*0xAC*	*0x06*	*0xAC*	*0x07*	MXMX	0001_2	*0x06,0x07*
8	*0x00*	*0x00*	*0x00*	*0x04*	MMMX	0000_2	*0x04*
9	*0xAC*	*0x08*	*0xAC*	*0x09*	MXMX	0001_2	*0x08,0x09*
10	*0x00*	*0x00*	*0x00*	*0x05*	MMMX	0000_2	*0x05*
11	*0xAC*	*0x0A*	*0xAC*	*0x0B*	MXMX	0001_2	*0x0A,0x0B*
12	*0x00*	*0x00*	*0x00*	*0x06*	MMMX	0000_2	*0x06*
13	*0xAC*	*0x0C*	*0xAC*	*0x0D*	MXMX	0001_2	*0x0C,0x0D*
14	*0x00*	*0x00*	*0x00*	*0x07*	MMMX	0000_2	*0x07*
15	*0xAC*	*0x0E*	*0xAC*	*0x0F*	MXMX	0001_2	*0x0E,0x0F*

Table 6.1 – Simple example of conventional compression using a generic dictionary-based pattern compressor with up to sixteen dictionary entries and four possible patterns: MMMM, MMMX, MXMX, and XXXX, where M is a byte match, and X is a byte mismatch. The total compressed size is $(numChunks \cdot encodingSize) + (numPointers * pointerSize) + totalXBitsSize = (16 \cdot 2) + (14 \cdot 4) + (2 \cdot 32 + 21 \cdot 8) = 320$ bits.

6.2 Latency of a Region-Chunk Compressor

As depicted in Figure 6.3, regions are just re-ordered smaller portions of a cache line; thus, compressing each region is analogous to compressing the whole line — *i.e.*, any existing compressor can be used. The input of each region's compressor is directly wired to its respective fixed portions in the original cache line. *This means that the compression and decompression latency of a compressor using the Region-Chunk concept is equal to the compression and decompression latency of its slowest region compressor.*

As mentioned previously, base-delta compressors [PSM+12] achieve minimal decompression latency because they allow a strictly limited number of patterns; yet, with the use of Region-Chunk compression, the patterns covered by the compressor becomes a compound of the patterns covered in each region, at no extra latency cost. These two concepts can be combined to create a compressor with low compression ratio **and** low decompression latency.

We hereby propose that each region's compressor encompasses multiple simpler sub-compressors based on the base-delta compression technique (*multi-compressors* — Figure 6.6). Consequently, adding the decompression latency of base-delta compressors to the calculation, a multi-compressor

chunk	R_1	R_0	$Pattern_1$	$Pointer_1$	X_1 bits	$Pattern_0$	$Pointer_0$	X_0 bits
0	*0x0000*	*0x0000*	XXXX	-	*0x0000*	XXXX	-	*0x0000*
1	*0xAC00*	*0xAC01*	XXXX	-	*0xAC00*	XXXX	-	*0xAC01*
2	*0x0000*	*0x0001*	MMMM	00_2	-	MMMX	000_2	*0x1*
3	*0xAC02*	*0xAC03*	MMMX	01_2	*0x2*	MMMX	001_2	*0x3*
4	*0x0000*	*0x0002*	MMMM	00_2	-	MMMX	000_2	*0x2*
5	*0xAC04*	*0xAC05*	MMMX	01_2	*0x4*	MMMX	001_2	*0x5*
6	*0x0000*	*0x0003*	MMMM	00_2	-	MMMX	000_2	*0x3*
7	*0xAC06*	*0xAC07*	MMMX	01_2	*0x6*	MMMX	001_2	*0x7*
8	*0x0000*	*0x0004*	MMMM	00_2	-	MMMX	000_2	*0x4*
9	*0xAC08*	*0xAC09*	MMMX	01_2	*0x8*	MMMX	001_2	*0x9*
10	*0x0000*	*0x0005*	MMMM	00_2	-	MMMX	000_2	*0x5*
11	*0xAC0A*	*0xAC0B*	MMMX	01_2	*0xA*	MMMX	001_2	*0xB*
12	*0x0000*	*0x0006*	MMMM	00_2	-	MMMX	000_2	*0x6*
13	*0xAC0C*	*0xAC0D*	MMMX	01_2	*0xC*	MMMX	001_2	*0xD*
14	*0x0000*	*0x0007*	MMMM	00_2	-	MMMX	000_2	*0x7*
15	*0xAC0E*	*0xAC0F*	MMMX	01_2	*0xE*	MMMX	001_2	*0xF*

Table 6.2 – Simple example with two regions using a generic $R_{16}C_{32}$ dictionary-based pattern compressor with four patterns: MMMM, MMMX, MMXX, and XXXX, where M is a 4-bit match, and X is a 4-bit mismatch. Each region has its own compressor: The MSB region supports up to 4 dictionary entries, and the LSB supports 8. The total compressed size is $(16 \cdot 2 + 14 \cdot 2 + (2 \cdot 16 + 7 \cdot 4)) + (16 \cdot 2 + 14 \cdot 3 + (2 \cdot 16 + 14 \cdot 4)) = 282$ bits.

chunk	R_7	R_6	R_5	R_4	R_3	R_2	R_1	R_0
0	*0x00*	*0x00*	*0x00*	*0x00*	*0xAC*	*0x00*	*0xAC*	*0x01*
1	*0x00*	*0x00*	*0x00*	*0x01*	*0xAC*	*0x02*	*0xAC*	*0x03*
2	*0x00*	*0x00*	*0x00*	*0x02*	*0xAC*	*0x04*	*0xAC*	*0x05*
3	*0x00*	*0x00*	*0x00*	*0x03*	*0xAC*	*0x06*	*0xAC*	*0x07*
4	*0x00*	*0x00*	*0x00*	*0x04*	*0xAC*	*0x08*	*0xAC*	*0x09*
5	*0x00*	*0x00*	*0x00*	*0x05*	*0xAC*	*0x0A*	*0xAC*	*0x0B*
6	*0x00*	*0x00*	*0x00*	*0x06*	*0xAC*	*0x0C*	*0xAC*	*0x0D*
7	*0x00*	*0x00*	*0x00*	*0x07*	*0xAC*	*0x0E*	*0xAC*	*0x0F*

Table 6.3 – Example cache line, divided into 64-bit chunks and 8-bit regions (R_8C_{64}).

of base-delta compressors generates an extra data access latency of 1 cycle and 3 cycles, when storing encoding in the tags and data arrays, respectively. In the context of Region-Chunk compression, it is preferred to store the encoding bits of multi-compressors in the tags only when the region size is close to the chunk size compression.

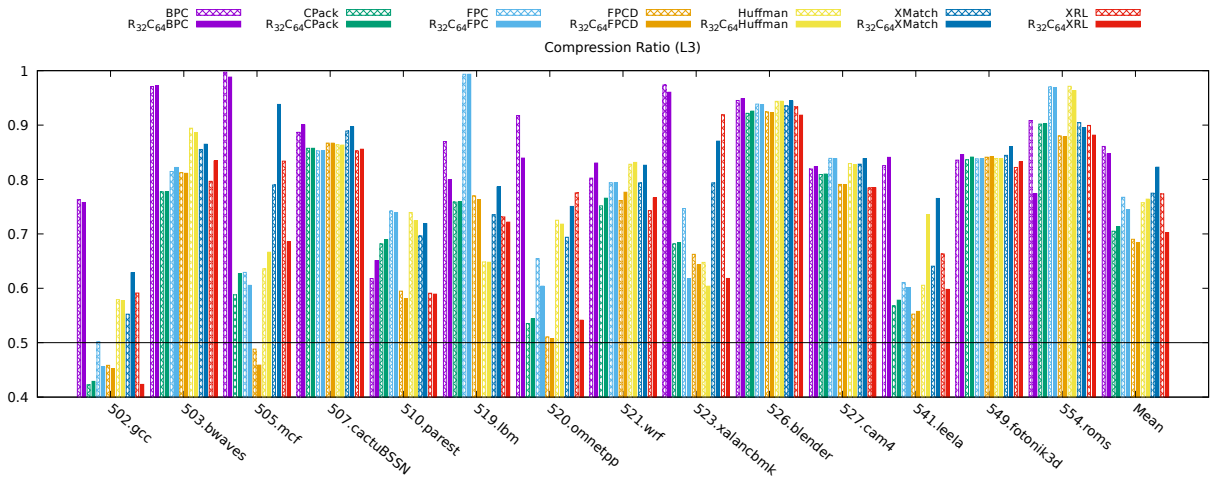


Figure 6.4 – The $R_{32}C_{64}$ concept was applied to multiple state-of-the-art proposals, so that there are two compressors per chunk. Some of them benefit from Region-Chunk compression.

6.2.1 Generalizing Base-Delta Compressors

BDI [PSM+12] is a a multi-compressor composed of multiple base-delta compressors, exchanging compression efficiency for decompression speed. Each base-delta compressor's compressed data contains three fields: a *base*, which is the first unique non-zero chunk seen when parsing a line; an array of *deltas*, which represent the value difference between a given chunk and a base; and a *bitmask* that associates a base to each delta. This last field is needed because, besides the explicitly stored base, BDI has an implicit second base, the zero value. This way, when the value zero is used as the base for a delta, a special index is used in the bitmask field, yet the zero-base itself is not stored.

We have previously examined the efficacy of state-of-the-art compressors in Chapter 5, and it is clear that BDI presents a poor average compression ratio (Figure 5.1. One of the main reasons for this outcome is its highly restrictive number of bases: for example, in SPEC 2017 the average percentage of cache lines that are compressible with each of BDI's sub-compressors — Zeros, Repeated Values, $B8\Delta4$, $B8\Delta2$, $B8\Delta1$, $B4\Delta2$, $B4\Delta1$, $B2\Delta1$ — is, respectively, 11.3%, 12.7%, 15.0%, 17.1%, 42%, 14.6%, 19.3%, 15.1% (Figure 6.7). This means that most of the cache lines being compressed would have needed more than the available pair of bases. We herewith generalize the concept of a base-delta compressor to extend its support to any number of bases: *a base-delta compressor that parses cache lines in chunks of w bits to store up to x implicit and y explicit w-bit bases, and whose deltas have z bits is represented as* $\boldsymbol{C}_w\boldsymbol{I}_x\boldsymbol{E}_y\boldsymbol{D}_z$.

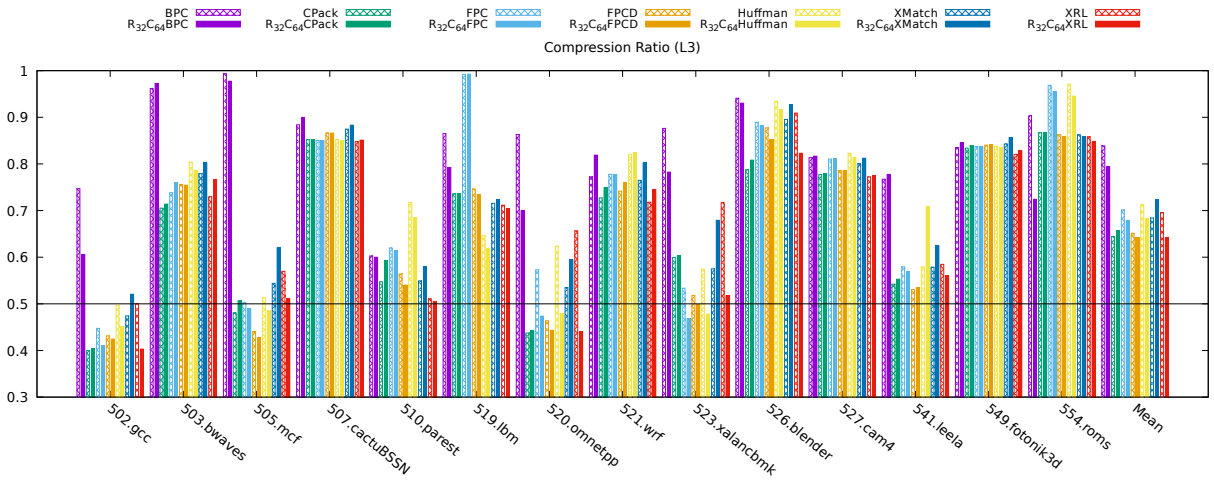


Figure 6.5 – The $R_{32}C_{64}$ concept was applied to multiple state-of-the-art proposals, **coupled with Pairwise Space Sharing**. Most of them benefit from Region-Chunk compression.

6.2.2 Optimizing Base-Delta Compressors

In compression, every bit matters; saving bits becomes even more important at smaller chunk sizes, since the amount of metadata is increased; therefore, before defining the ideal set of base-delta compressors to be used with Region-Chunk, we review $C_wI_xE_yD_z$ compressors to maximize their efficiency while still keeping their single-cycle decompression latency.

Ideally, base-delta compressors should select their bases based on the range of values in the cache line: the selection should minimize the number of bases required to compress a line. In practice, Pekhimenko *et al.* show that arbitrarily picking the first occurrence of a new value only marginally degrades performance, while reducing both hardware complexity and compression latency [PSM+12]. Consequently, assuming that there is a uniform distribution of values, we can expand on this idea to affirm that the probability of being able to compress subsequent values after the base is set is the same, regardless of the LSB of such base. This can be leveraged to reduce the number of bits needed to represent the bases; if the base's z least-significant bits are always fixed at a value — *e.g.*, zero — they can be known implicitly, and only its MSB need to be explicitly stored in the base's field. The mapping on the top of Figure 6.8 depicts this idea.

For instance, assume that a $C_{32}I_1E_1D_8$ compressor is used to parse the 64-bit cache line *0x0123456701234568*. In its non-optimized version, the first 32-bit value (*0x01234567*) would be stored as the base *0x01234567* with a delta of *0x00*, and the second value would be stored as the delta *0x01* and a reference to the previous base. Elseways, when the base optimization is applied, the first value would be stored as *0x012345* with a delta of *0x67*, and the second value would be stored as the delta *0x68*. In the latter version, the deltas are relative to the base's implicit extended value, *0x01234500*. Although not formally defined, this idea can be seen in the

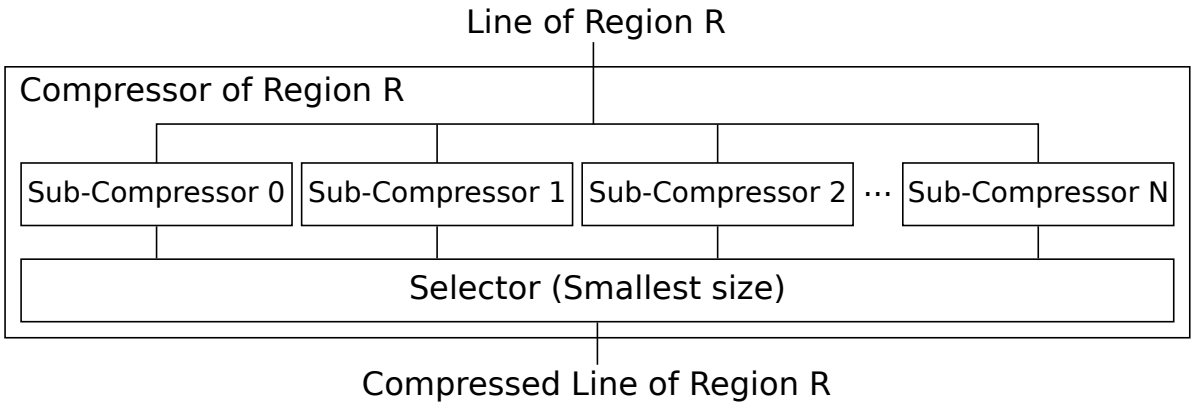


Figure 6.6 – A multi-compressor consisting of N+1 sub-compressors, which compresses the contents of region R.

literature in DISH's Scheme-II [PS16] — it is a $C_{28}I_0E_4D_4$ compressor.

Conceptually, this optimization narrows the gap between base-delta compressors and regular pattern compressors; a $C_wI_xE_yD_z$ compressor behaves akin to a dictionary-based pattern compressor with 2 patterns: the uncompressed pattern and the pattern that matches all bits from the MSB up to, but not including, the z delta bits. Basically, the difference between these compressors is that the number of times that each pattern can occur in a $C_wI_xE_yD_z$ compressor is fixed. For example, when compressing a 64B cache line, a $C_{32}I_1E_2D_8$ compressor allows precisely two occurrences of the pattern XXXX, and fourteen occurrences of the pattern MMMX.

Another minor optimization can be done regarding the bitmask/pointer field of dictionary-based pattern compressors in general. Their dictionaries are populated on the fly, and they start in either an empty state, or pre-populated with fixed values (the implicit bases). This means that the initial chunks being parsed will only reference the initial dictionary entries; thus, some bits can be cut off from the initial bitmasks. For example, a $C_wI_1E_3D_z$ compressor will always initialize the dictionary with the implicit value. When parsing the cache line, the first chunk can refer to two possible values — the implicit base, or a new base — which only requires 1 bitmask bit. The second chunk must assume the worst case scenario, in which the first chunk did not use the implicit base; therefore, it can refer to three possible values — the implicit base, the base added by the first chunk, and a new base — which starts requiring two bits. This process is repeated until the maximum number of dictionary entries is reached, which defines an upper limit for the number of bits. The number of bits required to represent the index of the N_{th} chunk is generalized by Equation 6.1.

$$numBitmaskBits_N = \log_2(numImplicitValues + \min(N, numExplicitValues)) \quad (6.1)$$

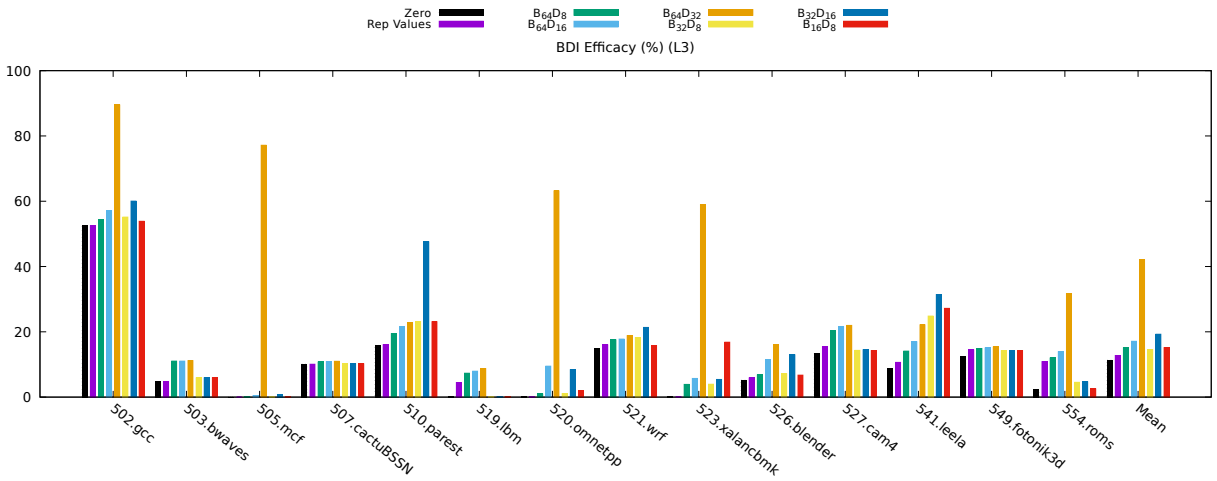


Figure 6.7 – BDI's sub-compressors have a low rate of success on the task of compressing data. The y axis is the percentage of blocks successfully compressed using the given scheme.

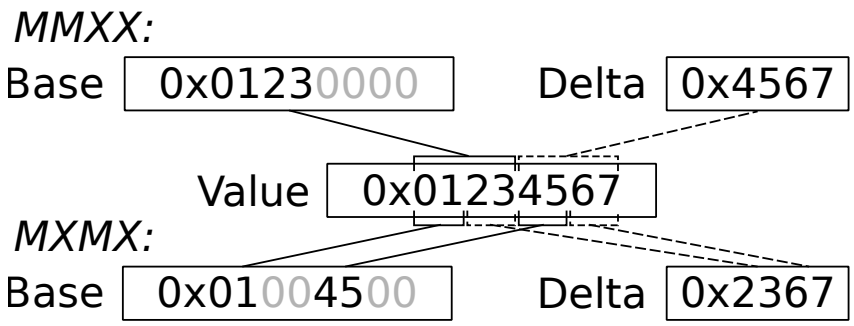


Figure 6.8 – Mapping deltas to different positions of the values. M is a base byte, and X is a delta byte. When using mapping MMXX (top), the deltas are calculated relative to the last two bytes of the values. When using mapping MXMX (bottom), deltas are relative to the first and third least-significant bytes.

Neither of these optimizations increase hardware complexity, because they are defined on compressor conception — for example, the number of bitmask bits that each chunk needs is predefined with respect to the number of implicit and explicit bases the compressor has. Unless stated otherwise, it should be assumed that $\mathrm{C}_w\mathrm{I}_x\mathrm{E}_y\mathrm{D}_z$ compressors default to using both these optimizations.

6.2.3 Dissociating Base Size from Parsing Type

As mentioned before, there are multiple basic types, and even if workloads have a single predominant type, it is likely that other types are also present, in smaller portions. Therefore, some lines could benefit from base-delta compressors that cover these corner cases. Base-delta compressors, however, assume that there will be byte matches in the most-significant bits of the

chunk, and that the deltas are relative to the least-significant bits, which means that a given base size does not process well data types smaller than it.

Similarly to regular pattern compressors, $C_wI_xE_yD_z$ can use other patterns to loosen the relationship between the size of a base and its ability to parse data types — *i.e.*, instead of assuming that the deltas are the least-significant bits of the base, they can be remapped to the least-significant bits of the desired type. For example, the default $C_{32}I_xE_yD_{16}$ compressor is designed to target 32-bit data types, using patterns XXXX and MMXX. To cover 16-bit data types, one could substitute the MMXX pattern by the MXMX pattern, as presented in the bottom mapping of Figure 6.8. Unless specified otherwise, it is assumed that the delta bytes of $C_wI_xE_yD_z$ compressors are taken from the chunks' least-significant bits.

6.3 Stride Compressor

Sometimes, the deltas in a base-delta compressor present a certain characteristic that can be described by a mathematical equation. When this happens, the deltas can be compressed into a smaller representation, by providing only the variables of the equation. A common occurrence of this class of base-delta compressors is the arithmetic sequence — deltas differing by a fixed value (a stride).

To improve compression in these cases, we propose the **Stride Compressor**. Instead of storing every delta entry as in base-delta compressors, the Stride Compressor stores a single base-delta pair, where the base is the first term in the sequence, and the delta is the common difference between terms. For example, in the sequence of 16-bit values *0xF511*, *0xF514*, *0xF517*, *0xF51A*, *0xF51D*, *0xF520*, *0xF523*, and *0xF526*, the deltas differ by 3 units, thus the Stride Compressor compresses the sequence as the base *0xF511*, and the stride *0xF503*. Decompression is trivial — any chunk C is given by $C_n = base + (n-1) \cdot delta$ — requiring no more than a single cycle.

6.4 Selecting Sub-Compressors

A R_xC_w is composed of $\frac{w}{x}$ region compressors. These region compressors can be instances of any compressor (*e.g.*, C-Pack [CYD+10] or FPC-D [AA18]); however, to leverage on the speed of the $C_wI_xE_yD_z$ compressors, we propose a set of multi-compressors ($SubR_xC_w$) specialized for fast decompression. The process used to select which sub-compressors worked best for each $SubR_xC_w$ is described here. These configurations are defined once, **at design time** (*i.e.*, no runtime overhead). Each $SubR_xC_w$ has $S = 2^k - 1$ sub-compressors — one encoding must be reserved to indicate that all sub-compressors failed, and data is in uncompressed format.

In order to select which sub-compressors worked best for each $SubR_xC_w$ configuration, we

defined a "best compression ratio" (BCR) configuration that would determine the upper limit of the compressibility of each workload. Ideally, each BCR $\mathrm{SubR}_x\mathrm{C}_w$ configuration should contain all possible values for y and z in $\mathrm{C}_w\mathrm{I}_x\mathrm{E}_y\mathrm{D}_z$; however, the exploration space would become unreasonably large, so a few constraints were established to reduce redundancy: $0 \equiv (x + y) \mod 2$, and $x|x \in \{0, \frac{w}{2}, \frac{w}{4}, \frac{w}{8}\}$. Finally, whenever the BCR did not have enough sub-compressors to fill S under these constraints, a few fill sub-compressors that did not abide by these rules were added. The following values were used to fill the implicit bases, in this order: 0, -1, 1, 2, 8. These have been chosen due to their higher frequency in workloads [KGJ98]. For instance, if a compressor uses two implicit bases, then the values 0 and -1 are selected.

After each $\mathrm{SubR}_x\mathrm{C}_w$'s BCR configuration is determined, the sub-compressors were individually removed with respect to their redundancy — "Does a configuration with fewer explicit bases or smaller deltas compress as well as it?" — and usefulness — "Does a configuration with more explicit bases or bigger deltas cover more cases at a low extra compressed size cost?". If the IPC, compression or compaction ratio were significantly changed after a removal, the sub-compressor would be reinstated, and the process would be repeated with another sub-compressor. Table 6.4 contains the final configuration for each of the $\mathrm{SubR}_x\mathrm{C}_w$ compressors. On average, each BCR baseline's average compressed size differs from the final configurations' by 3 bits. It is important to notice that configurations and results for the compressors based on 8 and 16-bit chunks may not be representative of the adopted workloads' most frequent data types, and thus are possibly sub-optimal.

6.5 Results

This section contains an analysis of the efficiency and effectiveness of the Region-Chunk concept, as well as the proposed multi-compressors that make use of this concept. We do not show results for $\mathrm{R}_x\mathrm{C}_{16}$ and $\mathrm{R}_x\mathrm{C}_8$ since these chunk sizes are not representative of the adopted workload's most frequent data types. When suitable the compressors are compared against the best performing state-of-the-art compressor, FPC-D; however, the 2-cycle decompression latency adopted in FPC-D's original proposal is utterly unrealistic, given that it has to handle back propagation of patterns, besides having a significantly higher number of patterns when compared to base-delta compressors. As a result, we assume it requires 4 cycles to decompress. The methodology to generate the results in this chapter was the same as the one used in Chapter 5 (see Section 5.2).

6.5.1 Base-Delta Optimizations

Figure 6.9 shows a comparison of the compaction ratio of $\mathrm{R}_{16}\mathrm{C}_{64}$ using the regular base-delta compression, the base-delta compressor with delta bits represented implicitly, and the

Compressor	Sub-Compressors
$SubR_8C_8$	Stride (8-bits delta), $C_8I_0E_2D_0$, $C_8I_0E_8D_0$
$SubR_8C_{16}$	Stride (8-bits delta), $C_8I_1E_0D_0$, $C_8I_1E_0D_6$, $C_8I_1E_1D_0$, $C_8I_2E_2D_0$, $C_8I_3E_1D_0$, $C_8I_1E_3D_0$, $C_8I_0E_4D_0$, $C_8I_0E_4D_4$, $C_8I_5E_3D_0$, $C_8I_4E_4D_0$, $C_8I_1E_7D_4$, $C_8I_0E_8D_0$, $C_8I_5E_{11}D_0$, $C_8I_2E_{14}D_0$
$SubR_{16}C_{16}$	$C_{16}I_1E_0D_0$, $C_{16}I_1E_0D_8$, $C_{16}I_1E_1D_0$, $C_{16}I_1E_1D_8$, $C_{16}I_2E_0D_0$, $C_{16}I_3E_1D_0$, $C_{16}I_2E_2D_0$, $C_{16}I_1E_3D_0$, $C_{16}I_1E_3D_4$, $C_{16}I_0E_4D_0$, $C_{16}I_4E_4D_0$, $C_{16}I_3E_5D_0$, $C_{16}I_2E_6D_0$, $C_{16}I_1E_7D_0$, $C_{16}I_0E_8D_0$
$SubR_8C_{32}$	Stride (8-bits delta), $C_8I_1E_0D_0$, $C_8I_0E_1D_0$, $C_8I_0E_1D_5$, $C_8I_1E_1D_0$, $C_8I_2E_0D_0$, $C_8I_0E_2D_0$, $C_8I_2E_2D_0$, $C_8I_1E_3D_0$, $C_8I_0E_4D_0$, $C_8I_5E_3D_0$, $C_8I_4E_4D_0$, $C_8I_3E_5D_0$, $C_8I_2E_6D_0$, $C_8I_1E_7D_0$
$SubR_{16}C_{32}$	$C_{16}I_1E_0D_0$, $C_{16}I_1E_0D_8$, $C_{16}I_1E_1D_0$, $C_{16}I_0E_2D_0$, $C_{16}I_0E_2D_8$, $C_{16}I_3E_1D_0$, $C_{16}I_2E_2D_0$, $C_{16}I_2E_2D_8$, $C_{16}I_1E_3D_0$, $C_{16}I_0E_4D_0$, $C_{16}I_0E_4D_8$, $C_{16}I_3E_5D_0$, $C_{16}I_3E_5D_8$, $C_{16}I_2E_6D_0$, $C_{16}I_1E_7D_0$
$SubR_{32}C_{32}$	$C_{32}I_1E_0D_{16}$ (BDBD), $C_{32}I_1E_1D_8$, $C_{32}I_1E_1D_{16}$, $C_{32}I_2E_2D_0$, $C_{32}I_2E_2D_8$, $C_{32}I_1E_3D_0$, $C_{32}I_1E_3D_8$, $C_{32}I_0E_4D_0$, $C_{32}I_0E_4D_8$, $C_{32}I_4E_4D_0$, $C_{32}I_3E_5D_0$, $C_{32}I_3E_5D_8$, $C_{32}I_2E_6D_0$, $C_{32}I_1E_7D_0$, $C_{32}I_0E_8D_0$
$SubR_8C_{64}$	Stride (8-bits delta), $C_8I_0E_1D_0$, $C_8I_1E_1D_0$, $C_8I_2E_0D_0$, $C_8I_2E_2D_0$, $C_8I_1E_3D_0$, $C_8I_5E_3D_0$
$SubR_{16}C_{64}$	Stride (16-bits delta), $C_{16}I_1E_0D_0$, $C_{16}I_0E_1D_0$, $C_{16}I_0E_1D_8$, $C_{16}I_1E_1D_0$, $C_{16}I_1E_1D_8$, $C_{16}I_0E_2D_0$, $C_{16}I_0E_2D_8$, $C_{16}I_3E_1D_0$, $C_{16}I_2E_2D_0$, $C_{16}I_2E_2D_8$, $C_{16}I_1E_3D_0$, $C_{16}I_0E_4D_0$, $C_{16}I_4E_4D_0$, $C_{16}I_3E_5D_0$
$SubR_{32}C_{64}$	Stride (32-bits delta), $C_{32}I_1E_0D_0$, $C_{32}I_0E_1D_0$, $C_{32}I_1E_1D_0$, $C_{32}I_1E_1D_8$, $C_{32}I_1E_1D_{16}$, $C_{32}I_3E_1D_0$, $C_{32}I_2E_2D_0$, $C_{32}I_2E_2D_8$, $C_{32}I_1E_3D_0$, $C_{32}I_1E_3D_8$, $C_{32}I_1E_3D_{16}$, $C_{32}I_4E_4D_0$, $C_{32}I_3E_5D_0$, $C_{32}I_3E_5D_8$
$SubR_{64}C_{64}$	Stride (64-bits delta), $C_{64}I_1E_0D_0$, $C_{64}I_1E_0D_{32}$, $C_{64}I_1E_0D_{32}$ (BBDDBBDD), $C_{64}I_1E_0D_{32}$ (BDBDBDBD), $C_{64}I_0E_1D_0$, $C_{64}I_1E_1D_0$, $C_{64}I_1E_1D_8$, $C_{64}I_1E_1D_{16}$, $C_{64}I_2E_2D_0$, $C_{64}I_2E_2D_8$, $C_{64}I_2E_2D_{16}$, $C_{64}I_2E_2D_{16}$ (BBBDBBBD), $C_{64}I_1E_3D_0$, $C_{64}I_1E_3D_8$

Table 6.4 – List of sub-compressors used in the best configuration for each of the R_xC_w compressors. For example, $R_{16}C_{64}$ is composed of four instances of $SubR_{16}C_{64}$, one per region.

fully optimized base-delta compressor (implicit deltas + bitmask optimization). These results confirm the hypothesis: these optimizations increase the efficiency and efficacy of the base-delta compressors, for all workloads.

6.5.2 The R_xC_w Compressors

When using Region-Chunk, the bigger the chunk size, the more data types are covered by the compressor; and the smaller the region size, the higher the chance to deduplicate bases. However, the more regions exist, the more metadata is added. Figure 6.10 shows the average compression and compaction ratios for each of the R_xC_{64} and R_xC_{32} compressors proposed,

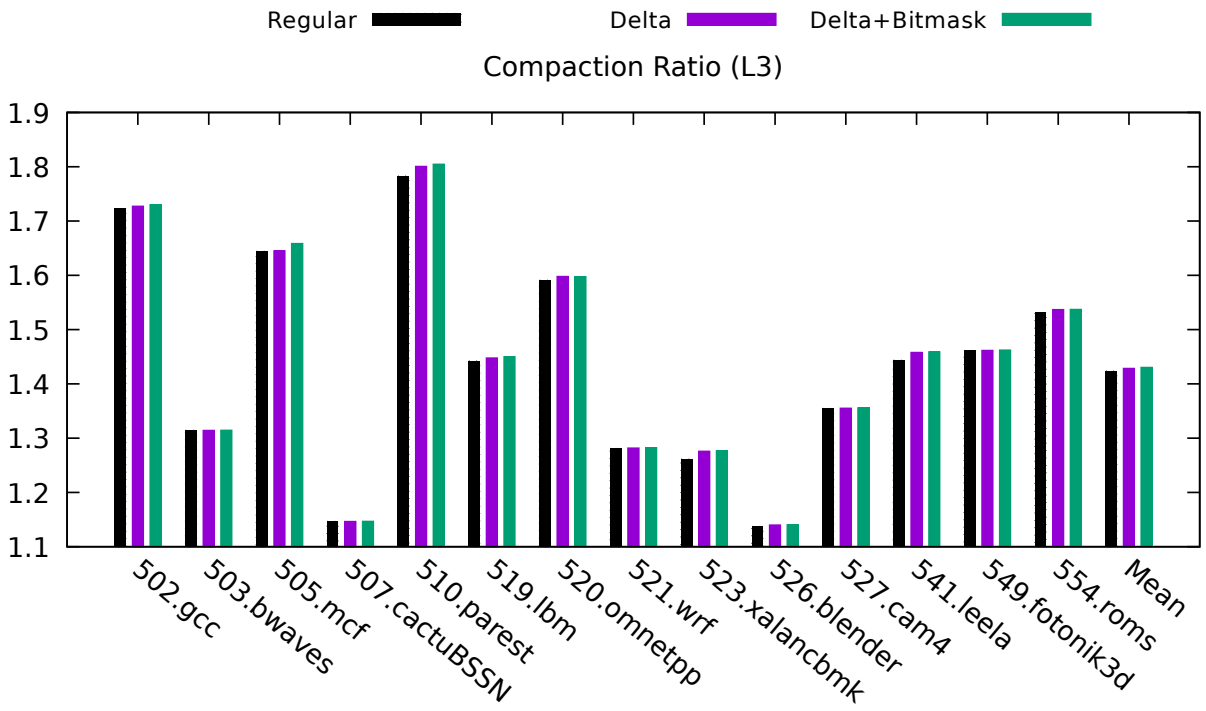


Figure 6.9 – Comparison of $R_{16}C_{64}$ using different levels of optimization.

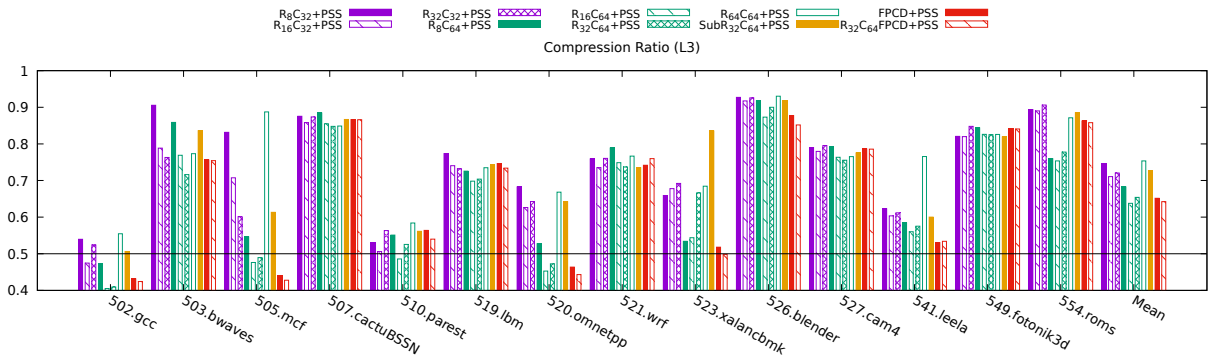
as well as a $SubR_{32}C_{64}$ without the Region-Chunk concept. Overall, a region size of 16 bits generates a positive trade-off between data duplication and metadata increase.

The proposed R_xC_w compressors perform comparably to state-of-the-art compressors. In addition, the results for $SubR_{32}C_{64}$ show that *the main improvements of R_xC_w are due to the region abstraction, **not** the new selection of base-delta compressors.* Finally, FPC-D improved its average compression ratio by 2% — BPC, C-Pack, FPC, Huffman and X-RL, although not depicted, have improvents in the range of 1-8%. This means that *further dividing chunks into regions is beneficial to dictionary-based cache compressors in general.*

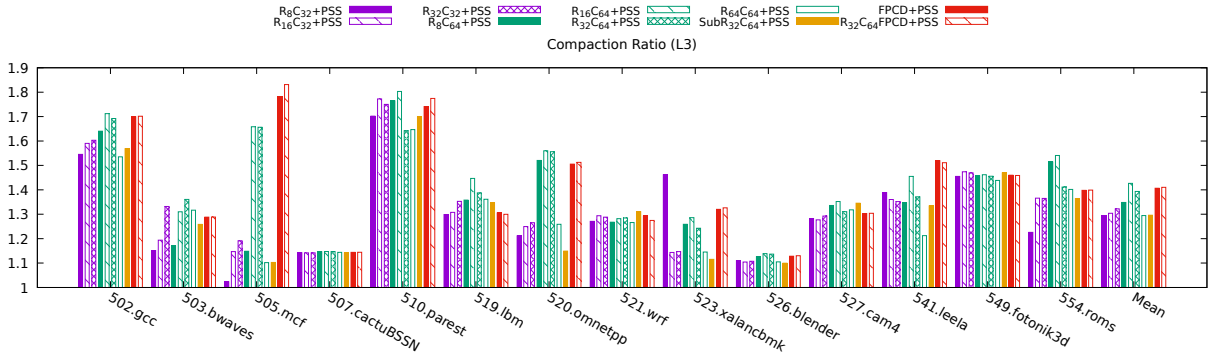
6.5.3 Single-Cycle Decompression

So far all experiments have assumed that the regions' encodings are stored in the data entry; thus, each compressor's decompression latency is 3 cycles. However, if the encodings are stored in the tag entry — as in BDI — part of the decompression process can be done in parallel with the data access, reducing the effective decompression latency to 1 cycle. Furthermore, when bits are removed from the data entry, some sub-compressors — notably the ones whose compressed size is close or equal to half the entry's size — co-allocate better.

Figures 6.11a and 6.11b present, respectively, the IPC and compaction ratio when storing the encoding in the tag and in the data entry, as well as a comparison against BDI, FPC-D and



(a) Compression ratio — the lower, the better.



(b) Compaction ratio — the higher, the better.

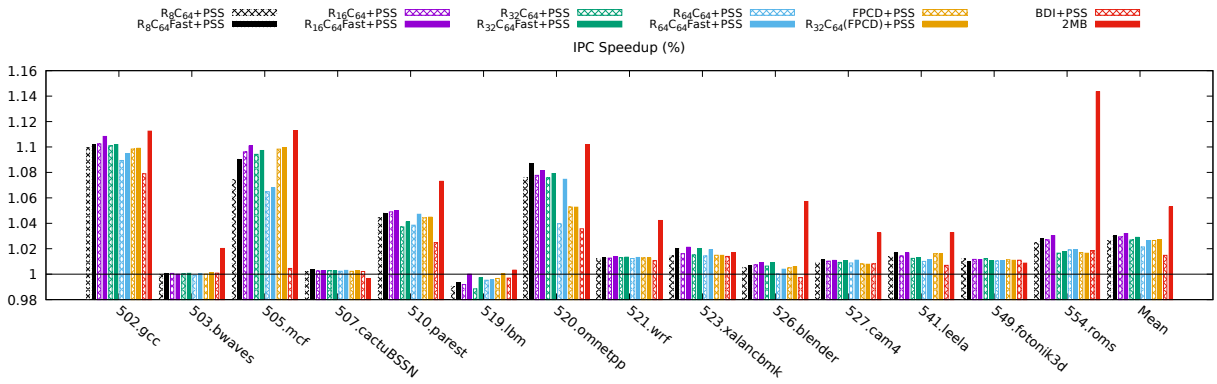
Figure 6.10 – Comparison of the $R_Y C_X$ compressors.

a twice larger uncompressed cache. To reduce visual pollution, and since the best results are achieved with the $R_x C_{64}$ compressors, results are only shown for these configurations.

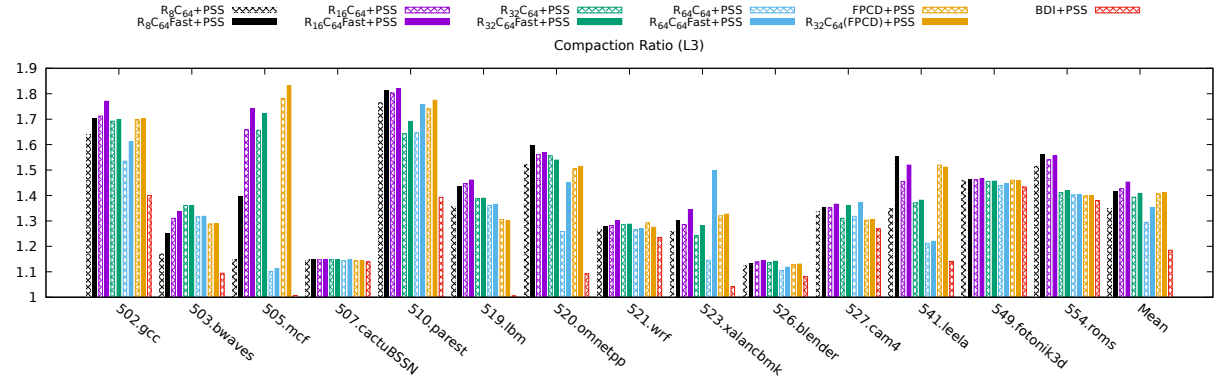
Storing encoding in the tags, although beneficial, can introduce a high area overhead for configurations with multiple regions; therefore, we recommend using this latency improvement in configurations with up to two regions. In any case, one can achieve similar compaction ratio improvements by removing a single delta bit from a few key sub-compressors. For example, turning $C_{64}I_1E_0D_{32}$ into $C_{64}I_1E_0D_{31}$ makes enough room to fit its compressed data and the compressor identification metadata in half a data entry.

6.5.4 Compressor Area overhead

Multiple multi-compressor configurations are listed in Table 6.4, each of which containing a certain number of $C_w I_x E_y D_z$ base-delta compressors. The circuit of any individual $C_w I_x E_y D_z$ is analogous to any base-delta sub-compressor of BDI — $C_w I_x E_y D_z$ is a generalization of base-delta compressors. We have synthesized $SubR_{32}C_{64}$ using Qflow [Edw19] (Region-Chunk concept not applied), and the generated circuit has twice BDI's area.



(a) IPC speedup, normalized to uncompressed baseline.



(b) Compaction ratio.

Figure 6.11 – Comparison of the R_xC_{64} compressors regarding storing the encoding in the tag entry versus in the data entry.

Region-Chunk compression is a conceptual change that merely divides the input among its multiple region compressors. This means that the area of any compressor using the Region-Chunk concept is approximately equal to the sum of the areas of its region compressors. However, region compressors using the Region-Chunk concept are simpler than their non-Region-Chunk counterpart. This is the case because Region-Chunk naturally parallelizes compression and decompression by splitting the input line among its region compressors; therefore, the region compressors themselves can reduce or remove their level of parallelization. For example, C-Pack parses two chunks in parallel per cycle. The same level of parallelization can be achieved with a Region-Chunk compressor using two instances of C-Pack, where each instance parses one chunk per cycle.

In the specific case of base-delta compressors, they achieve 1-cycle latencies by processing all chunks of the line simultaneously. Therefore, instead of a base-delta compressor that parses 64-byte lines using sixteen 32-bit adders operating on two 32-bit values, one can use the Region-Chunk concept to split the input into two 32-byte lines, each parsed by its own region compressor.

Each region compressor would then be composed of sixteen 16-bit adders operating on two 16-bit values. This means that a $R_{32}C_{64}$ encompassing two instances of $SubR_{32}C_{64}$ has 2.5x BDI's area, instead of 4x.

Chapter 7

Conclusion

The memory hierarchy provides a great speedup when compared to direct accesses to the primary or secondary storage; nonetheless, the many memory layers added increase the total system's cost, energy consumption, and bandwidth requirements. In addition to these drawbacks, each level of the hierarchy is limited by its capacity, and the delays of trying to locate missing lines accumulate throughout the levels, adding to the final access latency.

Multiple fields of research address the pitfalls of memory hierarchies, and how to reduce their negative impact. One of these fields is cache and memory compression, which works by detecting duplicated values to reduce the size of stored data. Although compressed memories are able to reduce some of the drawbacks described, they also have their own set of positive and negative aspects. In any case, the benefits of compressed memories, and the increasingly high cost of larger memories have resulted in the slow emergence of a few commercial applications over the last decades.

In Chapter 2 we have thoroughly detailed the challenges of compressed cache designs, and presented the strategies adopted by the literature to tackle these issues. We show that the design of a compressed cache relies on a compound of decisions. The foremost decision is the goal of the compressed system: to increase effective capacity, decrease energy consumption, improve bandwidth, or to reduce the overhead of orthogonal techniques. Although results may overlap among different goals, each of these impacts the system differently, and the rest of the decisions must reflect on this on the objective adopted here.

Then the compression algorithm is selected, as well as the compound of compaction decisions to accommodate the compressed lines, given the system's goal. This means that the cache layout itself must change: for instance, in capacity-focused designs there must be a way to map more blocks into physical data entries. In this case, a solution could be to add more tag entries, but this would imply in a large overhead. Alternative representations have smaller costs, but typically imply sacrificing mapping freedom or compressibility.

Another important consideration is that the compressor should not affect the area budget or the access latency significantly; thus, they need to find a reasonable trade-off between compression efficiency, circuit complexity and decompression speed. Typically, compression algorithms tend to aim for low complexity, moderate compressibility, and some degree of parallelization. Nonetheless, compression can still be a burden for some workloads, even when the decompression

latency is low; therefore, it is essential to provide means to reduce its negative impact. Solutions usually include caching uncompressed data to avoid performing decompressions on the critical path, and dynamically determining if compression should be enabled or disabled, a decision based on the usefulness of compression given the execution characteristics of workloads.

Chapter 3 continues the scrutinization of hardware compression techniques with the challenges added when applying compression to memory and the links between memory levels. When the level being compressed is the memory, the compression algorithms remain valid; however, a few assumptions need to change regarding the layout, and extra challenges arise. The Operating System is agnostic to the cache hierarchy, but relies on the memory size information to determine how pages are managed. In addition, while conventional caches use tags to locate lines, memories typically do not have such structures; therefore, finding compressed blocks and pages requires special handling.

Ultimately, compression is not restrained to be used in a single memory level. Furthermore, even if only a single level is compressed, the moment at which compression and decompression are applied is not enforced to be on the exact time of physical access. Finally, even if no memory level is compressed, compression can still be applied to improve bandwidth or energy consumption. These situations can occur by transferring compressed data between memory levels, and common approaches taken by the literature to handle these so-called link and hierarchy compression methods are also described.

In Chapter 6 we explore the granularity of cache compressors, aiming to increase their efficiency. The goal of the exploration was to better isolate and compress cache lines, and to lower the impact of the workloads' primary data types. This is achieved by the Region-Chunk concept, which further divides cache lines into regions to increase the likelihood of deduplicating data.

The Region-Chunk concept can be applied to any cache compressor; however, it is highly advantageous for base-delta compressors, since they have a limited pattern selection. For this reason we formally define optimizations to base-delta compressors, and select and group these optimized compressors into multi-compressors specialized for Region-Chunk compression. These proposed configurations perform comparably to compressors of higher complexity in terms of both compression efficiency and efficacy, while still keeping the low decompression latency of base-delta compressors.

To take advantage of these improvements in compressibility, in Chapter 5 we propose Pairwise Space Sharing. PSS modifies the size field representation to allow more co-allocations to happen at a lower cost than previous proposals. It co-allocates lines in pairs, so that it can use implicit information to reduce compression-size metadata overhead, increase the number of co-allocation opportunities, and remove the need to re-compact due to small changes in the compressed data's size. Combined, these ideas maximize the compression potential of base-delta compressors; caches using the proposed compressors achieve an average effective capacity of 1.43x, greatly improving

from BDI's 1.18x. Finally, these proposals are concepts; thus they can be applied to most of the existing compressors. When applying both Region-Chunk compression and PSS to the best performing state-of-the-art compressor examined, FPC-D, its compaction factor improved from 1.39x to 1.42x.

7.1 Future Work

This work scrutinizes base-delta compressors to make them efficient for workloads consisting mostly of a single data type. Although out of the scope of this thesis, it would be interesting to analyse which combination of the $C_w I_x E_y D_z$ sub-compressors generates a hybrid that covers any generic workload. This could be further expanded by dynamically choosing (*e.g.*, via set dueling) which compressors should be used based on the workload's behavior.

As stated previously, different data regions respond differently to compression. The results presented in this paper use compressor configurations where all regions in a chunk use the same set of sub-compressors; yet, tailoring sub-compressors based on the regions they belong to could be beneficial. For example, given that MSB tend to have lower variability, the regions that encapsulate such bits could use sub-compressors with fewer bases or smaller deltas. On the other hand, since LSB-related regions likely have higher entropy, they could benefit more from using sub-compressors with more bases or greater deltas.

Analogously, the base-delta compressor can be further generalized by keeping the number of deltas fixed, but reducing it so that it no longer matches the number of chunks. This compressor, besides having the conventional bitmask field to associate a base to each chunk, adds a new bitmask field to inform to which delta a chunk is associated. Studying which configurations of number of bases and number of deltas are sufficient to compress cache lines may allow further improving compression efficiency.

BPC [KSCE16] is a technique that further enhances the compressibility of base-delta compressors by adding bitwise operations to reduce entropy. As such, it can be applied on top of the new multi-base compressors whose delta size is not 0. This would increase their decompression latency, but their efficiency would also be increased.

The stride compressor presented here was fairly simple. Stride schemes have been explored for decades, and some of the proposed ideas can be adapted to upgrade this compressor. Furthermore, stride is just one of the common mathematical relationships between values in a cache line, and further equations could be explored. For example, workloads may exhibit multiple strides relative to different values; thus, the stride compressor's could be increased by using more bases and deltas. In addition, one can add a bitmask to inform to which chunks a stride should be applied.

Finally, the addition of regions makes single-cycle-decompression compressors reach compet-

itive levels; therefore, caches closer to the core can perceive less the impact of the decompression step, and a fully-compressed memory hierarchy becomes more feasible.

7.2 Final Remarks

Compressed systems have advanced throughout the years, requiring lower metadata overhead, and decompression cycles than earlier approaches; however, despite all the advancements, their average improvement of the effective capacity, in general, is still fairly low when compared to their theoretical upper limit. The typical hardware data-compression algorithm is a simplification of dictionary-based data-compression algorithms; yet, software data-compression is a matured topic that can be looked upon to further improve hardware proposals.

Finally, compression has the ability to enhance levels in the memory hierarchy, and its potential further increases when multiple levels are compressed. A completely compressed memory hierarchy is the holy grail of hardware compression. A slim selection of previous works partially tackle the idea of a completely compressed memory hierarchy; however, many challenges still remain for its adoption by industry. Different levels do not abide by the same restraints — *e.g.*, the impact of the decompression latency varies between levels — so when designing such systems it is important to be flexible enough to attain a good per-level trade-off between compressibility and speed, while still maintaining some kind of compressor compatibility between different memory levels.

Bibliography

[AA18] Alaa R. Alameldeen and Rajat Agarwal. Opportunistic compression for direct-mapped dram caches. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '18, page 129–136, Alexandria, Virginia, USA, 2018. Association for Computing Machinery.

[AB19] ZeroPoint Technologies AB. Zeropoint's ziptilion ip-block family for transparent expansion of memory capacity and improvement of effective bandwidth. Technical report, ZeroPoint Technologies AB, Falkenbergsgatan 3, 412 85 Göteborg, Sweden, August 2019.

[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*, MICRO-48, page 38–49, Waikiki, Hawaii, 2015. Association for Computing Machinery.

[AF00] Bulent Abali and Hubertus Franke. Operating system support for fast hardware compression of main memory contents. In *Workshop on Solving the Memory Wall Problem*, 2000.

[AFS+01] Bulent Abali, Hubertus Franke, Xiaowei Shen, Dan E Poff, and T Basil Smith. Performance of hardware compressed main memory. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 73–81. IEEE, 2001.

[AGN+15] Chloe Alverti, Georgios Goumas, Konstantinos Nikas, Angelos Arelakis, Nectarios Koziris, and Per Stenström. Memory link compression to speedup scientific workloads. In *8th Workshop on Programmability Issues for Heterogeneous Multicores*, Amsterdam, Netherlands, 2015.

[AInVnL13] Jorge Albericio, Pablo Ibáñez, Víctor Viñals, and José M. Llabería. The reuse cache: Downsizing the shared last-level cache. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, page 310–321, Davis, California, 2013. Association for Computing Machinery.

[ALSW18] Akhil Arunkumar, Shin-Ying Lee, Vignesh Soundararajan, and Carole-Jean Wu. Latte-cc: Latency tolerance aware adaptive cache compression management for

energy efficient gpus. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 221–234, Vienna, Austria, 2018. IEEE, IEEE Computer Society.

[ALYK12] Baik Song An, Manhee Lee, Ki Hwan Yum, and Eun Jung Kim. Efficient data packet compression for cache coherent multiprocessor systems. In *2012 Data Compression Conference*, pages 129–138. IEEE, 2012.

[AP93] Anant Agarwal and Stephen D. Pudar. *Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches*. ISCA '93. Association for Computing Machinery, San Diego, California, USA, 1993.

[AS14a] Angelos Arelakis and Per Stenstrom. A case for a value-aware cache. *Computer Architecture Letters*, 13(1):1–4, 2014.

[AS14b] Angelos Arelakis and Per Stenstrom. Sc2: A statistical compression cache scheme. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, page 145–156, Minneapolis, Minnesota, USA, 2014. IEEE Press.

[ATGK07] Ali-Reza Adl-Tabatabai, Anwar M. Ghuloum, and Shobhit O Kanaujia. Compression in cache design. In *Proceedings of the 21st Annual International Conference on Supercomputing*, ICS '07, pages 190–201, Seattle, Washington, 2007. ACM.

[AW04a] Alaa R. Alameldeen and David A. Wood. Adaptive cache compression for high-performance processors. *SIGARCH Comput. Archit. News*, 32(2):212, March 2004.

[AW04b] Alaa R Alameldeen and David A Wood. Frequent pattern compression: A significance-based compression scheme for l2 caches. *Dept. Comp. Scie., Univ. Wisconsin-Madison, Tech. Rep*, 1500, 2004.

[AW07] Alaa R Alameldeen and David A Wood. Interactions between compression and prefetching in chip multiprocessors. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 228–239, Scottsdale, AZ, USA, 2007. IEEE, IEEE.

[AYK01] Edward Ahn, Seung-Moon Yoo, and Sung-Mo Steve Kang. Effective algorithms for cache-level compression. In *Proceedings of the 11th Great Lakes Symposium on VLSI*, GLSVLSI '01, pages 89–92, West Lafayette, Indiana, USA, 2001. ACM.

[BBB+11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D.

Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[BBMM02] L. Benini, D. Bruni, A. Macii, and E. Macii. Hardware-assisted data compression for energy minimization in systems with embedded processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '02, page 449, Paris, France, 2002. IEEE Computer Society.

[BDMM+98] Luca Benini, Giovanni De Micheli, Enrico Macii, Donatella Sciuto, and Cristina Silvano. Address bus encoding techniques for system-level power optimization. In *Proceedings Design, Automation and Test in Europe*, pages 861–866. IEEE, 1998.

[BFG+03] Árpád Beszédes, Rudolf Ferenc, Tibor Gyimóthy, André Dolenc, and Konsta Karsisto. Survey of code-size reduction methods. *ACM Comput. Surv.*, 35(3):223–267, September 2003.

[BFR01] Caroline D Benveniste, Peter A Franaszek, and John T Robinson. Cache-memory interfaces in compressed memory systems. *IEEE Transactions on computers*, 50(11):1106–1116, 2001.

[BH09] Luiz André Barroso and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 4(1):1–108, 2009.

[BL17] Abhishek Bhattacharjee and Daniel Lustig. Architectural and operating system support for virtual memory. *Synthesis Lectures on Computer Architecture*, 12(5):1–175, 2017.

[BLN+13] Seungcheol Baek, Hyung Gyu Lee, Chrysostomos Nicopoulos, Junghee Lee, and Jongman Kim. Ecm: Effective capacity maximizer for high-performance compressed caching. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on High-Performance Computer Architecture*, pages 131–142, Shenzhen, China, 2013. IEEE, IEEE Computer Society.

[BLNK14] Seungcheol Baek, Hyung Gyu Lee, Chrysostomos Nicopoulos, and Jongman Kim. Designing hybrid dram/pcm main memory systems utilizing dual-phase compression. *ACM Trans. Des. Autom. Electron. Syst.*, 20(1), November 2014.

[BR10] Martin Burtscher and Paruj Ratanaworabhan. gfpc: A self-tuning compression algorithm. In *Data Compression Conference (DCC), 2010*, pages 396–405, Snowbird, UT, USA, 2010. IEEE, IEEE Computer Society.

[CCZ13] Long Chen, Yanan Cao, and Zhao Zhang. Free ecc: An efficient error protection for compressed last-level caches. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 278–285, Asheville, NC, USA, 2013. IEEE, IEEE Computer Society.

[CD73] Edward Grady Coffman and Peter J Denning. *Operating systems theory*, volume 973. prentice-Hall Englewood Cliffs, NJ, 1973.

[CEA18] Esha Choukse, Mattan Erez, and Alaa R. Alameldeen. Compresso: Pragmatic main memory compression. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, page 546–558, Fukuoka, Japan, 2018. IEEE Press.

[CGS00] Ramon Canal, Antonio González, and James E. Smith. Very low power pipelines using significance compression. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, pages 181–190, Monterey, California, USA, 2000. ACM.

[CH84] Chin-Long Chen and MY Hsiao. Error-correcting codes for semiconductor memory applications: A state-of-the-art review. *IBM Journal of Research and Development*, 28(2):124–134, 1984.

[CHE11] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, Seattle, Washington, 2011. Association for Computing Machinery.

[CKD+10] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Cache hierarchy and memory subsystem of the amd opteron processor. *IEEE micro*, 30(2):16–29, 2010.

[CKH+16] Kevin K. Chang, Abhijith Kashyap, Hasan Hassan, Saugata Ghose, Kevin Hsieh, Donghyuk Lee, Tianshi Li, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. Understanding latency variation in modern dram chips: Experimental characterization, analysis, and optimization. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, SIGMETRICS '16, page 323–336, Antibes Juan-les-Pins, France, 2016. Association for Computing Machinery.

[Cor17] S. P. E. Corporation. Spec cpu 2017. `https://www.spec.org/cpu2017/`, 2017. Accessed: 2019-10-10.

[CPR03] David Chen, Enoch Peserico, and Larry Rudolph. A dynamically partitionable compressed cache. In *Proceedings of the Singapore-MIT Alliance Symposium*, 2003.

[CR95] Daniel Citron and Larry Rudolph. Creating a wider bus using caching techniques. In *High-Performance Computer Architecture, 1995. Proceedings., First IEEE Symposium on*, pages 90–99. IEEE, 1995.

[CS21] Daniel R. Carvalho and André Seznec. Understanding cache compression. *Transactions on Architecture and Code Optimization (TACO)*, 2021.

[CSO+19] Esha Choukse, Michael Sullivan, Mike O'Connor, Mattan Erez, Jeff Pool, David Nellans, and Steve Keckler. Buddy compression: Enabling larger memory for deep learning and hpc workloads on gpus. *arXiv preprint arXiv:1903.02596*, 2019.

[CYD+10] Xi Chen, Lei Yang, Robert P Dick, Li Shang, and Haris Lekatsas. C-pack: A high-performance microprocessor cache compression algorithm. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 18(8):1196–1208, 2010.

[DKL+17] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. Inside 6th-generation intel core: New microarchitecture code-named skylake. *IEEE Micro*, 37(2):52–62, 2017.

[DPS09] Julien Dusser, Thomas Piquet, and André Seznec. Zero-content augmented caches. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, page 46–55, Yorktown Heights, NY, USA, 2009. Association for Computing Machinery.

[Dre07] Ulrich Drepper. What every programmer should know about memory. *Red Hat, Inc*, 11:2007, 2007.

[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*, HiPEAC '11, page 77–86, Heraklion, Greece, 2011. Association for Computing Machinery.

[DZC+13] Yu Du, Miao Zhou, Bruce Childers, Rami Melhem, and Daniel Mossé. Delta-compressed caching for overcoming the write bandwidth limitation of hybrid main memory. *ACM Trans. Archit. Code Optim.*, 9(4), January 2013.

[Edw19] R. Timothy Edwards. Qflow. `http://opencircuitdesign.com/qflow/reference.html`, 2019. Accessed: 2020-10-07.

[EEF+97] Jens Ernst, William Evans, Christopher W. Fraser, Todd A. Proebsting, and Steven Lucco. Code compression. *SIGPLAN Not.*, 32(5):358–365, May 1997.

[ES05] Magnus Ekman and Per Stenstrom. A robust main-memory compression scheme. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA '05, page 74–85, Madison, Wisconsin, USA, 2005. IEEE Computer Society.

[FHPR01] Peter A Franaszek, Philip Heidelberger, Dan E Poff, and John T Robinson. Algorithms and data structures for compressed-memory machines. *IBM Journal of Research and Development*, 45(2):245–258, 2001.

[FHW99] Peter A. Franaszek, Philip Heidelberger, and Michael Wazlowski. On management of free space in compressed memory systems. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '99, page 113–121, Atlanta, Georgia, USA, 1999. Association for Computing Machinery.

[FP91] Matthew Farrens and Arvin Park. Dynamic base register caching: A technique for reducing address bus width. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ISCA '91, page 128–137, Toronto, Ontario, Canada, 1991. Association for Computing Machinery.

[FR01] Peter A Franaszek and John T Robinson. On internal organization in compressed random-access memories. *IBM Journal of Research and Development*, 45(2):259–270, 2001.

[FRT96] P. Franaszek, J. Robinson, and J. Thomas. Parallel compression with cooperative dictionary construction. In *Proceedings of the Conference on Data Compression*, DCC '96, pages 200–, Washington, DC, USA, 1996. IEEE Computer Society.

[FSGAB+16] Alexandra Ferreron, Dario Suarez-Gracia, Jesus Alastruey-Benede, Teresa Monreal-Arnal, and Pablo Ibanez. Concertina: Squeezing in cache content to operate at near-threshold voltage. *IEEE Transactions on Computers*, 65(3):755–769, March 2016.

[GAS16] Jayesh Gaur, Alaa R Alameldeen, and Sreenivas Subramoney. Base-victim compression: an opportunistic cache compression architecture. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 317–328, Seoul, Republic of Korea, 2016. IEEE, IEEE Press.

[GH96] Ricardo Gonzalez and Mark Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of solid-state circuits*, 31(9):1277–1284, 1996.

[GH09] James R Goodman and Herbert Hing Jing Hum. Mesif: A two-hop cache coherency protocol for point-to-point interconnects. *University of Auckland, Tech. Rep*, 2009.

[GHZ18] Yuncheng Guo, Yu Hua, and Pengfei Zuo. Dfpc: A dynamic frequent pattern compression scheme in nvm-based main memory. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1622–1627, Dresden, Germany, 2018. IEEE.

[GNL20] Amin Ghasemazar, Prashant Nair, and Mieszko Lis. Thesaurus: Efficient cache compression via dynamic clustering. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 527–540, Lausanne, Switzerland, 2020. Association for Computing Machinery.

[H+52] David A Huffman et al. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[HAB+19] Seokin Hong, Bulent Abali, Alper Buyuktosunoglu, Michael B. Healy, and Prashant J. Nair. Touché: Towards ideal and efficient cache compression by mitigating tag area overheads. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 453–465, Columbus, OH, USA, 2019. Association for Computing Machinery.

[HKO+14] Per Hammarlund, Rajesh Kumar, Randy B Osborne, Ravi Rajwar, Ronak Singhal, Reynold D'Sa, Robert Chappell, Shiv Kaushik, Srinivas Chennupaty, Stephan Jourdan, et al. Haswell: The fourth-generation intel core processor. *IEEE Micro*, 34(2):6–20, 2014.

[HNA+18] Seokin Hong, Prashant J. Nair, Bulent Abali, Alper Buyuktosunoglu, Kyu-Hyoun Kim, and Michael B. Healy. Attaché: Towards ideal memory compression by mitigating metadata bandwidth overheads. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, page 326–338, Fukuoka, Japan, 2018. IEEE Press.

[HP12] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[HPV+16] Hasan Hassan, Gennady Pekhimenko, Nandita Vijaykumar, Vivek Seshadri, Donghyuk Lee, Oguz Ergin, and Onur Mutlu. Chargecache: Reducing dram la-

tency by exploiting row access locality. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 581–593. IEEE, 2016.

[HR00] Erik G. Hallnor and Steven K. Reinhardt. *A Fully Associative Software-Managed Cache Design*. ISCA '00. Association for Computing Machinery, Vancouver, British Columbia, Canada, 2000.

[HR04] Erik G. Hallnor and Steven K. Reinhardt. A compressed memory hierarchy using an indirect index cache. In *Proceedings of the 3rd Workshop on Memory Performance Issues: In Conjunction with the 31st International Symposium on Computer Architecture*, WMPI '04, page 9–15, Munich, Germany, 2004. Association for Computing Machinery.

[HR05] Erik G Hallnor and Steven K Reinhardt. A unified compressed memory hierarchy. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 201–212, San Francisco, CA, USA, 2005. IEEE, IEEE Computer Society.

[HSPE08] Allan Hartstein, Vijayalakshmi Srinivasan, T Puzak, and P Emma. On the nature of cache miss behavior: Is it$\sqrt{2}$. *The Journal of Instruction-Level Parallelism*, 10:1–22, 2008.

[IBM] IBM. Active memory expansion. `https://www.ibm.com/support/knowledgecenter/en/ssw_aix_71/performance/intro_ame_process.html`. Accessed: 2020-10-23.

[Int17] Intel. 5-level paging and 5-level ept. Technical report, Intel, May 2017.

[JL16] Akanksha Jain and Calvin Lin. Back to the future: Leveraging belady's algorithm for improved cache replacement. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, page 78–89, Seoul, Republic of Korea, 2016. IEEE Press.

[JL19] Akanksha Jain and Calvin Lin. Cache replacement policies. *Synthesis Lectures on Computer Architecture*, 14(1):1–87, 2019.

[Jou90] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, page 364–373, Seattle, Washington, USA, 1990. Association for Computing Machinery.

[JT17] Daniel A. Jiménez and Elvira Teran. Multiperspective reuse prediction. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 436–448, Cambridge, Massachusetts, 2017. Association for Computing Machinery.

[JTP21] Uthayakumar Jayasankar, Vengattaraman Thirumal, and Dhavachelvan Ponnurangam. A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications. *Journal of King Saud University - Computer and Information Sciences*, 33(2):119–140, 2021.

[JTSE10] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, page 60–71, Saint-Malo, France, 2010. Association for Computing Machinery.

[JWN10] Bruce Jacob, David Wang, and Spencer Ng. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2010.

[JZZ+12] Lei Jiang, Bo Zhao, Youtao Zhang, Jun Yang, and Bruce R Childers. Improving write operations in mlc phase change memory. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 201–210, New Orleans, LA, USA, 2012. IEEE, IEEE Computer Society.

[KAK06] Georgios Keramidas, Konstantinos Aisopos, and Stefanos Kaxiras. Dynamic dictionary-based data compression for level-1 caches. *Architecture of Computing Systems-ARCS 2006*, 3894:114–129, 2006.

[KAM02] N Kim, Todd Austin, and Trevor Mudge. Low-energy data cache using sign compression and cache line bisection. In *Proceedings of the 2nd Annual Workshop on Memory Performance Issues (WMPI'02)*, 2002.

[Kel02] John Kelsey. Compression and information leakage of plaintext. In *International Workshop on Fast Software Encryption*, volume 2365 of *Lecture Notes in Computer Science*, pages 263–276, Leuven, Belgium, 2002. Springer, Springer.

[KGJ96] Morten Kjelso, Mark Gooch, and Simon Jones. Design and performance of a main memory hardware data compressor. In *EUROMICRO 96. Beyond 2000: Hardware and Software Design Strategies., Proceedings of the 22nd EUROMICRO Conference*, pages 423–430, Prague, Czech Republic, 1996. IEEE, IEEE Computer Society.

[KGJ98] M. Kjelso, M. Gooch, and S. Jones. Empirical study of memory-data: characteristics and compressibility. *IEE Proceedings - Computers and Digital Techniques*, 145(1):63–67, 1998.

[KI03] Krishna Kant and Ravi Iyer. Design and performance of compressed interconnects for high performance servers. In *Computer Design, 2003. Proceedings. 21st International Conference on*, pages 164–169. IEEE, 2003.

[KL13] Mushfique Junayed Khurshid and Mikko Lipasti. Data compression for thermal mitigation in the hybrid memory cube. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 185–192, Asheville, NC, USA, 2013. IEEE, IEEE Computer Society.

[KLKH11] Soontae Kim, Jongmin Lee, Jesung Kim, and Seokin Hong. Residue cache: A low-energy low-area l2 cache architecture via compression and partial hits. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 420–429, Porto Alegre, Brazil, 2011. ACM.

[KPA04] Sumeet Kumar, Prateek Pujara, and Aneesh Aggarwal. Bit-sliced datapath for energy-efficient high performance microprocessors. In *International Workshop on Power-Aware Computer Systems*, volume 3471 of *Lecture Notes in Computer Science*, pages 30–45, Portland, OR, USA, 2004. Springer, Springer.

[KPM15] Raghavendra K, Biswabandan Panda, and Madhu Mutyam. Pbc: Prefetched blocks compaction. *IEEE Transactions on Computers*, 65:1–1, 01 2015.

[KPM17] Raghavendra Kanakagiri, Biswabandan Panda, and Madhu Mutyam. Mbzip: Multiblock data compression. *ACM Trans. Archit. Code Optim.*, 14(4), December 2017.

[KSCE16] Jungrae Kim, Michael Sullivan, Esha Choukse, and Mattan Erez. Bit-plane compression: Transforming data for better compression in many-core architectures. *SIGARCH Comput. Archit. News*, 44(3):329–340, June 2016.

[KSGE15] Jungrae Kim, Michael Sullivan, Seong-Lyong Gong, and Mattan Erez. Frugal ecc: Efficient and versatile memory error protection through fine-grained compression. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, Austin, Texas, 2015. Association for Computing Machinery.

[KTJ10] Samira Manabi Khan, Yingying Tian, and Daniel A. Jimenez. Sampling dead block prediction for last-level caches. In *Proceedings of the 2010 43rd Annual*

IEEE/ACM International Symposium on Microarchitecture, MICRO '43, page 175–186, Atlanta, Georgia, USA, 2010. IEEE Computer Society.

[LH11] Gabriel H. Loh and Mark D. Hill. Efficiently enabling conventional block sizes for very large die-stacked dram caches. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, page 454–464, Porto Alegre, Brazil, 2011. Association for Computing Machinery.

[LHK99] Jang-Soo Lee, Won-Kee Hong, and Shin-Dug Kim. Design and evaluation of a selective compressed memory system. In *Computer Design, 1999.(ICCD'99) International Conference on*, pages 184–191, Austin, Texas, USA, 1999. IEEE, IEEE Computer Society.

[LHK00] Jang-Soo Lee, Won-Kee Hong, and Shin-Dug Kim. An on-chip cache compression technique to reduce decompression overhead and design complexity. *Journal of systems Architecture*, 46(15):1365–1382, 2000.

[LI06] Peter Lindstrom and Martin Isenburg. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1245–1250, September 2006.

[LKP+15] Donghyuk Lee, Yoongu Kim, Gennady Pekhimenko, Samira Khan, Vivek Seshadri, Kevin Chang, and Onur Mutlu. Adaptive-latency dram: Optimizing dram timing for the common-case. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 489–501. IEEE, 2015.

[LLG+90] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. *The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor*. ISCA '90. Association for Computing Machinery, Seattle, Washington, USA, 1990.

[MAMJ15] Joshua San Miguel, Jorge Albericio, Andreas Moshovos, and Natalie Enright Jerger. Doppelgänger: A cache for approximate computing. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 50–61, Waikiki, Hawaii, 2015. Association for Computing Machinery.

[Mär14] Christian Märtin. Multicore processors: challenges, opportunities, emerging trends. In *Proc. Embedded World Conference*, pages 1–9, 2014.

[MGST70] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[MHS12] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, July 2012.

[Mic16a] Pierre Michaud. Best-offset hardware prefetching. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 469–480. IEEE, 2016.

[Mic16b] Pierre Michaud. Some mathematical facts about optimal cache replacement. *ACM Trans. Archit. Code Optim.*, 13(4), December 2016.

[Mit16] Sparsh Mittal. A survey of techniques for approximate computing. *ACM Comput. Surv.*, 48(4), March 2016.

[MKG98] Srilatha Manne, Artur Klauser, and Dirk Grunwald. Pipeline gating: Speculation control for energy reduction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ISCA '98, page 132–141, Barcelona, Spain, 1998. IEEE Computer Society.

[MN19] Sparsh Mittal and Subhrajit Nag. A survey of encoding techniques for reducing data-movement energy. *Journal of Systems Architecture*, 97:373–396, 2019.

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

[NFBJ99] Jose Luis Nunez, Claudia Feregrino, Stephen Bateman, and Simon Jones. The x-matchlite fpga-based data compressor. In *EUROMICRO Conference, 1999. Proceedings. 25th*, volume 1, pages 1126–1132, Milan, Italy, 1999. IEEE, IEEE Computer Society.

[NFJB01] Jose Luis Nunez, Claudia Feregrino, Simon Jones, and Stephen Bateman. X-matchpro: A proasic-based 200 mbytes/s full-duplex lossless data compressor. In *International Conference on Field Programmable Logic and Applications*, volume 2147 of *Lecture Notes in Computer Science*, pages 613–617, Belfast, Northern Ireland, UK, 2001. Springer, Springer.

[NFW18] Tri M. Nguyen, Adi Fuchs, and David Wentzlaff. Cable: A cache-based link encoder for bandwidth-starved manycores. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, page 312–325, Fukuoka, Japan, 2018. IEEE Press.

[NW15] Tri M. Nguyen and David Wentzlaff. Morc: A manycore-oriented compressed cache. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, page 76–88, Waikiki, Hawaii, 2015. Association for Computing Machinery.

[Oln85] Howard T Olnowich. Set associative sector cache, January 8 1985. US Patent 4,493,026.

[Ost10] Igor Ostrovsky. Gallery of processor cache effects. `http://igoro.com/archive/gallery-of-processor-cache-effects/`, 2010. Accessed: 2020-10-16.

[PA05] Prateek Pujara and Aneesh Aggarwal. Restrictive compression techniques to increase level 1 cache capacity. In *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on*, pages 327–333, San Jose, CA, USA, 2005. IEEE, IEEE Computer Society.

[PBL+17] Jaehyun Park, Seungcheol Baek, Hyung Gyu Lee, Chrysostomos Nicopoulos, Vinson Young, Junghee Lee, and Jongman Kim. Hope: Hot-cacheline prediction for dynamic early decompression in compressed llcs. *ACM Trans. Des. Autom. Electron. Syst.*, 22(3), April 2017.

[PBV+16] Gennady Pekhimenko, Evgeny Bolotin, Nandita Vijaykumar, Onur Mutlu, Todd C Mowry, and Stephen W Keckler. A case for toggle-aware compression for gpu systems. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 188–200. IEEE, 2016.

[PHC+15] Gennady Pekhimenko, Tyler Huberty, Rui Cai, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. Exploiting compressed block size as an indicator of future reuse. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 51–63, Burlingame, CA, USA, 2015. IEEE, IEEE Computer Society.

[PHM15] Bhargavraj Patel, Nikos Hardavellas, and Gokhan Memik. Scp: Synergistic cache compression and prefetching. In *2015, 33rd IEEE International Conference on Computer Design (ICCD)*, pages 164–171, New York City, NY, USA, 2015. IEEE, IEEE Computer Society.

[PKL15] David J. Palframan, Nam Sung Kim, and Mikko H. Lipasti. Cop: To compress and protect main memory. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 682–693, Portland, Oregon, 2015. Association for Computing Machinery.

[PM17] Poovaiah M. Palangappa and Kartik Mohanram. Compex++: Compression-expansion coding for energy, latency, and lifetime improvements in mlc/tlc nvms. *ACM Trans. Archit. Code Optim.*, 14(1):10:1–10:30, April 2017.

[PP84] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, ISCA '84, page 348–354, New York, NY, USA, 1984. Association for Computing Machinery.

[PS16] Biswabandan Panda and André Seznec. Dictionary sharing: An efficient cache compression scheme for compressed caches. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, Taipei, Taiwan, 2016. IEEE Press.

[PS18] Biswabandan Panda and André Seznec. Synergistic cache layout for reuse and compression. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, Limassol, Cyprus, 2018. Association for Computing Machinery.

[PSK+13] Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Linearly compressed pages: A low-complexity, low-latency main memory compression framework. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, page 172–184, Davis, California, 2013. Association for Computing Machinery.

[PSM+12] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Base-delta-immediate compression: Practical data compression for on-chip caches. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, page 377–388, Minneapolis, Minnesota, USA, 2012. Association for Computing Machinery.

[QL12] Moinuddin K. Qureshi and Gabe H. Loh. Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, page 235–246, Vancouver, B.C., CANADA, 2012. IEEE Computer Society.

[QSP07] Moinuddin K Qureshi, M Aater Suleman, and Yale N Patt. Line distillation: Increasing cache capacity by filtering unused words in cache lines. In *High Perfor-*

mance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on, pages 250–259, Phoenix, Arizona, USA, 2007. IEEE, IEEE Computer Society.

[QT17] Inc. Qualcomm Technologies. Architecting a multi-core server soc for the cloud. Technical report, Qualcomm Technologies, Inc, 5775 Morehouse Dr., San Diego CA 92121, USA, October 2017.

[RKB+09] Brian M. Rogers, Anil Krishna, Gordon B. Bell, Ken Vu, Xiaowei Jiang, and Yan Solihin. Scaling the bandwidth wall: Challenges in and avenues for cmp scaling. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 371–382, Austin, TX, USA, 2009. ACM.

[RKP01] Sumit Roy, Raj Kumar, and Milos Prvulovic. Improving system performance with compressed memory. In *Parallel and Distributed Processing Symposium., Proceedings 15th International*, pages 7–pp. IEEE, 2001.

[Sal04] David Salomon. *Data Compression: The Complete Reference, 3rd Edition.* Springer, 2004.

[Sal07] David Salomon. *Variable-length Codes for Data Compression.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[SASW15] Somayeh Sardashti, Angelos Arelakis, Per Stenström, and David A Wood. A primer on compression in the memory hierarchy. *Synthesis Lectures on Computer Architecture*, 10(5):1–86, 2015.

[SAYN09] Dinesh C Suresh, Banit Agrawal, Jun Yang, and Walid A Najjar. Tunable and energy efficient bus encoding techniques. *IEEE Transactions on Computers*, 58(8):1049–1062, 2009.

[SB95] Mircea R Stan and Wayne P Burleson. Bus-invert coding for low-power i/o. *IEEE Transactions on very large scale integration (VLSI) systems*, 3(1):49–58, 1995.

[Sez93] André Seznec. A case for two-way skewed-associative caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, page 169–178, San Diego, California, USA, 1993. Association for Computing Machinery.

[Sez94] A. Seznec. Decoupled sectored caches: Conciliating low tag implementation cost. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, ISCA '94, page 384–393, Chicago, Illinois, USA, 1994. IEEE Computer Society Press.

[SGGS98] Abraham Silberschatz, Peter B Galvin, Greg Gagne, and A Silberschatz. *Operating system concepts*, volume 4. Addison-Wesley Reading, 1998.

[SGSB99] Kenneth James Schultz, Garnet Frederick Randall Gibson, Farhad Shafai, and Armin George Bluschke. Content addressable memory, January 12 1999. US Patent 5,859,791.

[SHW11] Daniel J Sorin, Mark D Hill, and David A Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011.

[SK13] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 475–486, Tel-Aviv, Israel, 2013. ACM.

[SM08] Seok-Won Seong and Prabhat Mishra. Bitmask-based code compression for embedded systems. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 27(4):673–685, 2008.

[SMAJJ16] Joshua San Miguel, Jorge Albericio, Natalie Enright Jerger, and Aamer Jaleel. The bunker cache for spatio-value approximation. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 43:1–43:12, Taipei, Taiwan, 2016. IEEE, IEEE Computer Society.

[Smi82] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, September 1982.

[SPHC02] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, page 45–57, San Jose, California, 2002. Association for Computing Machinery.

[SS82] James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, October 1982.

[SS86] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the ieee futurebus. *SIGARCH Comput. Archit. News*, 14(2):414–423, May 1986.

[SSK12] Vijay Sathish, Michael J Schulte, and Nam Sung Kim. Lossless and lossy memory i/o link compression for improving performance of gpgpu workloads. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 325–334. IEEE, 2012.

[SSW14] Somayeh Sardashti, André Seznec, and David A. Wood. Skewed compressed caches. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, page 331–342, Cambridge, United Kingdom, 2014. IEEE Computer Society.

[SSW16] Somayeh Sardashti, Andre Seznec, and David A. Wood. Yet another compressed cache: A low-cost yet effective compressed cache. *ACM Trans. Archit. Code Optim.*, 13(3), September 2016.

[STBD14] Ali Shafiee, Meysam Taassori, Rajeev Balasubramonian, and Al Davis. Memzip: Exploring unconventional benefits from memory compression. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 638–649, Orlando, FL, USA, 2014. IEEE, IEEE Computer Society.

[STD94] C-L Su, C-Y Tsui, and Alvin M Despain. Saving power in the control path of embedded processors. *IEEE Design & Test of Computers*, 11(4):24–31, 1994.

[Ste90] Per Stenstrom. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, 1990.

[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*, MICRO-46, page 62–73, Davis, California, 2013. Association for Computing Machinery.

[SW17] Somayeh Sardashti and David A. Wood. Could compression be of general use? evaluating memory compression across domains. *ACM Trans. Archit. Code Optim.*, 14(4):44:1–44:24, December 2017.

[SWAF09] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-temporal memory streaming. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, page 69–80, Austin, TX, USA, 2009. Association for Computing Machinery.

[TFR+01] R Brett Tremaine, Peter A Franaszek, John T Robinson, Charles O Schulz, T Basil Smith, Michael E Wazlowski, and P Maurice Bland. Ibm memory expansion

technology (mxt). *IBM Journal of Research and Development*, 45(2):271–285, 2001.

[TG05] Irina Chihaia Tuduce and Thomas R Gross. Adaptive main memory compression. In *USENIX Annual Technical Conference, General Track*, pages 237–250, Anaheim, CA, USA, 2005. USENIX.

[TGS18] Po-An Tsai, Yee Ling Gan, and Daniel Sanchez. Rethinking the memory hierarchy for modern languages. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, page 203–216, Fukuoka, Japan, 2018. IEEE Press.

[TKJL14] Yingying Tian, Samira M. Khan, Daniel A. Jiménez, and Gabriel H. Loh. Last-level cache deduplication. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, page 53–62, Munich, Germany, 2014. Association for Computing Machinery.

[TOP19] TOP500. Top500. `https://www.top500.org/lists/`, 2019. Accessed: 2019-12-26.

[TS07] Tian Tian and Chiu-Pi Shih. Software techniques for shared-cache multi-core systems. *Intel Software Network*, 2007.

[TS19] Po-An Tsai and Daniel Sanchez. Compress objects, not cache lines: An object-based compressed memory hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 229–242, Providence, RI, USA, 2019. ACM.

[TSFS20] Po-An Tsai, Andres Sanchez, Christopher W. Fletcher, and Daniel Sanchez. Safecracker: Leaking secrets through compressed caches. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1125–1140, Lausanne, Switzerland, 2020. Association for Computing Machinery.

[TSS08] Martin Thuresson, Lawrence Spracklen, and Per Stenstrom. Memory-link compression schemes: A value locality perspective. *IEEE Transactions on Computers*, 57(7):916–927, 2008.

[TSW+01] R Brett Tremaine, T Basil Smith, Mike Wazlowski, David Har, Kwok-Ken Mak, and Sujith Arramreddy. Pinnacle: Ibm mxt in a memory controller chip. *IEEE Micro*, 21(2):56–68, 2001.

[TZ07] Xinhua Tian and Minxuan Zhang. A unified compressed cache hierarchy using simple frequent pattern compression and partial cache line prefetching. In *International Conference on Embedded Software and Systems*, volume 4523 of *Lecture Notes in Computer Science*, pages 142–153, Daegu, Republic of Korea, 2007. Springer, Springer.

[USCM16] Hiroyuki Usui, Lavanya Subramanian, Kevin Kai-Wei Chang, and Onur Mutlu. Dash: Deadline-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators. *ACM Trans. Archit. Code Optim.*, 12(4), January 2016.

[VKI+00] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye. Energy-driven integrated hardware-software optimizations using simplepower. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, page 95–106, Vancouver, British Columbia, Canada, 2000. Association for Computing Machinery.

[VPJ+15] Nandita Vijaykumar, Gennady Pekhimenko, Adwait Jog, Abhishek Bhowmick, Rachata Ausavarungnirun, Chita Das, Mahmut Kandemir, Todd C. Mowry, and Onur Mutlu. A case for core-assisted bottleneck acceleration in gpus: Enabling flexible data compression with assist warps. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 41–53, Portland, Oregon, 2015. Association for Computing Machinery.

[VZA00] Luis Villa, Michael Zhang, and Krste Asanović. Dynamic zero compression for cache energy reduction. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, page 214–220, Monterey, California, USA, 2000. Association for Computing Machinery.

[Wal02] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, December 2002.

[Wel84] Terry A. Welch. A technique for high-performance data compression. *Computer*, 6(17):8–19, 1984.

[WJH+11] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, and Joel Emer. Ship: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, page 430–441, Porto Alegre, Brazil, 2011. Association for Computing Machinery.

[WKS99] Paul R Wilson, Scott F Kaplan, and Yannis Smaragdakis. The case for compressed caching in virtual memory systems. In *USENIX Annual Technical Conference, General Track*, pages 101–116, Monterey, California, USA, 1999. USENIX.

[WSY95] Hong Wang, Tong Sun, and Qing Yang. Cat—caching address tags: A technique for reducing area cost of on-chip caches. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, page 381–390, S. Margherita Ligure, Italy, 1995. Association for Computing Machinery.

[XL11] Yuejian Xie and Gabriel H. Loh. Thread-aware dynamic shared cache compression in multi-core processors. In *Proceedings of the 2011 IEEE 29th International Conference on Computer Design*, ICCD '11, pages 135–141, Washington, DC, USA, 2011. IEEE Computer Society.

[XWL02] Yuan Xie, Wayne Wolf, and Haris Lekatsas. Code compression for vliw processors using variable-to-fixed coding. In *Proceedings of the 15th international symposium on System Synthesis*, pages 138–143, 2002.

[Yel11] Katherine Yelick. Exascale opportunities and challenges. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, page 1–2, San Jose, California, USA, 2011. Association for Computing Machinery.

[YG02] Jun Yang and Rajiv Gupta. Energy efficient frequent value data cache design. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 35, pages 197–207, Istanbul, Turkey, 2002. IEEE Computer Society Press.

[YGZ04] Jun Yang, Rajiv Gupta, and Chuanjun Zhang. Frequent value encoding for low power data buses. *ACM Trans. Des. Autom. Electron. Syst.*, 9(3):354–384, July 2004.

[YJ18] Chao Yan and Russ Joseph. Cocoa: Synergistic cache compression and error correction in capacity sensitive last level caches. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '18, page 117–128, Alexandria, Virginia, USA, 2018. Association for Computing Machinery.

[YKQ18] Vinson Young, Sanjay Kariyappa, and Moinuddin K Qureshi. Cram: Efficient hardware-based memory compression for bandwidth enhancement. *arXiv preprint arXiv:1807.07685*, 2018.

[YKQ19] Vinson Young, Sanjay Kariyappa, and Moinuddin Qureshi. Enabling transparent memory-compression for commodity memory systems. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 570–581. IEEE, 2019.

[YLK+04] Keun Soo Yim, Jang-Soo Lee, Jihong Kim, Shin-Dug Kim, and Kern Koh. A space-efficient on-chip compressed cache organization for high performance computing. In *Proceedings of the Second International Conference on Parallel and Distributed Processing and Applications*, ISPA'04, page 952–964, Hong Kong, China, 2004. Springer-Verlag.

[YNQ17] Vinson Young, Prashant J. Nair, and Moinuddin K. Qureshi. Dice: Compressing dram caches for bandwidth and capacity. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 627–638, Toronto, ON, Canada, 2017. Association for Computing Machinery.

[YZG00] Jun Yang, Youtao Zhang, and Rajiv Gupta. Frequent value compression in data caches. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, page 258–265, Monterey, California, USA, 2000. Association for Computing Machinery.

[ZIM+07] Li Zhao, Ravi Iyer, Srihari Makineni, Jaideep Moses, Ramesh Illikkal, and Donald Newell. Performance, area and bandwidth implications on large-scale cmp cache design. In *Proceedings of the Workshop on Chip Multiprocessor Memory Systems and Interconnect*, 2007.

[ZJCP18] Qi Zeng, Rakesh Jha, Shigang Chen, and Jih-Kwon Peir. Data locality exploitation in cache compression. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 347–354, Singapore, 2018. IEEE, IEEE.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

[ZL78] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory*, 24(5):530–536, 1978.

[ZLC+15] Jishen Zhao, Sheng Li, Jichuan Chang, John L. Byrne, Laura L. Ramirez, Kevin T. Lim, Yuan Xie, and Paolo Faraboschi. Buri: Scaling big-memory computing with hardware-based memory expansion. *ACM Trans. Archit. Code Optim.*, 12(3):31:1–31:24, 2015.

[ZPS+04] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, page 177–188, Boston, MA, USA, 2004. Association for Computing Machinery.

[ZYG00] Youtao Zhang, Jun Yang, and Rajiv Gupta. Frequent value locality and value-centric data cache design. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, page 150–159, Cambridge, Massachusetts, USA, 2000. Association for Computing Machinery.

List of Publications

[1] Daniel R. Carvalho and André Seznec. A case for partial co-allocation constraints in compressed caches. 2021.

[2] Daniel R. Carvalho and André Seznec. Understanding cache compression. *Transactions on Architecture and Code Optimization (TACO)*, 2021.

UNIVERSITÉ DE RENNES 1

Titre : Vers la Vompression à Tous les Niveaux de la Hiérarchie de la Mémoire

Mot clés : Cache, Mémoire, Compression de Hardware

Résumé : Les techniques de compression matérielle sont généralement des simplifications des méthodes de compression logicielle. Elles doivent, toutefois, se conformer aux contraintes de surface, de puissance et de latence. Cette étude dévoile les défis de l'adoption de la compression dans la conception de la mémoire. Le but de l'analyse n'est pas de résumer les propositions, mais de mettre en évidence les solutions qu'ils emploient pour relever ces défis. Une description détaillée des principales caractéristiques de plusieurs méthodes est fournie, ainsi que des critères qui peuvent être utilisés comme base pour l'évaluation de ces systèmes.

Généralement, ces schémas ne sont pas trés efficaces, et les schémas qui compressent bien décompressent lentement. Ce travail explore leur granularité pour redéfinir leurs perspectives et améliorer leur efficacité, à travers un concept appelé compression Region-Chunk. Son objectif est d'obtenir un haut (bon) taux de compression et une latence de décompression rapide. L'observation clé est qu'en subdivisant davantage les blocs de données compressés, on peut réduire la duplication des données. Ce concept peut être appliqué à plusieurs compresseurs précédemment proposés, entraînant une réduction de leur taille moyenne compressée. En particulier, un compresseur à décompression à cycle unique est boosté pour atteindre un niveau de compressibilité compétitif par rapport aux propositions de pointe.

Enfin, pour augmenter la probabilité de coallouer avec succès des lignes compressées, Pairwise Space Sharing (PSS) est proposé. PSS peut être appliqué orthogonalement aux méthodes de compactage sans pénalité de latence supplémentaire, et avec une surcharge de métadonnées rentable. Le système proposé (Region-Chunk + PSS) améliore encore la capacité normalisé moyenne du cache de 2,7% (moyenne géométrique), tout en offrant une courte latence de décompression.

Title: Towards Compression At All Levels In The Memory Hierarchy

Keywords: Cache, Memory, Hardware Compression

Abstract: Hardware compression techniques are typically simplifications of software compression methods. They must, however, comply with area, power and latency constraints. This study unveils the challenges of adopting compression in memory design. The goal of this analysis is not to summarize proposals, but to put in evidence the solutions they employ to handle those challenges. An in-depth description of the main characteristics of multiple methods is provided, as well as criteria that can be used as a basis for the assessment of such schemes.

Typically, these schemes are not very efficient, and those that do compress well decompress slowly. This work explores their granularity to redefine their perspectives and improve their efficiency, through a concept called Region-Chunk compression. Its goal is to achieve low (good) compression ratio and fast decompression latency. The key observation is that by further sub-dividing the chunks of data being compressed one can reduce data duplication. This concept can be applied to several previously proposed compressors, resulting in a reduction of their average compressed size. In particular, a single-cycle-decompression compressor is boosted to reach a compressibility level competitive to state-of-the-art proposals.

Finally, to increase the probability of successfully co-allocating compressed lines, Pairwise Space Sharing (PSS) is proposed. PSS can be applied orthogonally to compaction methods at no extra latency penalty, and with a cost-effective metadata overhead. The proposed system (Region-Chunk+PSS) further enhances the normalized average cache capacity by 2.7% (geometric mean), while featuring short decompression latency.