Multi-Gigabyte On-Chip DRAM Caches for Servers

THÈSE N^{O} 6631 (2015)

PRÉSENTÉE LE 15 SEPTEMBRE 2015

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE D'ARCHITECTURE DE SYSTÈMES PARALLÈLES
PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Djordje JEVDJIC

acceptée sur proposition du jury:

Prof. C. Koch, président du jury
Prof. B. Falsafi, directeur de thèse
Prof. G. Loh, rapporteur
Prof. . H.-H. S. Lee, rapporteur
Prof. E. Bugnion, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE
Suisse
2015

Either you repeat the same conventional doctrines everybody is saying, or else you say something true, and it will sound like it's from Neptune.
— prof. Noam Chomsky

In loving memory of my dearest cousin, Jovana Jevđić, may her soul rest in peace…

Acknowledgements

This thesis has been a long and dynamic journey, which I certainly would not be able to complete alone. I would like to spend a few words to express my enormous gratitude to some of the people who, directly or indirectly, helped me make this thesis happen.

First and foremost, I owe my biggest gratitude to my advisor and my friend, Babak Falsafi. I thank Babak for believing in me, for giving me the level of independence that I needed, for always having great and nonjudgmental understanding for every problem I encountered, for understanding and tolerating my disobedient personality, for supporting me in my numerous extracurricular activities, for teaching me so many things about computer science, and, most importantly, for shaping my views and being a great role model regarding those things in life that matter the most.

I was very fortunate to have the most highly regarded experts in my thesis committee. I thank Edouard Bugnion, Hsien-Hsin Sean Lee and Gabriel Loh for being part of my thesis, for giving me constructive feedback on my research, for shaping the direction of the thesis, and, most importantly, for giving me a hard time at the thesis proposal exam and thesis defense.

The past six years of my PhD life were tough, yet wonderful and full of memorable moments, thanks to my closest collaborators and friends, and my currently *youngest* academic siblings: Cansu Kaynak, Stavros Volos, and Onur Koçberber.[1] The four of us started the PhD program together, worked together, shared offices together, coauthored many papers together, spent sleepless nights together, ate together, partied together, travelled together, skied together, and ended our PhD trip together. Their presence, support and help were instrumental in making this thesis happen. Thank you guys!

I spent only one year less in the PARSA lab with Javier Picorel *Obando*. I owe him many thanks for many different things, but if I had to be specific, I would have to thank Javier for two important things: first, for taking the risk to be the first student I've had a chance to supervise. And second, for volunteering to drive that van full of PARSA members around the French Alps under heavy snow. Thank you, Javier!

[1] Sorted by the graduation date, not by sex

Acknowledgements

I don't have enough words to express my gratitude to the rest of the members, current or past, of the PARSA lab at EPFL, for their huge contributions to my professional and social well-being during my PhD. Listing all those contributions would take a separate book. But I cannot miss my opportunity to thank Sotiria Fytraki for filling the air of PARSA offices with her melodic Cretan accent. I will always remember Almutaz Adileh and Pejman Lotfi-Kamran for the fun we had on our frequent trips to exotic European countries for the EuroCloud project meetings.[2]. I'm truly grateful to Alexandros Daglis for organizing the trip to Las Vegas after ASPLOS'14 and for taking us to Stratosphere. I greatly thank Mohammad Alisafaee for referring me to some of the best black comedies. I thank Michael Ferdman and Alisa Yurovski for organizing the best tea parties, and Boris Grot and Gaya Khachatryan-Grot for being amazing hosts in Edinburgh. Thank you Effi Georgala, Stanko Novaković, Nooshin Mirzadeh, Georgios Psaropoulos, Mario Drumond, and Dmitri Ustiugov for all the moments, either fun or busy, that we shared together. Thank you Stéphanie Baillargues and Rodolphe Buret for always providing great administrative and technical help. Last, but not least, I thank monsieur Damien Hilloulin for translating the abstract of this thesis into French.

PhD is only the last step in my formal education. I owe enormous gratitude to my closest friends and collaborators, Andrijana Svorcan, Ivan Kuraj, Ana Romandić, Katarina Maksimović, Marija Popović, Una Aksentijević, Slavko Perišić, Đorđe Vuksanović, Viktor Vilotijević, among many others, for being great friends and supporting me throughout my long education in Serbia and Switzerland.

I owe special thanks to my lifelong friend, Darko Petrović. Darko and I were classmates since the kindergarten, then later in the elementary school, also during our undergraduate studies at the University of Belgrade, and finally during our PhD at EPFL. Occasionally, we were neighbors too. I thank Darko, his wife Ivana and their adorable daughter Iva, for being great friends and great neighbors during my entire stay in Switzerland.

For some of the most cheerful moments during my stay in Switzerland, I owe very special thanks to choir Fa7, in which I spent three wonderfull years singing jazz and gospel throughout the country. Thank you Jacqueline Savoyant, for sharing your positive energy, for encouraging me to sing solo, and trusting me on sometimes quite demanding but always beautiful and powerful solo tunes. Thank you Sylvie Füllemann and Olivier Nicole, singing with you every Tuesday was pure joy. Thank you Aurélie Burnier (I will never forget singing *Do you know what it means to miss New Orleans* with you), Vimi Gobin, Thaddeus Castillo, and Coralie Wenger Bonny for so many fun moments outside the choir. Thank you all Fa7 members for

[2] I will certainly not forget Pejman's and mine long layover in Amsterdam

accepting me genuinely and warmly into your choir, despite my poor French skills. Thank you for encouraging me to speak French and for helping me learn it. The time I spent with you gave me such an invaluable insight into the Swiss culture and the real Swiss spirit, and made my whole stay in Switzerland much more enjoyable.

Finally, I owe special thanks to Marko Milanović and Nemanja Kostić, for being my very best friends and helping me survive my PhD journey.

My family has always played an important role in my private and professional life. I'm eternally thankful to my grandparents, Vera and Dušan Žunić, Milosava and Mijailo Jevđić, for all the great moments I spent with them in their village, for all the healthy, fresh and delicious food I ate with them. I thank my grandad Mijajlo for teaching me very early that being different and thinking against the odds is something one should be proud of. I thank my uncles, Zoran Jevđić, Jovica Jevđić, Predrag Žunić, my aunts, Vesna Bjekić, Snežana Mitrić, Milica Radović, my first cousins, Jovana Jevđić, Milan Jevđić, Nikola Žunić, Stefan Žunić, Milutin Jevđić, Joviša Jevđić, Mirjana Radović and Miloš Radović being part of my dearest childhood memories and for their unconditional support. I thank my dear friend, Desimir Krupiniković, for being there for me and my family when it mattered the most.

Above all, I thank my parents, Dobrinka Jevđić and Vidoje Jevđić, and my brother Aleksandar Jevđić, for their endless love and support throughout my whole life.

Lausanne, July 22nd 2015

Abstract

While DRAM latency has long been recognized as a major bottleneck in servers, DRAM bandwidth is emerging as an important bottleneck as server processors shift to many-core architectures to allow for sustainable throughput improvements. The rapid expansion of the digital universe, increasingly stored in memory, rapidly pushes the need for higher DRAM density as well.

Emerging die-stacked DRAM technology dramatically improves the three major DRAM properties: latency, bandwidth and density. Recent advancements in die-stacking technology made it possible to integrate a sizeable amount of DRAM directly on top of the processor. While the feasible on-chip DRAM capacities are insufficient to satisfy the memory needs of modern servers, architecting on-chip DRAM as a high-capacity low-latency high-bandwidth cache has the potential to provide significant reduction both in off-chip memory traffic and in average memory access latency.

We make the observation that high-capacity on-chip DRAM caches expose abundant spatial locality present in server applications and a modest amount of temporal data reuse. As a consequence, DRAM caches that manage and fetch data at a coarser granularity, e.g., in 2KB pages, exhibit overall superior properties compared to caches that do fine-grain management using 64B blocks. These properties include substantially higher hit rates, smaller tag storage, higher energy efficiency and set-associativity. Unfortunately, naïve employment of page-based caches results in excessive data overfetch and capacity waste, as some of the fetched and allocated blocks are never accessed prior to their eviction. We demonstrate that if the cache is organized in pages, then page footprints — i.e., the set of blocks that are touched while the page is in the cache — are highly predictable using well-established code-correlation techniques. Accurately predicting access patterns within a page can eliminate most of the bandwidth overhead and capacity waste that page-based caches suffer from.

Key words: die-stacked DRAM, 3D integration, caches, memory bandwidth, memory latency, servers

Résumé

Tandis que la latence des mémoires DRAM est depuis longtemps reconnue comme un goulet d'étranglement pour les serveurs, la bande passante devient elle aussi limitante à mesure que les processeurs intègrent de plus en plus de cœurs d'exécution dans l'objectif d'augmenter les capacités de traitement. L'expansion rapide de l'univers digital, de plus en plus placé en mémoire, fait apparaître la nécessité de mémoires DRAM à plus grandes densités.

La technologie émergente de mémoires DRAM à dies empilés améliore les trois caractéristiques déterminantes de la DRAM : la latence, la bande passante et la densité. Les récents progrès dans l'empilement des dies rend possible d'intégrer des capacités importantes de DRAM directement sur le processeur. Tandis que la quantité de mémoire DRAM qu'il est possible d'intégrer sur une puce est à l'heure actuelle insuffisante pour satisfaire les besoins en mémoire des serveurs modernes, utiliser de la mémoire DRAM sur les puces comme un cache de grande capacité, à faible latence et grande bande passante a le potentiel de permettre une réduction significative du trafic mémoire externe et de réduire la latence mémoire en moyenne.

Nous faisons l'observation que les structures de caches locales de grandes capacités exposent la localité spatiale abondante des données présente dans les applications serveurs ainsi qu'un modeste degré de réutilisation temporel des données. Par conséquent, les caches DRAM qui gèrent et récupèrent les données à une granularité plus grossière, par exemple par pages de 2ko, présentent de meilleures caractéristiques que des structures de caches utilisant une gestion plus fine. Ces caractéristiques incluent de meilleurs taux d'utilisation, des étiquettes à stocker plus petites, une meilleure efficacité énergétique ainsi que la set-associativity. Cependant, l'utilisation naïve de tels caches reposant sur l'utilisation de pages amène à un préchargement excessif de données et à une perte de capacité étant donné que certains des blocs chargés et alloués ne sont pas utilisés avant leur éviction. Nous montrons que si le cache est organisé par pages, les motifs d'accès — i.e., l'ensemble des blocs accédés quand la page est dans le cache — sont hautement prévisibles en utilisant des techniques éprouvées de code-correlation. Prédire de manière précise les motifs d'accès au sein d'une même page permet d'éliminer une grande partie de l'augmentation du besoin en bande passante et aussi la perte en capacité dont les

caches reposant sur l'utilisation de pages souffrent.

Mots clefs : mémoires DRAM à dies empilés, l'integration 3D, cache, bande passante mémoire, latence mémoire, serveurs

Contents

Acknowledgements **i**

Abstract **v**

List of figures **xiii**

List of tables **xvii**

1 Introduction **1**

1.1 DRAM Caches and Server Applications 3

1.2 Footprint Cache 4

1.3 Scalable DRAM Caches 5

1.4 Improving DRAM Cache Efficiency 6

1.5 Thesis Statement and Contributions 8

2 DRAM Caches and Server Applications **11**

2.1 Background and Motivation 12

2.1.1 DRAM Cache Design Objectives 14

2.1.2 Block-Based Caches 15

2.1.3 Page-Based Caches 16

2.1.4 Summary 18

2.2 Spatial and Temporal Characterization 18

2.2.1 Methodology 18

2.2.2 Temporal Behavior 18

2.2.3 Spatial Behavior 19

2.2.4 Block-Based vs. Page-Based 21

3 Footprint Cache **25**

3.1 Footprint Cache 25

3.1.1 Footprint Prediction 26

3.1.2 Capacity Optimization . . . 27
3.2 Footprint Cache Design . . . 28
3.2.1 Footprint Cache Tag Array . . . 28
3.2.2 Prediction History . . . 29
3.2.3 Footprint Generation . . . 30
3.2.4 Capacity Optimization . . . 30
3.3 Methodology . . . 31
3.3.1 Baseline System . . . 31
3.3.2 DRAM Cache Organizations . . . 32
3.3.3 Workloads . . . 34
3.3.4 Simulation Infrastructure . . . 35
3.4 Results . . . 35
3.4.1 Spatial Characterization . . . 35
3.4.2 Coverage and Off-Chip Bandwidth . . . 36
3.4.3 Performance . . . 38
3.4.4 Sensitivity to Page Size and History Size . . . 39
3.4.5 Impact of Capacity Optimization . . . 40
3.4.6 Energy Implications . . . 40
3.4.7 Other Page-Based Proposals . . . 42
3.5 Discussion . . . 43
3.5.1 Footprint Cache and Coherence . . . 43
3.5.2 Knowledge and Transfer of PC . . . 43
3.5.3 SRAM Area Overhead . . . 45
3.5.4 Other Processor Architectures . . . 46
3.5.5 Cache Capacity . . . 46
3.5.6 Footprint Cache in Non-3D Systems . . . 47
3.6 Conclusion . . . 47

4 Scalable DRAM caches 49
4.1 Block-Based Caches with DRAM-Based Tags . . . 51
4.2 Unison Cache . . . 53
4.2.1 Alternative Approaches . . . 60
4.2.2 Summary and Comparisons . . . 62
4.3 Methodology . . . 63
4.3.1 Simulation Infrastructure . . . 63
4.3.2 Baseline System Configuration . . . 63

4.3.3 DRAM Cache Organizations . . . 64
4.3.4 Workloads . . . 65
4.4 Evaluation . . . 65
4.4.1 Predictor Accuracy . . . 65
4.4.2 Miss Ratio . . . 67
4.4.3 Performance . . . 69
4.4.4 Energy Considerations . . . 70
4.5 Tag Cache . . . 70
4.5.1 Tag Cache vs. Miss Predictor . . . 73
4.6 Efficient Footprint Tracking through Sampling . . . 74
4.7 Page Alignment . . . 76
4.7.1 Effects of Misalignments . . . 76
4.7.2 Mitigating Page Alignment Problems in Unison Cache . . . 77
4.8 Unison Cache in the Context of Near-Memory Acceleration . . . 80
4.9 Summary . . . 82

5 Research Directions for Improving DRAM Cache Efficiency 83
5.1 Increasing the Associativity in DRAM Caches . . . 84
5.1.1 Associativity in Block-Based DRAM Caches . . . 84
5.1.2 Associativity in Page-Based DRAM Caches . . . 87
5.2 Cache Insertion and Promotion Policies . . . 89
5.2.1 Promotion upon Evictions . . . 92
5.2.2 Summary . . . 93
5.3 Cache Bypassing . . . 93
5.3.1 Random Cache Bypassing . . . 95
5.3.2 PC-Based Cache Bypassing . . . 97
5.3.3 Cache Bypassing in Page-Based Designs . . . 103
5.4 Dead-Block and Dead-Page Prediction . . . 103
5.4.1 Quantitative Comparison . . . 105
5.5 Prefetching . . . 107

6 Related Work 109
6.1 Die-Stacked DRAM . . . 109
6.2 Spatial Prefetchers . . . 110
6.3 Cache Bypassing . . . 111
6.4 Way Prediction . . . 111

6.5 Dead-Block Prediction 112

7 Conclusions **113**

Bibliography **123**

Curriculum Vitae **125**

List of Figures

2.1 Opportunity for performance improvement with high-bandwidth and low-latency die-stacked DRAM. 13
2.2 A DRAM die stacked on top of the logic die architected as a block-based cache. One tag entry corresponds to one data block. 15
2.3 A DRAM die stacked on top of the logic die architected as a page-based cache. Only the useful blocks (accessed by the cores) are shown in the figure. One tag entry corresponds to one page. 17
2.4 Miss ratio of a block-based cache design for TPC-H database queries on a machine with 128GB of main memory. 20
2.5 Miss ratio of a block-based cache design for TPC-H database queries on a machine with 128GB of main memory as compared to a mix of SPEC2006 INT applications). 21
2.6 Page density as a function of cache size for Data Analytics, Data Serving and the Multiprogrammed workloads. 22
2.7 Page density as a function of cache size for Software Testing, Web Search, and Web Serving. 22
2.8 Miss ratio in block-based and page-based DRAM cache designs. 23
2.9 Off-chip traffic in block-based and page-based DRAM cache designs normalized to the baseline system without a DRAM cache. 23

3.1 Footprint Cache tag array and Footprint History Table (FHT). 28
3.2 Miss ratio of Block-based, Footprint, and Page-based cache organizations, normalized to a baseline system without a DRAM cache. 36
3.3 Off-chip bandwidth requirements of Block-based, Footprint, and Page-based cache organizations, normalized to a baseline system without a DRAM cache. . 37
3.4 Performance improvement of various designs over the baseline system for the Data Analytics and Multiprogrammed workloads. 38

3.5 Performance improvement of various designs over the baseline system for Data Serving. . . . 39
3.6 Predictor accuracy sensitivity to the page size, for a 256MB cache with 16K FHT entries. . . . 40
3.7 Hit ratio sensitivity to the history size. The DRAM cache capacity is 256MB and the page size is 2KB. . . . 41
3.8 Off-chip DRAM dynamic energy per instruction normalized to the baseline system. 42
3.9 Stacked DRAM dynamic energy per instruction normalized to the block-based design. . . . 43
3.10 Minimum size of an ideal cache needed to cover a given fraction of cache accesses. 44
3.11 Relative change in hit ratio and off-chip traffic for a 256MB Footprint Cache with 1KB pages using a tagless history table, normalized to the design with complete tags. . . . 46

4.1 Overview of the (a) Alloy Cache and (b) Footprint Cache designs. . . . 52
4.2 DRAM row content in Unison Cache (not drawn to scale). . . . 54
4.3 DRAM row organization in the Unison Cache design. . . . 56
4.4 DRAM row organizations for (a) block-based cache with footprint prediction, and (b) page-based cache with tagged blocks. . . . 59
4.5 Unison Cache's miss ratio as a function of associativity. . . . 67
4.6 Miss ratio comparison of Alloy Cache, Footprint Cache, and Unison Cache. . . . 68
4.7 Performance comparison of Alloy, Footprint, and Unison Caches. Note the difference in scale for Data Serving. . . . 68
4.8 Performance comparison for TPC-H queries. . . . 70
4.9 Tag cache hit ratio as a function of the number of tag cache entries. Each entry corresponds to one cache set (four 1KB pages) and requires 40B of storage. . . . 72
4.10 Cache accesses eliminated by a practical tag cache with a three-cycle access latency. The figure shows the fraction of unnecessary probes upon dirty evictions and cache misses that are eliminated, as well as the fraction of LRU/metadata updates upon cache hits that can be coalesced. The cache capacity is 256MB (2GB for TPC-H queries). . . . 73
4.11 Storage required for PC & offset pairs for an 8GB cache with 1KB pages. . . . 75
4.12 Increase in miss ratio due to misaligned pages. . . . 77
4.13 Alternative data layout with 1KB pages. . . . 78
4.14 Physical address bits. . . . 79

5.1 Miss ratio as a function of associativity for a 256MB block-based cache (2GB for TPC-H queries). 85

5.2 Way prediction accuracy as a function of associativity for a 256MB block-based cache (2GB for TPC-H queries). 86

5.3 The layout in DRAM of a single cache set in a two-way associative cache with 2KB pages and 2KB DRAM rows. Each cache set consists of two DRAM rows, and each way spans both rows. 88

5.4 Hit ratio in a four-way associative block-based DRAM cache with the traditional LRU policy and MRU Insertion (LRU) and with LRU Insertion (LIP) for Data Serving (left) and Media Streaming (right). Note the difference in the scale. Increase in hit ratio is directly proportional to the off-chip bandwidth savings (vertical dimension). The horizontal distance in the figure directly corresponds to the difference in effective capacity between the designs. 90

5.5 Hit ratio in a four-way associative block-based DRAM cache with the traditional LRU policy and MRU Insertion (LRU) and with LRU Insertion (LIP) for Web Search (left) and Web Serving (right). Note the difference in scale. Increase in hit ratio is directly proportional to the off-chip bandwidth savings (vertical dimension). The horizontal distance in the figure directly corresponds to the difference in effective capacity between the designs. The point in the figure on the right at which the curves cross each other is a starting point after which a block is more likely to be reused than not. 91

5.6 Fraction of evicted blocks that were reused before their eviction for various cache sizes. The cache is organized as direct-mapped block-based cache. The cache size for CloudSuite applications is between 256MB-1024MB, whereas for TPC-H queries the cache size is between 1GB and 4GB. The reuse of all blocks that flow in and out of the cache on average ranges between 23% for the smallest cache size, and 37% for the largest cache size. 94

5.7 DRAM row layout in Alloy Cache and BEAR. 95

5.8 Average number of accesses per block with (BEAR) and without (Baseline) random bypassing for a 256MB cache (2GB for TPC-H Queries), and the ratio between the two. 96

5.9 Hit ratio of a direct-mapped Alloy Cache (baseline), Alloy Cache with random cache bypassing (BEAR) and PC-based bypassing for Data Serving (left) and Media Streaming (right), normalized to the baseline design at 256MB. Note the difference in the scale. Increase in hit ratio is directly proportional to the off-chip bandwidth savings (vertical dimension). The horizontal distance in the figure directly corresponds to the difference in effective capacity between the designs 100
5.10 Hit ratio of a direct-mapped Alloy Cache (baseline), Alloy Cache with random block bypassing (BEAR), and PC-based bypassing for Web Search (left) and Web Serving (right), normalized to the baseline design at 256MB. Note the difference in the scale. The horizontal distance in the figure directly corresponds to the difference in effective capacity between the designs 101
5.11 Hit ratio of a direct-mapped Alloy Cache (baseline), Alloy Cache with random block bypassing (BEAR), and PC-based bypassing for Data Analytics, normalized to the baseline design at 256MB (left) and TPC-H queries, normalized to the 1GB baseline (right). Note the difference in the scale. The horizontal distance in the figure directly corresponds to the difference in effective capacity between the designs. 102
5.12 Off-chip traffic in Data Serving and Media Streaming in the baseline four-way Unison Cache, Unison Cache with counter-based dead-page prediction and Unison Cache with cache partitioning. The results are normalized to the off-chip traffic of the baseline design at 256MB. The horizontal distance in the figure directly corresponds to the difference in effective capacity between the designs. 105

List of Tables

2.1 Comparison of block-based and page-based designs. 17
2.2 Architectural system parameters. 19

3.1 Block state encoding 30
3.2 Architectural system parameters. 32
3.3 Footprint Cache parameters. 33
3.4 Block-based cache parameters. 33
3.5 Page-based cache parameters. 33

4.1 Comparison of Alloy Cache (AC), Footprint Cache (FC), and Unison Cache (UC). 51
4.2 Comparison of key characteristics of different DRAM cache schemes. 60
4.3 Architectural system parameters. 64
4.4 Footprint Cache parameters. 64
4.5 Accuracy of various predictors: Miss Predictor (MP) in Alloy Cache, and Footprint Predictor (FP) in Footprint Cache and Unison Cache, and Way Predictor (WP) in Unison Cache for a 1GB cache (8GB for TPC-H queries). 66

1 Introduction

Computer systems are among the worst affected by the ongoing energy crisis. For many decades the advancements in semiconductor technology have regularly provided exponential increase in computing power within a constant power budget. The technology advancements have recently started reaching the limits of physics, making it impossible to gain more in computing without investing more energy. Unfortunately, the overall energy budget allocated to computing is already prohibitively high, while the demand for computing is increasing exponentially as a consequence of data explosion and the constant emergence of new IT-based services. Most of those services are hosted in massive datacenters, whose space and energy footprints are constantly increasing. Minimizing the energy footprint and maximizing the compute and storage density are therefore the highest priority goals for datacenter operators, with substantial global impact. Efforts toward these goals must not, however, sacrifice the quality of these services, which is vital to their economic success.

Maximizing the compute density while minimizing the energy footprint mandates a dramatic increase in throughput per server chip with each technology generation at a reduced energy cost. With the slowdown both in Dennard Scaling and of Moore's law, server chips are resorting to larger numbers of increasingly less complex cores and specialized accelerators to support big data processing, while maintaining a practical power envelope and transistor budget. The resulting highly parallel processor organizations greatly benefit datacenter server workloads, which exhibit abundant request-level parallelism. This growth in core count, however, ultimately drives server processor designs into a memory bandwidth wall due to poor pin count scalability. Emerging many-core server chips with hundreds of cores are already able to utilize and even exceed their bandwidth budgets [21, 50], hitting the bandwidth wall before the power wall [21], and making the memory bandwidth a scarce resource.

The strict response latency requirements of IT-based services put enormous pressure on storage systems in datacenters. The dramatic increase in the amount of data mandates the use of high-density storage systems and devices. The high density and low access latency requirements are in a direct conflict. To meet both requirements, datacenter operators must keep massive amounts of frequently accessed data in main memory, which increasingly acts as a DRAM-based data cache. It is thus common for a today's server to accommodate hundreds of gigabytes of main memory per processor chip. Unfortunately, the need for high memory capacity puts further pressure on memory bandwidth. To be able to attach hundreds of gigabytes of DRAM to a single chip, multiple DRAM modules must be connected to every DRAM channel. Sharing a DRAM channel by multiple DRAM modules requires lowering the frequency of the channel, directly reducing its bandwidth and creating a fast-track to the bandwidth wall.

Die-stacked DRAM has been advocated as a promising technology to break the memory bandwidth wall and improve memory latency and density. It delivers several times more internal bandwidth compared to off-chip memory due to dense on-chip TSV buses, as well as lower access latency due to the reduction in physical distances enabled by die stacking. Recent advancements in die-stacking technologies have made it possible to tightly integrate a sizeable amount of DRAM in the same package as the processor. Having die-stacked DRAM on the chip or in the package could virtually eliminate the memory bandwidth wall by exposing all of its internal bandwidth at lower access latency. The latency advantage that die-stacked on-chip DRAM provides over conventional off-chip DRAM is particularly important in server applications, which are known to be memory-bound and suffer from low memory-level parallelism [1, 12].

Technological constraints, however, limit the on-chip stacked DRAM capacity to levels that are orders of magnitude lower than what modern server applications demand. It is impossible to fit all the main memory distributed across multiple multi-chip DRAM modules onto a single processor chip. Such a constraint forces the architects to use the on-chip stacked DRAM as a hardware-managed cache or as a software-managed cache or scratchpad. Managing die-stacked DRAM in software is a preferable option in custom designs where hardware and software evolve together, such as embedded systems. In contrast, deep, diverse and rapidly changing software stacks in server systems rely on general-purpose processors and operating systems, mandating non-intrusive hardware-based solutions.

This thesis investigates the use of on-chip die-stacked DRAM as a hardware-managed cache in processor chips for datacenters with the purpose of reducing memory traffic on the processor

side and improving memory latency. We provide a detailed characterization of real-world server software stacks with respect to DRAM caches in order to gain the critical insights that will lead us to appropriate cache designs. We demonstrate the potential of die-stacked DRAM caches to reduce memory traffic and improve memory latency in server systems, and propose effective, scalable and energy-efficient designs that realize that potential.

1.1 DRAM Caches and Server Applications

Our key observation is that server applications exhibit abundant spatial locality that becomes visible in high-capacity caches, such as on-chip DRAM caches. The reason behind the abundance of spatial locality is in the nature of server applications, which typically manipulate large objects or streams of data. The reason why this locality becomes apparent in DRAM caches is related to the long residency of objects at this level of the memory hierarchy: the longer an object stays in the cache, the more of it eventually becomes accessed by the processor.

While spatial locality in DRAM caches is abundant, temporal locality is scarce. Most of the temporal data reuse that happens within a short amount of time, e.g., within a single request, is filtered by L1 caches and is rarely seen in lower levels of the cache hierarchy. Data reuse seen in DRAM caches typically happens across independent requests and highly depends on object popularity distributions, also known as data skew; the more skewed the application data is, the more reuse takes place. As the cache size increases, the residency of objects in the cache increases and so does the probability of their reuse. Practical on-chip DRAM sizes, unfortunately, can capture only 1-2% of the hundreds of gigabytes stored in off-chip DRAM, which is typically not enough to capture the working set comprising the most frequently accessed data even under high skew.

The abundance of spatial locality and the paucity of temporal locality suggest that DRAM caches may benefit from large cache lines. DRAM caches that organize and fetch data in spatial regions of several kilobytes (e.g., 4KB pages) take advantage of the spatial locality and exhibit an order of magnitude lower miss rate compared to caches that manage data in conventional 64B blocks [18, 29, 30, 41]. We refer to such designs as *page-based caches*. Managing data at such coarse granularity also results in a commensurate reduction in tag space, which enables the placement of tags in SRAM for moderately sized caches. Unfortunately, the excessive data overfetch caused by accesses to sparse pages may substantially increase the off-chip traffic and offset any bandwidth benefits provided by caching.

Cache designs that employ conventional 64B blocks [46, 47, 48, 55], and which we refer to as

block-based caches, utilize the available off-chip bandwidth much more efficiently, but see an order of magnitude more misses compared to their page-based counterparts. Although some of the gap in miss rates between the designs could be bridged through prefetching, existing implementation of block-based designs do not provide efficient support for it. Because block-based caches do the bookkeeping at the level of individual blocks, they require prohibitively large tag space that cannot be stored in SRAM even for the smallest DRAM cache sizes and therefore store the tag array in the stacked DRAM. Storing tags in DRAM either significantly increases the cache access latency [46, 47, 48] or completely disables cache associativity in block-based designs [55], enforcing a direct-mapped organization. While the direct-mapped organization *per se* does not harm block-based DRAM caches due to the large number of sets, it does not support many standard cache optimization techniques that rely on associativity. In contrast to page-based caches, block-based designs efficiently utilize the available cache capacity, because they do not suffer from fragmentation. However, limited temporal locality implies that on-chip DRAM caches are less sensitive to small capacity variations; techniques that suboptimally use cache capacity will therefore not necessarily perform suboptimally.

1.2 Footprint Cache

Block-based and page-based designs show complementary properties. On one hand, block-based designs are much more efficient in using cache capacity and off-chip bandwidth, but suffer from low hit rates. Their tag array is huge and must be stored in DRAM at the cost of either high latency or associativity. On the other hand, page-based caches provide high hit rates and small and arbitrarily associative SRAM-based tag storage. However, they severely misuse the precious off-chip bandwidth resources, and as such are not a feasible option.

Our goal is to preserve the properties of page-based designs, but without the unnecessary traffic and with better capacity management. Toward that goal, we propose Footprint Cache, which is a sectored cache organization that separates the cache allocation unit from the fetch unit. Upon a cache miss, Footprint Cache allocates a page, but fetches only those 64-byte blocks within the page that are predicted to be useful in future. In doing so, Footprint Cache eliminates the unnecessary off-chip and on-chip traffic stemming from the movement of unused data.

To mitigate the poor capacity management in page-based designs, Footprint Cache identifies pages that have the fewest useful blocks and show neither spatial nor temporal reuse, and does not allocate entries in the cache for such pages. It instead fetches 64-byte blocks from

such pages, one by one and only on demand, and forwards them to the requestor, bypassing the cache. Such pages account for a significant fraction of all pages that are fetched and make the biggest contribution to the capacity waste.

Footprint Cache mitigates most of the bandwidth and capacity problems of page-based designs and manages to get the best of the page-based and block-based designs. The key to Footprint Cache's success is its *fooprint predictor*, a simple hardware structure that estimates the spatial *footprint* of each page — i.e., the exact set of blocks that will be demanded during the page's on-chip residency. To design an effective footprint predictor, we rely on the observation that the majority of huge server datasets are accessed by a limited number of code fragments that have repetitive and predictable behavior and result in recurring access patterns. Our footprint predictor fully relies on the correlation between the code and spatial locality [62] to predict access patterns within each page, which are then used to reduce both the bandwidth and the capacity waste in page-based designs. Furthermore, by fetching and evicting all useful data in a page at once, Footprint Cache significantly reduces the number of row activations in off-chip DRAM and saves a substantial amount of its dynamic power.

1.3 Scalable DRAM Caches

What allows Footprint Cache to store its tags in SRAM is its page-based organization, which minimizes the storage required for the tags. It is best suited for caches in the range of several hundred megabytes. However, as the technology rapidly enables multi-gigabyte stacked DRAM capacities, even page-based tags will quickly consume too much SRAM to be practical. To illustrate, 8GB of stacked DRAM organized in 4KB pages would need 16MB of SRAM in the best case, which is in the order of today's last-level cache sizes. This storage drastically increases if the cache uses sub-blocking to optimize for off-chip bandwidth, as Footprint Cache does. Furthermore, while the stacked DRAM provides a huge increase in bandwidth compared to conventional DDR channels, the *latency* of the die-stacked DRAM is not substantially better. If a DRAM cache architecture requires accessing the stacked-DRAM or a multi-megabyte SRAM table for tag lookups, then that could add several tens of cycles to the overall cache latency, offsetting any latency advantage of the stacked DRAM technology.

To overcome Footprint Cache's scalability limitation, we introduce a novel set-associative page-based DRAM cache design, called Unison Cache, which carefully incorporates the tag metadata directly into the stacked DRAM to enable scalability to arbitrary stacked-DRAM capacities. Unison Cache stores each cache set in a DRAM row, placing page tags at the beginning of

each DRAM row, followed by the corresponding data blocks. To support associativity without serializing tag and data accesses, Unison Cache employs a simple address-based way predictor, which is, thanks to the spatial locality and large page size, highly accurate. Upon a cache request, provided that the way prediction is correct, the exact location of the requested block can be correctly determined. Although every data block is physically separated from its tag within the DRAM row, the positions of both the tag and the data block are known in advance, so the tag and data accesses can be fully overlapped, removing the tag lookup latency from the critical path.

1.4 Improving DRAM Cache Efficiency

The main motivation behind the research on DRAM caches is the reduction in traffic between the processor and the memory. DRAM caches provide a traffic reduction on the processor side solely through reuse of locally stored copies of data within high capacity on-chip DRAM. What makes the reuse possible is data skew; certain types of data, such as metadata, are more frequently accessed than others; certain objects also happen to be more popular than others. Despite their high capacity, practical DRAM cache sizes are still two orders of magnitude smaller than the off-chip main memory in the subsequent level of the hierarchy and as such cannot accommodate the hot data structures for the majority of applications [28]. As a result, the amount of temporal reuse in on-chip DRAM caches is fairly low [8, 29].

The underlying mechanism through which capacity-constrained caches exploit reuse is associativity, which stands for the number of slots into which a new cache entry can be inserted. Associativity enables control over the placement of data in the cache and over their promotion through recency lists that aim to rank the possible victim options according to the likelihood of their reuse. Unfortunately, practical DRAM cache implementations provide either no associativity at all [8, 55] or very limited associativity [18, 28, 55], which severely limits the cache's ability to identify and keep reusable data in the cache.

The abundance of spatial locality and the lack of either temporal reuse or mechanisms to exploit it may lead to cache thrashing. In page-based designs, pages with high spatial locality typically show less temporal reuse and occupy space in the cache for a long time, but are often not useful after they are completely scanned. In block-based designs, most of the data that is inserted into the cache is *dead upon arrival* [8, 29]. It is therefore important to provide DRAM caches with mechanisms that would on one hand minimize the cache space and cache bandwidth resources allocated to data that is not reused, and on the other hand exploit the

existing and encourage more reuse among data that are prone to it, and therefore improve the overall cache efficiency.

In this thesis we revisit some of the traditional techniques for improving cache efficiency in the context of DRAM caches. We demonstrate that they are either ineffective or not directly applicable to DRAM caches. We recognize the problems associated with applying those techniques to DRAM caches and propose new research directions that have the potential to improve cache efficiency. The techniques we consider include:

- Increasing associativity, which is a trivial technique to reduce the number of conflict misses and support reuse. Increasing associativity in SRAM caches is expensive from the power perspective, but in DRAM caches has different benefits and cost implications. Increasing associativity in block-based DRAM caches provides minimal benefit in terms of hit rates, but enables a variety of cache optimization techniques that rely on associativity. Unfortunately, increasing associativity in block-based DRAM caches is not practical as it implies a commensurate increase in die-stacked DRAM bandwidth. On the contrary, we find that associativity is vital to performance of page-based DRAM caches and for which we propose techniques to support arbitrarily high associativity.

- Cache bypassing, which aims to identify non-reusable cache blocks and avoid storing them in the cache to prevent cache pollution. Cache bypassing avoids pollution in block-based DRAM caches, but existing block-based DRAM solution lack support for it, as they cannot identify non-reusable data in a practical way. Instead, bypassing of randomly selected cache blocks could be used not to improve the hit ratio, but to trade it for a reduction in cache activity [8]. We demonstrate that PC-based prediction techniques have the potential to more accurately identify non-reusable blocks, and we propose an efficient method for their integration into block-based DRAM caches. Our method relies on sampling and incurs no cost related to storage, bandwidth or latency. Unlike block-based caches, page-based DRAM caches can more easily identify non-reusable data. However, we show that applying cache bypassing to page-based DRAM caches may do more harm than good, as it reduces the number of cache hits and severely undermines the opportunity for energy savings in off-chip DRAM.

- Improved policies for cache insertion, promotion and replacement. We demonstrate that the fundamental cache replacement optimizations [56] are applicable neither to block-based nor to page-based DRAM caches. Namely, such techniques fundamentally rely on associativity, whereas state-of-the-art block-based DRAM caches are direct-

mapped [8, 55]. Page-based caches confuse the optimization techniques by promoting a whole page, i.e., by giving it a recently accessed status upon accesses to different blocks within the same page. While such situations happen due to page reuse — i.e., due to spatial locality — the effective temporal data reuse does not happen. The optimization techniques will therefore confuse spatial locality for temporal and promote the page in question, penalizing pages that do exhibit temporal reuse.

- Dead-block prediction. There has been a large body of research trying to mitigate the cache pollution problem in SRAM caches through dead-block prediction [35, 23, 36, 40, 44]. We show that while dead-block prediction can be accurately performed in block-based DRAM caches, the lack of cache associativity severely limits its usability. In contrast, we show that PC-based *dead-page prediction* could almost double the effective page-based cache capacity for certain applications, while leveraging existing Unison Cache's metadata structures to perform predictions.

- Prefetching. Page-based designs implicitly rely on spatial prefetching to boost their hit ratio. Although prefetching could significantly benefit block-based designs as well, we show that the lack of centralized information about the presence of neighboring blocks in the cache severely limits the applicability of spatial prefetchers in block-based DRAM caches. Further research on effective prefetchers for block-based DRAM caches is highly encouraged.

1.5 Thesis Statement and Contributions

The thesis statement reads as follows:

Effective and efficient on-chip DRAM cache designs for servers must leverage the abundant spatial locality in server applications and must do so in a bandwidth- and capacity-efficient manner.

Using analytic models, trace-driven and cycle-accurate full-system simulation of modern, real-world server workloads, this thesis demonstrates that:

- High capacity on-chip DRAM caches expose abundant spatial locality of server applications and their modest temporal locality. As a consequence, DRAM caches that manage and fetch data at a coarser granularity exhibit overall superior properties compared to caches that do fine-grain management. These properties include higher hit rates,

smaller tag storage, and higher energy efficiency. However, their naïve employment results in excessive data overfetch and capacity waste that can offsets any benefits of DRAM caches.

- If the cache is organized as page-based, page footprints — i.e., the set of blocks that are touched while the page is in the cache — are highly predictable using well-established code-correlation techniques [62]. Predicting page footprints can eliminate most of the bandwidth overhead and capacity waste that page-based caches suffer from.

- Fetching whole page footprints at once and writing them back together to the main memory greatly improves the energy efficiency in off-chip DRAM by reducing the number of DRAM row activations by an order of magnitude as compared to fetching the same set of blocks separately.

- Unlike block-based caches, page-based caches need a modest amount of associativity to avoid frequent conflicts. Associativity can be efficiently implemented, even in caches with DRAM-based tags through way prediction, which is highly accurate for and only for page-based designs. We demonstrate an efficient implementation of arbitrarily high associativity for page-based designs.

- It is possible to build a scalable, associative, low-latency page-based cache design with DRAM-based tags that achieves high hit rates and high bandwidth efficiency.

- Although associativity is not crucial for the baseline cache performance in block-based DRAM caches, its absence disables many standard cache optimization techniques that block-based caches could otherwise greatly benefit from.

- There is a significant correlation between the code and temporal data reuse. In the absence of associativity, block-based DRAM caches could leverage this correlation and perform PC-based cache bypassing not only to reduce the cache activity but also to increase the hit rate. We demonstrate an efficient implementation of cache bypassing with only 16KB of SRAM storage and without bandwidth, latency or storage costs in the die-stacked DRAM. Page-based caches can leverage the correlation between the code and data reuse to employ dead-page prediction and increase cache efficiency.

This thesis covers the DRAM cache design space in single-socket setups. While our findings are equally applicable to multi-socket setups, providing efficient support for cache coherence between multiple sockets equipped with multi-gigabyte DRAM caches is out of the scope of this thesis and is an important research topic for future work.

While this thesis is focused on die-stacked DRAM, the major conclusions of the thesis are not bound to any specific technology and may apply to other materials, such as PCM, STT-RAM or other MRAM technologies, with different performance, energy and durability implications.

The rest of the thesis is organized as follows. In Chapter 2 we study the behavior of real-world server applications in the context of DRAM caches and provide critial insights that will lead us to effective designs. In Chapter 4.2 we look at highly associative DRAM cache designs that keep the precise presence information about the cache content in SRAM, and propose Footprint Cache, a design that leverages spatial locality in a bandwidth-efficient way. Chapter 4 proposes a Unison Cache, an effective DRAM cache solution that scales to multi-gigabyte capacities thanks to its DRAM-based tags. In Chapter 5 we study various techniques that aim to improve cache efficiency and propose further research directions toward that goal. Chapter 6 presents the relevant related work, and Chapter 7 concludes the thesis.

2 DRAM Caches and Server Applications

Die-stacked DRAM has been advocated as a promising technology to break the memory bandwidth wall and improve memory latency and density. It delivers several times more internal bandwidth compared to off-chip memory due to dense on-chip TSV buses, as well as lower access latency due to reduction in physical distances enabled by die stacking. Recent advances in die-stacking technologies have made it possible to tightly integrate a sizeable amount of DRAM in the same chip as the processor. Having die-stacked DRAM on the chip could virtually eliminate the memory bandwidth wall by exposing all of its internal bandwidth at lower access latency. The latency advantage that die-stacked on-chip DRAM provides over conventional off-chip DRAM is particularly important in server applications, which are known for being memory-bound [1, 12].

Technological constraints, however, limit the on-chip stacked DRAM capacity to levels that are orders of magnitude lower than what modern server applications demand. It is impossible to fit all the main memory distributed across multiple multi-chip DRAM modules onto a single processor chip. Such a constraint forces the architects to use the on-chip stacked DRAM as a hardware-managed cache or as a software-managed cache or scratchpad. Managing die-stacked DRAM in software is a preferable option in custom designs where hardware and software evolve together, such as embedded systems. In contrast, deep, diverse and rapidly changing software stacks in server systems rely on general-purpose processors and operating systems, mandating non-intrusive hardware-based solutions.

In this chapter we investigate the use of on-chip die-stacked DRAM as a hardware-managed cache in processor chips for datacenters with the purpose of reducing memory traffic on the processor side and improving memory latency. We demonstrate the potential of die-stacked DRAM caches to reduce memory traffic and improve memory latency in server systems, and

we look at different trade-offs in DRAM cache designs that are specific to server settings.

2.1 Background and Motivation

With the slowdown in Dennard Scaling server chips are resorting to larger numbers of lean cores to maintain a practical power envelope. Scale-out server workloads benefit from such many-core processor organizations, which enable high throughput thanks to the abundant parallelism in these workloads. The growth in core count, however, ultimately drives designs into a memory bandwidth wall due to poor pin count scalability. Emerging many-core chips with hundreds of cores are already able to utilize and even exceed their bandwidth budgets [21, 34, 50], hitting the bandwidth wall before the power wall [50].

Recent research advocates using die-stacked DRAM to break the memory bandwidth wall and improve memory latency [21, 25, 30, 34, 43, 45, 46, 48]. Figure 2.1 assesses the opportunity of the technology to boost performance of scale-out server and multiprogrammed Desktop workloads from two aspects: bandwidth and latency (details on the experimental methodology are explained in Section 2.2.1). The first set of bars shows performance improvement for a many-core server system [50] with the main memory fully integrated on the chip using die stacking, providing 8x the bandwidth of the 2D baseline. The second set of bars shows performance improvement of the same high-bandwidth system, but with halved DRAM latency [48]. We see that both bandwidth and latency play a vital role in achieving high performance, which implies that future designs must exploit both opportunities given by the technology.

Technological constraints, however, limit the stacked DRAM capacity to levels that are far lower than what modern server workloads demand [48]. While today's servers need tens to hundreds of gigabytes of DRAM each, the projections for die-stacked DRAM capacity vary between hundreds of megabytes to several gigabytes. Thus, most proposals for die stacking advocate using the stacked DRAM as a cache [30, 46, 48]. Unfortunately, the inherent limitations of DRAM cache designs prevent them from achieving the full potential of the technology, depicted in Figure 2.1. Firstly, DRAM caches, regardless of their organization, require significant tag storage due to their large capacity, whose lookup necessarily adds extra latency to the critical path. Secondly, the limited capacity of the stacked DRAM limits the level of concurrency it can provide, despite the virtually unlimited TSV bandwidth. The stacked DRAM is orders of magnitude smaller than the off-chip DRAM, and, thus, experiences frequent bank conflicts and lower availability. In contrast to off-chip main memory systems

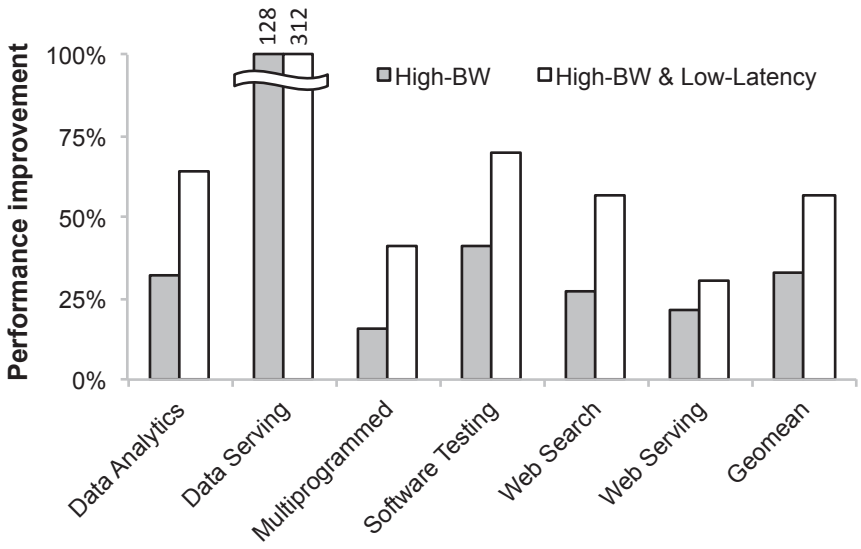

Figure 2.1 – Opportunity for performance improvement with high-bandwidth and low-latency die-stacked DRAM.

with hundreds of gigabytes of DRAM that provide more bandwidth than the memory channels can sustain, the bandwidth to the DRAM cache is restricted by the parallelism in the stacked DRAM itself, and not by the interface. Therefore, stacked DRAM caches fall short of fully leveraging the abundant on-chip bandwidth enabled by dense TSV buses. Cache designs must be aware of this limitation, and optimize for the stacked DRAM locality to allow for higher concurrency and availability.

Despite their capacity, DRAM caches may exhibit high miss ratios, with each miss being satisfied from the off-chip memory at full off-chip latency. This behavior is caused by the low reuse of the data in lower-level caches [20], in contrast to L1 caches, where data are frequently reused and where most of the temporal locality is exploited. This phenomenon is further exacerbated by vast datasets of scale-out workloads [12], which do not form any well-defined working sets within the cache sizes of interest. Besides their latency penalty, misses inherently result in DRAM cache evictions. We find that, for scale-out workloads, these are mostly dirty evictions, because data reside in the cache for long enough to become modified by dirty evictions from the upper-level caches. Dirty evictions consume additional off-chip and on-chip TSV bandwidth, affecting the stacked DRAM availability as well (the data have to be read from the stacked DRAM and written back to the off-chip DRAM). The same holds for cache fills that follow the misses. The bandwidth overhead caused by secondary cache traffic — e.g., evictions, fills, probes — is sometimes referred to as *bandwidth bloat* [8].

2.1.1 DRAM Cache Design Objectives

DRAM cache designs fall short of leveraging the benefits that die stacking technology provides. In this section we present a set of objectives and guidelines for designing effective DRAM caches aiming to bridge the gap between die-stacked main memory and die-stacked caches:

- Fast tag lookup. Because tag lookups are on the critical path of all requests coming to the cache, tag lookup latency must be minimized. While this statement holds for all cache designs that serialize tag and data lookups, it gains more importance in the context of DRAM caches, due to their tag array size.
- Small tag storage. The total storage dedicated to tags or other metadata should be minimal, as it does not directly contribute to better system performance, but does incur high storage cost.
- Low off-chip traffic. While cache misses are responsible for most of the off-chip bandwidth overhead, various cache features can adversely impact off-chip bandwidth even further. Examples include the use of large cache blocks that saturate off-chip bandwidth and the use of predictors for miss speculation. Reduction in off-chip traffic is the main driver for 3D-stacked DRAM adoption, and as such should be among the top priority goals.
- The stacked DRAM bandwidth overhead caused by secondary cache traffic — i.e., evictions, fills, probes, replacement policy metadata updates —should be minimized.
- High hit ratio is crucial to leveraging both the bandwidth and the latency advantages of the die-stacking technology, demonstrated by Figure reffig:motivation.
- Low hit and miss latency. To achieve the benefits depicted in Figure 2.1, DRAM caches must optimize for both hit and miss latency. Internal details of the cache organization should neither penalize hit latency nor postpone miss serving.
- High DRAM access locality. Accesses to DRAM structures experience unpredictable latency, highly dependent on the locality of references, availability, address-mapping schemes, row-buffer management policy, and access scheduling. To minimize the stacked DRAM and off-chip DRAM access latencies and energy per access, cache designs must take of all these parameters into account.
- Efficient capacity management. Allocation of space for data that are never used should be avoided. The problem is severe in page-based designs, which suffer from internal

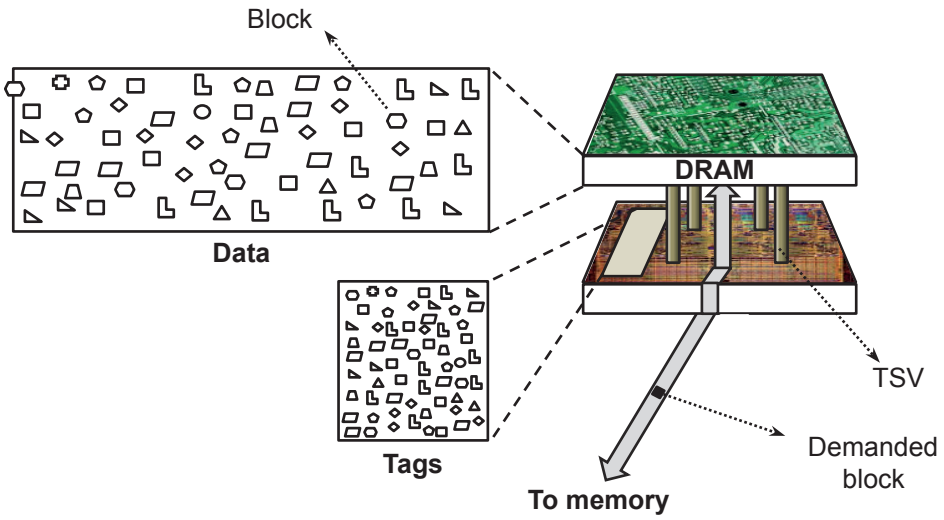

Figure 2.2 – A DRAM die stacked on top of the logic die architected as a block-based cache. One tag entry corresponds to one data block.

fragmentation.

Unfortunately, many of the aforementioned requirements are mutually conflicting, which makes the design process more challenging. To better understand such challenges, we focus on two main DRAM cache design classes that achieve different goals.

2.1.2 Block-Based Caches

On-chip caches have traditionally been designed to primarily exploit temporal locality, and to make the best use of their limited capacity. Trade-offs between the effective cache capacity, temporal and spatial locality resulted in 16- to 128-byte cache blocks, 64- byte being the most common block size employed today. Such a design is illustrated in Figure 2.2. For large DRAM caches, 64-byte blocks would require huge tag storage, as illustrated in Figure 2.2, which is infeasible to build in SRAM, thereby forcing the tags to be embedded in DRAM [30, 46, 48]. Embedding tags in DRAM, however, results either in multiple DRAM accesses per cache request — and, consequently, in substantially higher hit and miss latencies [46] — or in the absence of associativity.

Intelligent co-location of data with the corresponding tags in the same DRAM row [46] accompanied with optimized access scheduling, as done in Loh & Hill cache, [48], obviates the need for multiple DRAM accesses per request. However, this optimization only partially reduces

the high hit latency, because of the need for several operations to be performed within the DRAM row-buffer. Furthermore, the co-location of tags and data may in such a way mandates particular data placement policies that diminish DRAM locality. It also requires a way to determine the presence of a block in the cache prior to accessing the tags, as well as additional multi-megabyte storage for that purpose (not shown in Figure 2.2), whose access latency is on the critical path.

A more recent block-based cache design, called Alloy Cache [55], provides an architecture that completely avoids any large SRAM-based tag arrays, and overall provides low latencies on cache hits. Alloy Cache is organized as direct-mapped to avoid searching for the correct way throughout the DRAM-based tags and co-locates each data block with its tags, reading it together with the data block in a single access. However, these advantages come at the cost of relatively low cache hit rates, which are further penalized by the cache's direct-mapped organization, and high miss penalty. To avoid DRAM cache lookups on cache misses, Alloy Cache employs a miss predictor, sending cache requests to main memory speculatively, if a miss is predicted. However, because the miss predictor is imperfect, it can be relied upon for coherence or aggressive prefetching.

Regardless of their tag architecture, block-based designs fall short of exploiting abundant spatial locality. Instead, they focus on limited temporal locality, experiencing high miss ratios, thus frequently exposing full off-chip latency to incoming requests. However, due to the small fetch unit and the efficient management of cache capacity, block-based designs minimize off-chip traffic, making them a favorable option for high-throughput servers.

2.1.3 Page-Based Caches

Increasing the block size allows for a proportionate reduction in tag storage. The use of larger allocation/fetch units (e.g., 1-8KB) makes the placement of tags in SRAM feasible at an acceptable storage overhead [18, 29, 30]. We call such units *pages* and the corresponding designs *page-based* designs.

The large fetch unit allows for maximum DRAM access efficiency, fully exploiting locality in both off-chip and stacked DRAM. For instance, a single DRAM row opening is needed per off-chip DRAM fetch, eviction, or stacked DRAM fill, for a whole page, assuming that the page size does not exceed the DRAM row size. While large DRAM caches exhibit limited temporal locality, they show significant spatial locality, which can be easily leveraged by large fetch units, as illustrated in Figure 2.3, providing an order of magnitude more hits compared to a block-based

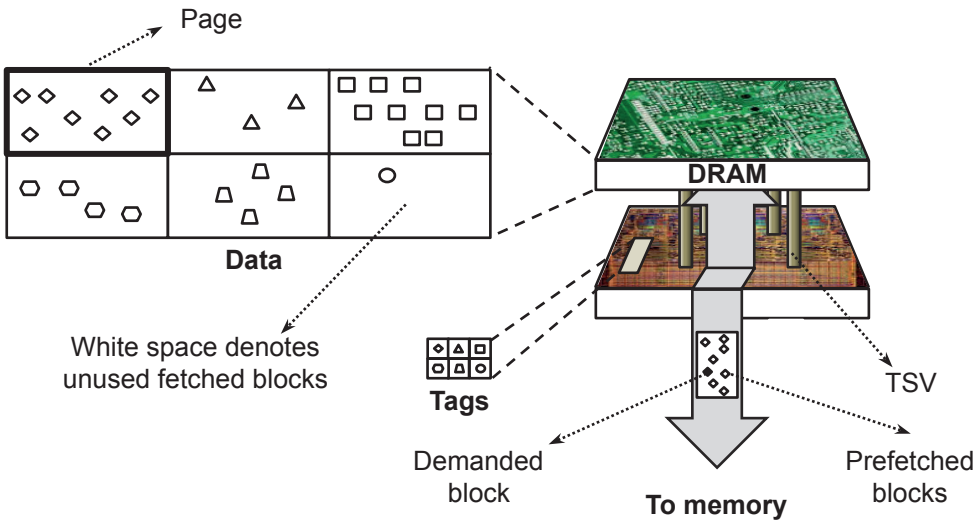

Figure 2.3 – A DRAM die stacked on top of the logic die architected as a page-based cache. Only the useful blocks (accessed by the cores) are shown in the figure. One tag entry corresponds to one page.

	Block-based – Loh & Hill	Block-based – Alloy	Page-based
Small and fast tag storage	✗	✗	✓
Low off-chip traffic	✓	✓	✗
High hit rate	✗	✗	✓
Low hit latency	✗	✓	✓
Associativity	✓	✗	✓
High DRAM locality	✗	✓	✓
Efficient capacity management	✓	✓	✗

Table 2.1 – Comparison of block-based and page-based designs.

cache of the same size [25, 30]. Cache hits are critical to exploiting the latency advantages of die-stacked DRAM and page-based caches provide them at lower latency. Unfortunately, many of the cached pages contain data that are not used prior to the page eviction, resulting in excessive data overfetch [30] and capacity waste. As a result, page-based caches tend to increase the off-chip traffic of the baseline system without a DRAM cache by up to an order of magnitude in the worst case, which negates a key benefit of die-stacked DRAM caches.

2.1.4 Summary

Table 4.1 provides a comparison between the two designs with respect to the most important features. The block-based and page-based designs show complementary, yet mutually exclusive, characteristics.

2.2 Spatial and Temporal Characterization

In this section we characterize server applications by studying their behavior in large-scale DRAM caches.

2.2.1 Methodology

To study the behavior of DRAM caches in server settings we use CloudSuite workloads [9], including Data Analytics, Data Serving, Software Testing, Web Search, and Web Serving. For comparison, we also include a mix of SPEC2006 Integer benchmarks as a representative of desktop applications. Because the datasets of CloudSuite benchmarks are slightly scaled down to allow for practical full-system simulation, we also include one unscaled server application, which is a modern column-oriented database, MonetDB, running a set of TPC-H queries on a server with 128GB of main memory. We evaluate the applications while running on one pod of a scale-out server chip [50]. The simulation parameters are given in Table 2.2.

2.2.2 Temporal Behavior

To study the temporal behavior of DRAM caches, we look at the miss ratio of block-based designs that use no prefetching. Because block-based designs do not leverage spatial locality, any observed cache hits come solely from temporal data reuse. Figure 2.4 shows the miss ratio of a block-based design for TPC-H queries running on a 16-core machine with 128GB of memory. The cache capacity varied on the x-axis is shown as a ratio between the cache capacity and the main memory size. We see that the working set size (WSS), which is between 10% and 15% of the dataset, is well beyond the reach of practical DRAM caches, which could accommodate up to a few gigabytes in the best case. As a result, the temporal reuse seen in DRAM caches is limited. Note that in server applications the temporal reuse observed in L1 caches typically comes from reuse within a single server request. This kind of reuse is typically fully filtered by SRAM caches and is not visible at the DRAM cache level. In contrast, the reuse in DRAM caches is likely to happen accross different server requests and is a result of

Technology	20nm, 0.85V, 3GHz
CMP Organization	16-core Scale-Out Processor pod
Core	ARM Cortex-A15-like, 3-way OoO @3GHz
L1-I/D caches	64KB, split, 64B blocks 2-cycle load-to-use latency
L2 cache per pod	4MB, unified, 16-way, 64B blocks, 4 banks, 13-cycle hit latency
Interconnect	16x4 crossbar
Off-chip DRAM	16-32GB, one DDR3-1600 (800MHz) channel 8 banks per rank, 8KB row buffer
Stacked DRAM	DDR-like interface (1.6GHz) 4 channels, 8 banks/rank, 8KB row buffer, 128-bit bus width
t_{CAS}-t_{RCD}-t_{RP}-t_{RAS} t_{RC}-t_{WR}-t_{WTR}-t_{RTP} t_{RRD}-t_{FAW}	11-11-11-28 39-12-6-6 5-24

Table 2.2 – Architectural system parameters.

difference in popularity among objects.

To zoom into the area of interest, we plot the same graph in Figure 2.5, but this time on a logarithmic scale. For a referrence, we show the miss ratio for a mix of 16 SPEC benchmarks. We observe dramatically different behavior between real-world server applications and desktop applications, whose working sets could fit in a small DRAM cache. The striking difference in behavior between server and desktop applications consequently has a significant impact on the DRAM cache design.

2.2.3 Spatial Behavior

Our key observation is that server applications exhibit abundant spatial locality that becomes visible in high-capacity caches, such as on-chip DRAM caches. The reason behind the abundance of spatial locality is in the nature of server applications, which typically manipulate large objects or streams of data. The reason why this locality becomes apparent in DRAM caches is related to the residency of objects at this level of the memory hierarchy: the longer an object stays in the cache, the more of it eventually becomes accessed by the processor. Page-based designs, whose allocation unit is in the order of a few kilobytes, expose on abundant spatial

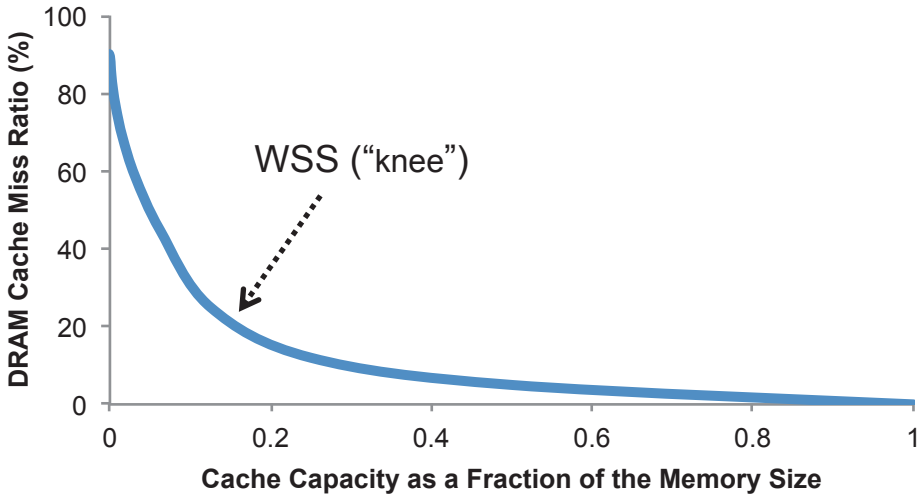

Figure 2.4 – Miss ratio of a block-based cache design for TPC-H database queries on a machine with 128GB of main memory.

locality that can benefit many workloads. As a result, page-based designs enjoy a much higher (up to 10x) hit ratio compared to block-based ones, allowing the application to experience the lower latency of stacked DRAM.

To understand the spatial locality of the emerging applications in the context of large caches, we examine the page density for each workload for a page size of 2KB while varying the cache capacity (Figure 2.6 and Figure 2.7). We define page density as the number of demanded 64-byte blocks within the page. Not only do scale-out workloads exhibit high page density, but they also show increase in page density as the cache capacity increases. This can be contributed to the longer on-chip residency of pages, which at larger cache sizes reaches hundreds of milliseconds, leaving more time for data to be accessed within a page. As the cache grows in capacity, the number of high-density pages increases while the number of low-density pages decreases, resulting in a bimodal distribution for some of the workloads. The Multiprogrammed desktop workload and Software Testing, on the contrary, do not show a regular trend. Our analysis reveals that, due to the small datasets of these applications, a 512MB cache captures their entire working sets, at which point most of the dense pages become cache-resident, while the few remaining pages that are constantly fetched and evicted exhibit lower density.

The wide variations in spatial locality among the workloads shows that no single fetch unit size can simultaneously exploit the available spatial locality while using bandwidth and storage efficiently [62]. Among the workloads with lower page density, the highest fraction of the pages

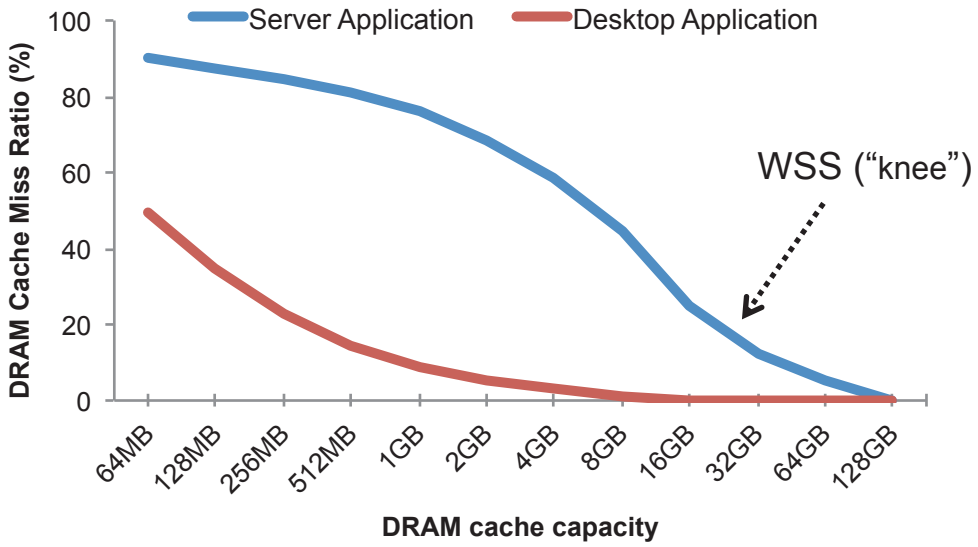

Figure 2.5 – Miss ratio of a block-based cache design for TPC-H database queries on a machine with 128GB of main memory as compared to a mix of SPEC2006 INT applications).

are singleton pages, which only have a single block accessed. While possibly reused in L1 and L2 caches, these blocks are rarely reused in the DRAM cache (less than 5% of the time), resulting in high bandwidth and capacity losses for page-based caches.

2.2.4 Block-Based vs. Page-Based

Figure 2.8 and Figure 2.9 compare block-based and page-based caches in terms of the miss ratio and bandwidth demands, respectively. The bars are plotted in a stacked fashion. For instance, the bar showing the miss ratio at 64MB for Data Serving (Figure 2.8) indicates that the miss ratio for the page-based cache and block-based cache is 18% (white part), and 62% (white and dark gray parts together), respectively. The same representation is used in Figure 2.9 to represent the off-chip bandwidth demands of the three cache organizations.

For all workloads, the page-based cache achieves up to an order of magnitude lower miss ratio, as expected, due to the high page access density. The exception is Data Analytics, which at 64MB and 128MB shows very low page access density, giving the block-based cache considerable capacity advantages. We observe the oposite trends on the bandwidth side, as Figure 2.9 demonstrates. The block-based cache achieves lower off-chip traffic, while the page-based increases the off-chip traffic by up to an order of magnitude compared to the baseline system that has no DRAM cache.

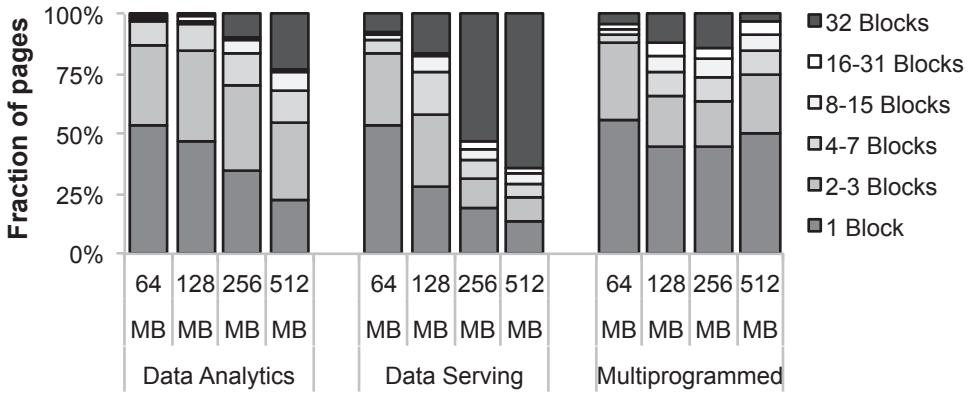

Figure 2.6 – Page density as a function of cache size for Data Analytics, Data Serving and the Multiprogrammed workloads.

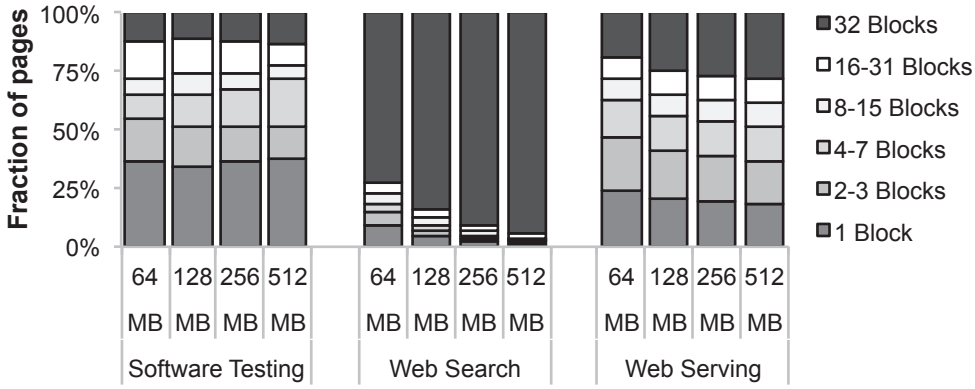

Figure 2.7 – Page density as a function of cache size for Software Testing, Web Search, and Web Serving.

In summary, block-based and page-based die-stacked DRAM cache designs trade off the effective hit ratio, and consequently, the effective memory latency for off-chip bandwidth, whereas server applications demand both low latency and efficient use of off-chip bandwidth.

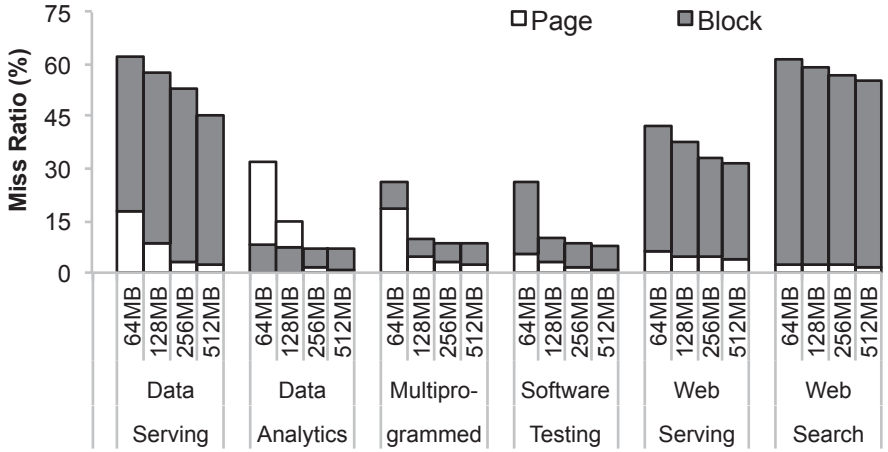

Figure 2.8 – Miss ratio in block-based and page-based DRAM cache designs.

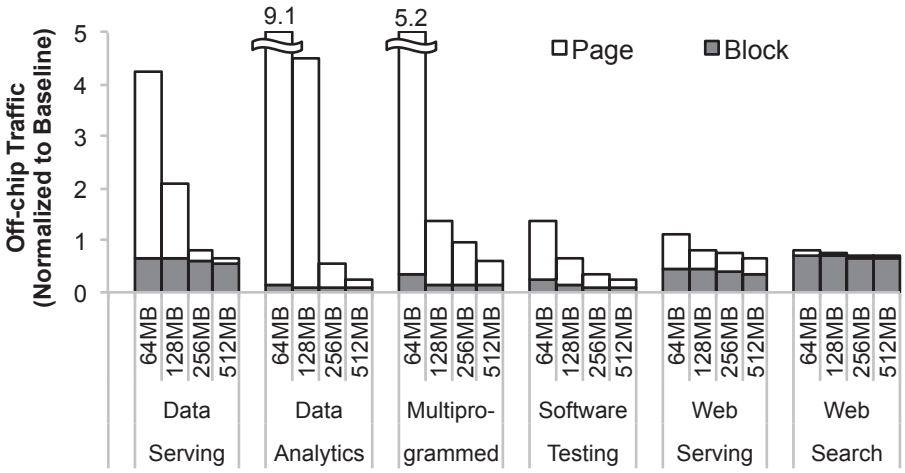

Figure 2.9 – Off-chip traffic in block-based and page-based DRAM cache designs normalized to the baseline system without a DRAM cache.

3 Footprint Cache

Block-based and page-based designs show complementary properties. On one hand, block-based designs are much more efficient in using cache capacity and off-chip bandwidth, but suffer from low hit rates. Their tag array is huge and must be stored in DRAM at the cost of either high latency or associativity. On the other hand, page-based caches provide high hit rates and small and arbitrarily associative SRAM-based tag storage. However, they severely misuse the precious off-chip bandwidth resources, and as such are not a feasible option.

Our goal is to preserve the properties of page-based designs, but without the unnecessary traffic and with better capacity management. Toward that goal, in this chapter we propose Footprint Cache, which mitigates most of the bandwidth and capacity problems of page-based designs and manages to get the best of the page-based and block-based designs.

3.1 Footprint Cache

While the block-based and page-based designs show complementary properties, never achieving the same goals, together they could meet all of the requirements of an ideal die-stacked cache as summarized in Section 2.1.1. The page-based design demonstrates superior properties overall, but due to its excessive off-chip traffic overheads, it is not a feasible option. Ideally, we would like to achieve the properties of the page-based design, but without the unnecessary traffic and with better capacity management. Our proposal, Footprint Cache, uses a page-based organization, but identifies the blocks that will be demanded by the cores during a page's on-chip residency. It then fetches only those blocks at the page allocation time, eliminating the unnecessary off-chip and on-chip traffic. Footprint Cache further identifies pages that have the fewest useful blocks and show no reuse, and neither allocates entries in the

cache for such pages nor fetches them. Footprint Cache instead fetches 64-byte blocks from such pages, one by one and only on demand, and forwards them to the requestor, bypassing the cache. Such pages account for a significant fraction of all pages that are fetched and are the biggest contributor to the capacity waste. Thus, Footprint Cache mitigates both the bandwidth and capacity problems of page-based designs and manages to get the best of the both designs.

Footprint Cache decouples the cache allocation unit from the fetch unit, similar to sub-blocked (or sectored) caches, allocating large pages while fetching 64-byte blocks. The set of useful blocks accessed during page's on-chip residency constitute the page's footprint. Upon a miss in the cache, a new page is allocated and the whole page's footprint is fetched at once from the main memory. To detect the useful blocks, we leverage prior work on spatial correlation [62] and design a simple and highly accurate predictor that identifies the page's footprint. As a result, Footprint Cache achieves high hit ratios, comparable to those of page-based approaches, and low off-chip traffic as in conventional block-based caches. Because the whole footprint is fetched and evicted at once, Footprint Cache enforces high DRAM access locality both for the off-chip and stacked DRAM, thus allowing for lower DRAM access latency. Last, but not least, Footprint Cache allows for a small and fast, SRAM-based tag array due to its large allocation unit. In summary, Footprint Cache meets all of the requirements of an ideal die-stacked cache as summarized in Section 2.1.1.

3.1.1 Footprint Prediction

To achieve the desired properties, Footprint Cache relies on the footprint predictor. The accuracy of the footprint predictor is crucial for both performance and energy efficiency. The predictor must have high coverage, ideally predicting all the blocks that will be later demanded. Every unpredicted block that is demanded later would result in a cache miss and consequently, in performance and energy loss. We call such an event underprediction. While it is important to correctly predict as many blocks as possible, it is essential that the predictor has minimal overprediction rate. Overpredictions represent blocks that are fetched but not used prior to eviction, and their transfer to and from main memory merely wastes off-chip and on-chip bandwidth and energy. An example of a system with no overpredictions is a sub-blocked cache, which allocates pages but fetches every block on demand. However, such a system would have the maximum number of underpredictions, as it would experience a miss for each demanded block in a page. A system with no underpredictions would be a page-based cache, that fetches all the data from the demanded page at once. Fetching all the blocks is, however, an even worse solution, as it produces the maximum number of overpredictions,

saturating the off-chip bandwidth. Minimizing both the underprediction and overprediction rates at the same time is a challenging task for a predictor. To achieve the two conflicting goals, Footprint Cache relies on the observation that there is a high correlation between data and the code that accesses those data. For instance, server software exposes a well-defined interface with only several functions for accessing their structured datasets — e.g., get and set methods of a class or various data structure iterators. Traversals of data structures require repetitive calls to these functions, resulting in recurring memory access patterns. The access pattern observed from the first call to such a function can be used to predict the memory access patterns of its subsequent calls. This fundamental property has been exploited in various contexts [40], mostly for data prefetching [6, 38, 62] and speculating on data granularity [37, 39, 70]. Footprint Cache achieves high prediction accuracy by monitoring the code that accesses data residing in the cache. The first instruction that accesses a page provides valuable information about the data that the page contains and is a good indicator of future accesses within that page [62], due to regular and repetitive layouts of data structures. By observing which blocks the code further accesses and by remembering that information, we can later predict, with high accuracy, which blocks will be demanded when another page, possibly previously unvisited, is accessed by the same piece of code [62]. Prediction based on the first instruction that accesses a page is highly accurate provided that data structures always have the same layout within a page. However, this is not necessarily the case for all the workloads and page sizes. To account for various possible data structure alignments across different pages, we base our prediction mechanism not only on the instruction that caused the page miss, but also on the distance between the requested block and the beginning of the page, which we call offset. The combination of PC and offset (noted as PC & offset) provides near-perfect prediction accuracy at low overhead. Previous work has a detailed study of other related prediction mechanisms and their trade-offs in the context of data prefetching [62].

3.1.2 Capacity Optimization

Our analysis shows that a significant fraction of pages (more than a quarter, on average) contain only a single useful block. Such pages often account for the largest share of the pages in workloads with low spatial locality. Moreover, we find that, on average, 95% of these pages show no reuse in the DRAM cache, and therefore waste capacity. We call such pages singleton pages. Footprint Cache is able to identify such pages with almost perfect accuracy, thanks to the fact that these pages are accessed by a single instruction, and obviously, with a single offset. Footprint Cache does not allocate entries for such pages. The requested block is directly forwarded to the higher cache level and its subsequent eviction is not tracked. This

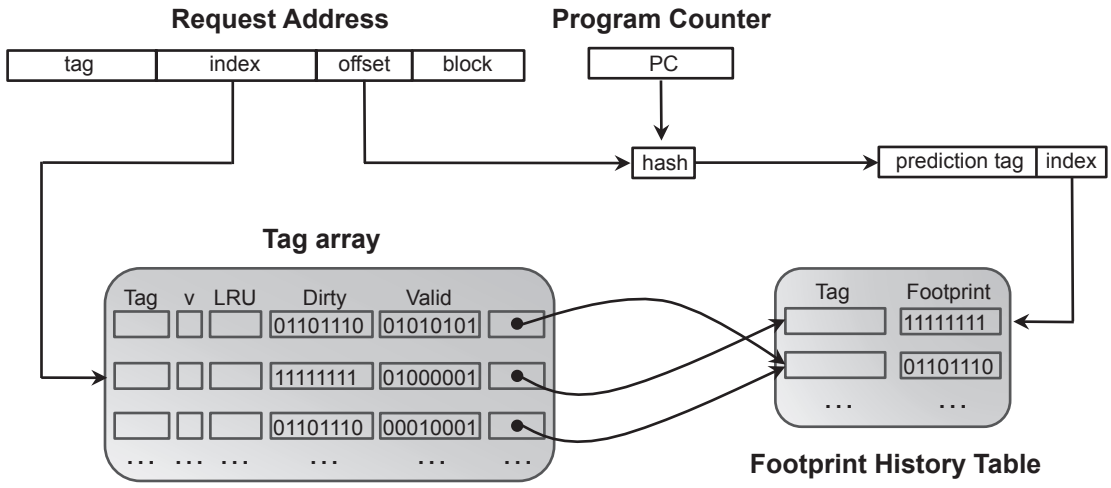

Figure 3.1 – Footprint Cache tag array and Footprint History Table (FHT).

mechanism increases the effective cache capacity reusing the existing footprint prediction mechanisms. It avoids eviction of useful pages, thus allowing for even higher hit ratios. The optimization plays an important role at smaller cache sizes, for which efficient use of cache capacity matters the most.

3.2 Footprint Cache Design

Footprint Cache tightly couples the footprint prediction mechanism with the tag array. The footprint predictor uses the information from the tag array to learn page footprints, storing the footprints into a history table upon page evictions and using the footprint information upon page misses to fetch useful blocks. We next detail the predictor design and its integration with the tag array, and we further explain the prediction history management.

3.2.1 Footprint Cache Tag Array

Similar to page-based and sub-blocked caches, Footprint Cache requires almost two orders of magnitude smaller tag arrays, which can be kept within reasonably small SRAM storage. The tag array is organized as a set-associative structure; set and way pairs directly determine physical addresses of pages cached in DRAM. The size of a page is selected to match commonly used DRAM row sizes (e.g., 1-4KB),[1] keeping in mind the impact on the prediction accuracy and tag overhead. Similarly to sub-blocked caches, Footprint Cache keeps two bit vectors to

[1] The exact matching of the page size and DRAM row size is not crucial for the proposed technique

track valid and dirty blocks, and a page-level valid bit (Figure 3.1). In the multicore system we evaluate, the tag array is distributed into four tiles organized in a page-interleaved fashion, each tile being responsible for a partition of the DRAM cache. We use page-interleaved placement in the Footprint Cache tag array for efficiency.[2] Each tag tile is attached to an on-chip memory controller that controls a 128-bit (16B) TSV channel associated with the corresponding partition of the DRAM cache. The only additional overhead introduced in the tag array is a pointer that links pages to the prediction history described in Section 3.2.2.

3.2.2 Prediction History

The prediction history, shared by all the tag tiles, is kept as a separate tiled structure, called Footprint History Table (FHT). The FHT is a set-associative structure indexed by PC & offset pairs, storing predicted footprints for PCs & offsets that trigger page misses. Each entry keeps a tag identifying the PC & offset key, while the corresponding prediction information is kept as a bit vector, determining the footprint associated with the key [62]. The FHT size is independent of the workload's dataset, as it holds only a small fraction of the workload's instruction footprint, measured in kilobytes. The FHT is updated upon every page eviction with the most recent footprint generated during the page's on-chip residency. The FHT is accessed only in case the page containing a requested address is not found in the cache. Upon a page miss, which we call a triggering miss, the table is queried by the PC of the instruction that caused the miss and the offset bits of the request address, returning the predicted footprint. The triggering miss is served by the off-chip memory and a page eviction takes place. If the PC & offset pair exists in the FHT, which is the common case, the rest of the blocks, encoded in the predicted footprint, are fetched from memory, and a pointer to the FHT entry is stored in the tag entry. If the FHT does not contain the PC & offset pair, which mostly happens at the beginning of the program execution, a new FHT entry is allocated, and a pointer to the new FHT entry is stored in the tag entry.

Upon a cache eviction, a bit vector containing the blocks that were indeed demanded by the cores, is sent to the FHT, using the pointer created during the allocation of the page. This bit vector gives feedback to the prediction mechanism, correcting mispredictions if any, and keeping the FHT in harmony with the workload's execution phase. As we do not store the PC in the tag entry, but only the pointer to the FHT entry, it is possible, although unlikely, that the pointer becomes stale, as a result of an FHT eviction. While this may affect prediction accuracy,

[2] The placement policy in the upper-level caches, however, is not affected by this design decision and can use either block- or pageinterleaving, with no observable performance difference for our workloads. In this work we assume block-interleaving.

Dirty-valid bits	Semantics
00	the block is not in the cache
01	the block is valid, clean and not demanded yet
10	the block is valid, clean and has been demanded
11	the block is valid, dirty and demanded

Table 3.1 – Block state encoding

we could not see any observable impact. The reason is, unlike the cached data, the content of the FHT is stable, therefore, such situations almost never happen. For practical reasons, similar to the tag array, the FHT is designed as a tiled structure, and its entries are distributed based on PC & offset keys of incoming requests, not necessarily matching the distribution criterion for the tag tiles (physical addresses). The mismatch can result in frequent accesses to neighboring FHT tiles. However, this does not have any timing impact due to the FHT's negligible access latency, which is not on the critical path of memory accesses.

3.2.3 Footprint Generation

Blocks that are placed in the cache must be set to the valid state regardless of whether they are demanded by a core or predicted by the predictor. On the contrary, providing correct feedback to the FHT requires a distinction between the blocks that are demanded and the ones that are in the cache but were not demanded during the page's on-chip residency (overpredictions). Upon a core's request, whether a hit or a miss, the corresponding block should be marked as demanded. To make this distinction possible without additional storage, we reuse the existing valid and dirty bits to create the block state encoding listed in Table 3.1. We are able to achieve this encoding, because a block cannot be in a dirty state if it has not been used by a core. The high order bits for all block states together represent the demanded bit vector (i.e., the page's footprint), used to update the FHT upon a page eviction. The exact matching of the page size and DRAM row size is not crucial for the proposed technique.The placement policy in the upper-level caches, however, is not affected by this design decision and can use either block- or page interleaving, with no observable performance difference for our workloads. In this work we assume block-interleaving.

3.2.4 Capacity Optimization

Footprint Cache increases the effective cache capacity by avoiding allocation of singleton pages — i.e., pages with a single useful block. In our design, if the footprint bit vector corresponding

to a missing PC & offset pair in the FHT has a single bit set, the corresponding cache entry is allocated neither in the cache nor in the tag array. Not allocating entries for singleton pages in the tag array implies that the FHT would never receive feedback regarding its single block predictions. Once a page is classified as singleton, it would remain singleton until its FHT entry is evicted, regardless of any mispredictions or changes in the application's behavior. To avoid this scenario, we add a small table, which is further partitioned and co-located with the tag array tiles, called Singleton Table (ST). In case the FHT predicts a singleton block, the page is not allocated in the cache, but an ST entry is allocated, containing the PC, offset, and the page tag. The ST is indexed by a page tag, and only upon a page miss. Finding the tag in the ST, but with a different offset, implies that there is a second access to a page that was originally predicted to be a singleton page (underprediction). In this case, the new tag and FHT entry is allocated with the PC & offset information found in the ST, and the corresponding ST entry is invalidated. An entry stays in the ST until a second access to the page, or until its eviction. The ST has negligible overhead (3KB, 512 entries), but allows for more accurate and adaptive prediction of singleton pages. This is important for smaller caches, where pages could be misclassified as singleton due to their short on-chip residency and capacity conflicts.

3.3 Methodology

We evaluate Footprint Cache in terms of performance, energy efficiency, and hardware overhead in the context of high-throughput, bandwidth-demanding scale-out processors, which can benefit from the die stacking technology [50]. We compare Footprint Cache to a state-of-the-art implementation of conventional block-based DRAM caches as well as to page-based DRAM caches.

3.3.1 Baseline System

We evaluate Footprint Cache using scale-out processors. The scale-out processor architecture splits the available chip resources into multiple stand-alone servers, called pods [50], which are multicore configurations designed to match the needs of scale-out workloads and deliver the highest throughput for given silicon real-estate. A pod is a complete server that tightly couples a number of cores to a modestly-sized last-level cache using a fast interconnect. Replicating the pod to fill the die area yields processors that have optimal performance density. Moreover, as each pod is a stand-alone server, scale-out processors avoid the expense of global (i.e., inter-pod) interconnect and coherence. These features synergistically maximize throughput, lower design complexity, and improve technology scalability. We model a chip in

Technology	20nm, 0.85V, 3GHz
CMP Organization	16-core Scale-Out Processor pod
Core	ARM Cortex-A15-like, 3-way OoO @3GHz
L1-I/D caches	64KB, split, 64B blocks 2-cycle load-to-use latency
L2 cache per pod	4MB, unified, 16-way, 64B blocks, 4 banks, 13-cycle hit latency
Interconnect	16x4 crossbar
Off-chip DRAM	16-32GB, one DDR3-1600 (800MHz) channel 8 banks per rank, 8KB row buffer
Stacked DRAM	DDR-like interface (1.6GHz) 4 channels, 8 banks/rank, 8KB row buffer, 128-bit bus width
t_{CAS}-t_{RCD}-t_{RP}-t_{RAS} t_{RC}-t_{WR}-t_{WTR}-t_{RTP} t_{RRD}-t_{FAW}	11-11-11-28 39-12-6-6 5-24

Table 3.2 – Architectural system parameters.

20nm technology with on-chip supply voltage of 0.85V, assuming area and power budgets of 250mm2 and 105W, respectively. The chip features six 16-core pods and six single-channel DDR3-1600 interfaces. Area and power estimates are measured by scaling down the published data at 40/45nm process technology [26, 50], following ITRS projections. Table 3.2 summarizes the parameters for the highest-performance baseline chip that can be designed under the area, power, and bandwidth constraints described above [50].

3.3.2 DRAM Cache Organizations

Footprint Cache parameters are listed in Table 3.3. We use the open-page policy both for the on-chip and off-chip DRAM, as our design exhibits near-optimal data locality for all off-chip DRAM accesses, onchip DRAM fills and on-chip DRAM evictions, while data locality for on-chip read/write requests(i.e., cache hits) is workload-dependent. We use 2KB pages and 2KB address-interleaving for on-chip memory channels. The FHT has 16K entries (144KB) while the ST has 512 entries (3KB).

Block-based caches are modelled after a state-of-the-art proposal that provides an elegant solution to tag handling, by co-locating tags from one cache set with all the blocks in that

Cache capacity (MB)	64	128	256	512
Tag SRAM storage (MB)	0.4	0.8	1.58	3.12
Tag latency (cycles)	4	6	9	11

Table 3.3 – Footprint Cache parameters.

Cache capacity (MB)	64	128	256	512
#MissMap entries	192K	192K	192K	288K
MissMap SRAM storage (MB)	1.95	1.95	1.95	2.92
MissMap associativity	24	24	24	36
MissMap latency (cycles)	9	9	9	11

Table 3.4 – Block-based cache parameters.

Cache capacity (MB)	64	128	256	512
Tag SRAM storage (MB)	0.22	0.44	0.86	1.69
Tag latency (cycles)	4	5	6	9

Table 3.5 – Page-based cache parameters.

set in the same DRAM row [48]. For 2KB DRAM rows, it is possible to fit 29 64- byte data blocks in one row, using the three remaining blocks for the corresponding tags. A crucial optimization to avoid two DRAM accesses per cache access includes intelligent memory controller scheduling [48]. An access to a cache block involves a single row activation, and one column activation signal (CAS) to read the tags, a one-cycle tag lookup to determine the location of the data block, another CAS to retrieve/write to the data block, and a third CAS to write back the updated tags. The last CAS is required as the tags must be updated, however, our evaluation does not account for that latency, as we assume the algorithm can be reengineered to take the tag updates off the critical path.

The exact location of the requested block is stored in DRAM tags and determined after the row is activated and the tags blocks are read. However, the presence of the block in the cache is tracked by a compact structure called *MissMap*, which keeps track of the cached data at 4KB granularity, storing bit vectors that determine only the presence of blocks within a page. If the requested block is found in MissMap, the cache is accessed, otherwise the request is serviced by memory. This optimization avoids unnecessary tag lookups in DRAM in case of cache misses [48].

Table 3.4 lists the parameters we used for evaluation of the proposed block-based DRAM cache. By dropping coherence bits from the tags, we were able to achieve higher storage density

with 30 data blocks per DRAM row, and only two tag blocks, assuming ARM's extended 40-bit physical addressing. This also increased the cache associativity from 29 to 30, and reduced the latency of tag retrieval, which is on the critical path. Within proposed 2MB of SRAM dedicated to MissMap [48], we were also able to fit more MissMap entries. However, with larger cache capacity, we observed performance degradation due to a high number of MissMap evictions, which in return generate many dirty cache evictions. Although this operation is not on the critical path, we found that MissMap evictions interfere with regular read/write cache requests as well as with cache fills of other blocks, contending for the same DRAM bank. The reason is that MissMap keeps and evicts spatially consecutive blocks, which are all in different cache sets, and therefore, in different DRAM rows, causing an excessive number of row activations. To avoid this situation, we increased the size of the MissMap structure by 50% to evaluate 512MB caches. We use close-page policy both for off-chip and on-chip DRAM, as we found that it performs better due to the complete absence of data locality in the die-stacked DRAM, and 64-byte address interleaving between memory channels to increase DRAM-level parallelism.

Page-based cache parameters are listed in Table 3.5. We use the open-page policy both for the die-stacked and off-chip DRAM, as page-based caches exhibit the optimal data locality in the off-chip DRAM, as well as the optimal data locality in die-stacked DRAM for fills and evictions, while data locality for on-chip read/write requests is workload-dependent. We use 2KB pages and 2KB address-interleaving for on-chip memory channels.

Row-buffer management policies and address-mapping schemes are chosen for each evaluated system separately to allow for optimal performance and DRAM-level parallelism.

3.3.3 Workloads

Our scale-out workloads, which include Data Serving, Data Analytics, Software Testing, Web Serving, and Web Search, are taken from CloudSuite 1.0 [9, 12]. Their memory footprints exceed the available memory, which is 16-32GB. As a reference, we also simulate a multiprogrammed desktop workload that consists of SPEC INT2006 applications using the reference input set. We run a mix of different applications and inputs to utilize all available cores. These workloads, when running on the baseline chip we consider, require between 0.6-1.6GB/s of off-chip bandwidth per core, or 60-150GB/s for the whole chip. While today's dominant server processors, which integrate a handful of fat cores, are not able to utilize the available bandwidth when running these workloads [12], our design utilizes and even exceeds the bandwidth budget [50].

3.3.4 Simulation Infrastructure

We evaluate Footprint Cache using a combination of trace-driven and cycle-accurate full-system simulations of a scale-out pod using Flexus [68]. Flexus extends the Simics functional simulator with timing models of out-of-order cores, caches, on-chip protocol controllers, interconnect, and DRAM. Flexus models the SPARC v9 ISA and is able to run unmodified operating systems and applications. The details of the simulated architecture are listed in Table 3.2. Our trace-based analyses use memory access traces collected from Flexus with in-order execution, no memory system stalls, and a fixed IPC of 1.0. For each workload, we collect a trace of 20-40 billion instructions per core and use one half to two thirds of the trace for cache warm-up prior to collecting the experimental results. For cycle-accurate simulations, we use the SMARTS sampling methodology [69]. Our samples are drawn over an interval of 10 seconds of simulated time, with 400-800 equidistant measurements. For each measurement, we launch simulations from checkpoints with warmed caches and branch predictors, and run 300K cycles (2M cycles for Data Serving) to achieve a steady state of detailed cycle-accurate simulation prior to collecting measurements for the subsequent 150K cycles (400K for Data Serving). To measure the performance of scale-out workloads, we use the ratio of the aggregate number of application instructions committed (i.e., summed over the 16 cores) to the total number of cycles (including the cycles spent executing operating system code); this metric has been shown to accurately reflect the overall system throughput [68]. For the multiprogrammed workload, we calculate the IPC improvement for each core independently and report the geometric mean. Performance measurements are computed at a 95% confidence level and an average error below 3%. To model on-chip and off-chip DRAM performance and power, we use two separately adapted and configured instances of DRAMSim2 [58], parametrized with data borrowed from commercial DDR3 device specifications. We report all results for one 16-core pod.

3.4 Results

3.4.1 Spatial Characterization

In Chapter 2 we already examined the page density of the server workloads as a function of cache size. The increase in page density with the cache capacity, demonstrated in Figure 2.6 and Figure 2.7, has an interesting implication on the footprint predictor effectiveness. Compared to the prior work [62] that uses a similar predictor to prefetch into block-based L1 caches, our predictor is more effective due to (1) the higher prediction opportunity at this

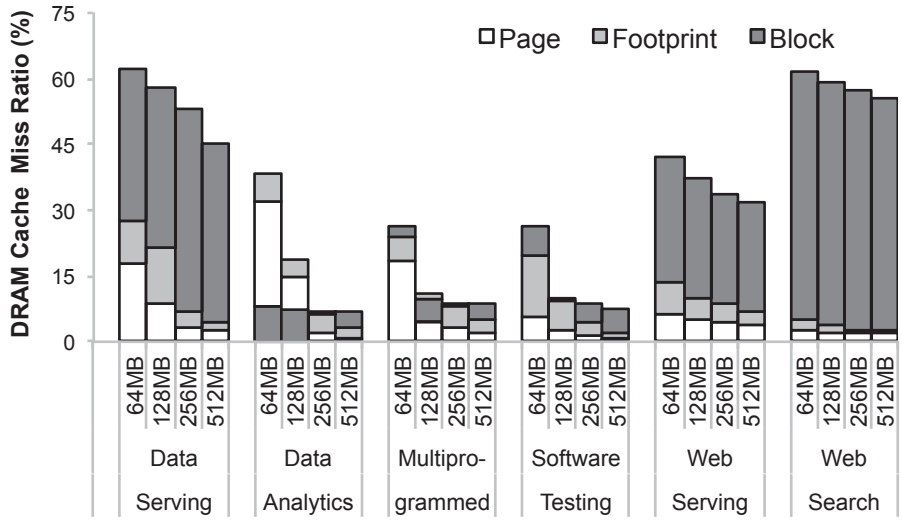

Figure 3.2 – Miss ratio of Block-based, Footprint, and Page-based cache organizations, normalized to a baseline system without a DRAM cache.

cache level (more blocks to predict per each obligatory page miss), and (2) a larger fraction of fully-accessed pages that are easier to identify due to their simple access patterns (sequential accesses). However, not all the workloads show high page density. In fact, Figure 2.6 and Figure 2.7 demonstrate wide variations in spatial locality among the workloads. Thus, no single fetch unit size can simultaneously exploit the available spatial locality while using bandwidth and storage efficiently [62]. Among the workloads with lower page density, the highest fraction of the pages are singleton pages, which only have a single block accessed. While possibly reused in L1 and L2 caches, these blocks are rarely reused in the DRAM cache (less than 5% of the time), resulting in high bandwidth and capacity losses for page-based caches. Footprint Cache successfully detects such pages and avoids their allocation in the cache. The high degree of spatial locality observed in most of the scale-out workloads implies that page-based caches achieve higher hit ratios compared to the block-based ones; due to their large fetch unit, they always experience a single miss per page. Unfortunately, fetching whole pages is a brute-force approach to achieving high hit ratios and it comes at an unacceptable bandwidth cost, which is the most severe for low-density pages.

3.4.2 Coverage and Off-Chip Bandwidth

Footprint Cache learns and predicts the footprint of each page and hence it is able to achieve high hit ratios and eliminate the fetch of blocks that will not be accessed during the residency of the page in the cache. Figure 3.2 and Figure 3.3compare Footprint Cache with block-based

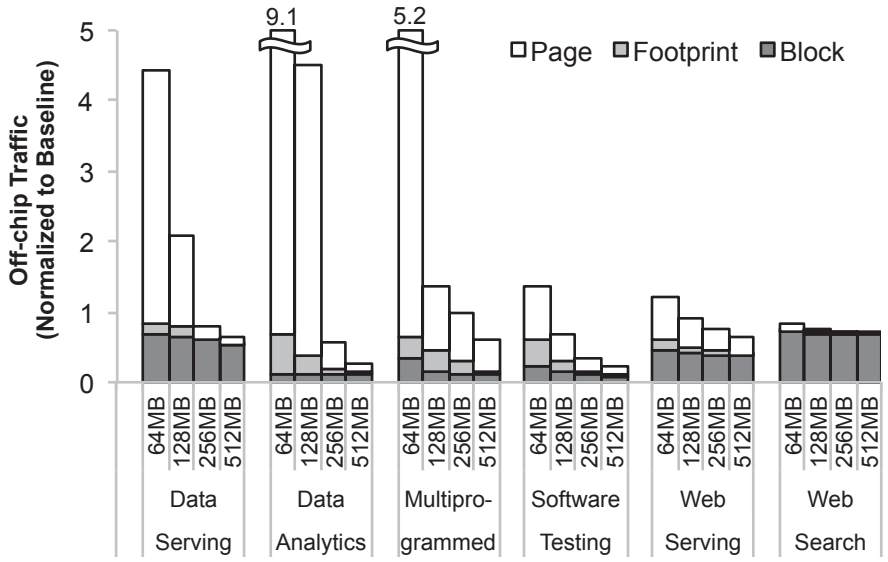

Figure 3.3 – Off-chip bandwidth requirements of Block-based, Footprint, and Page-based cache organizations, normalized to a baseline system without a DRAM cache.

and page-based caches with respect to the miss ratio and bandwidth demands, respectively. The bars are plotted in a stacked fashion. For instance, the bar showing the miss ratio at 64MB for Data Serving (Figure 3.2) indicates that the miss ratio for the page-based cache, Footprint Cache and block-based cache is 18% (white part), 27% (white and light gray parts together) and 62% (white, light gray and dark gray parts together), respectively. The same representation is used in Figure 3.3 to represent the off-chip bandwidth demands for the three cache organizations.

For all workloads, the page-based designs achieves up to an order of magnitude lower miss ratio, as expected due to the high page access density. The exception is Data Analytics, which at 64MB and 128MB shows very low page access density, giving the block-based cache considerable capacity advantages. As expected, Footprint Cache always achieves a miss ratio close to the page-based cache. Only Software Testing, at smaller caches sizes, shows significantly larger miss ratios compared to the page-based design, but still performing better than the block-based design. The reason is that Software Testing performs symbolic execution as part of software testing [9] and as such does not have a static, well-structured dataset. On the contrary, it creates its dataset on-the-fly, throughout the whole program execution, which interferes with the prediction mechanism.

As expected, we observe the opposite trend on the bandwidth side, as Figure 3.3 demonstrates. The block-based cache achieves the lowest off-chip traffic, while the page-based increases the

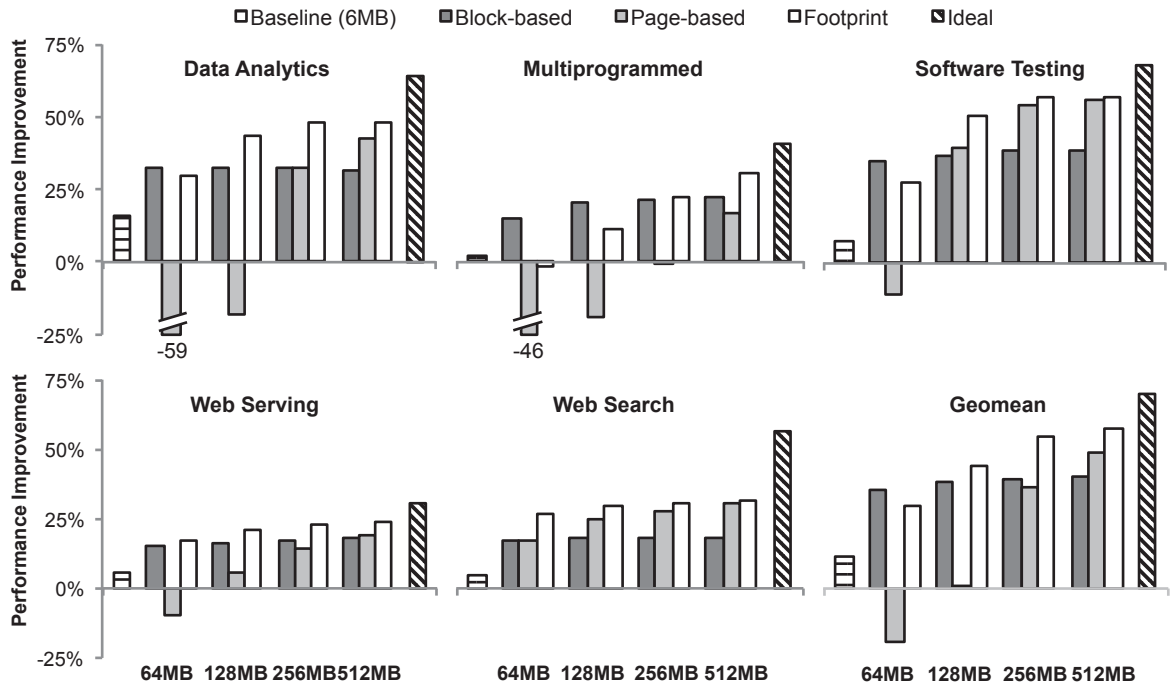

Figure 3.4 – Performance improvement of various designs over the baseline system for the Data Analytics and Multiprogrammed workloads.

off-chip traffic by up to an order of magnitude compared to the baseline. Footprint Cache, on the contrary, demands almost the same bandwidth as the block-based design by eliminating most of the unnecessary traffic.

3.4.3 Performance

Figure 3.4 compares performance of the three cache designs at various cache sizes for all workloads except Data Serving. We plot the results for Data Serving in Figure 3.5 due to the large difference in scale, caused by the excessive bandwidth requirements of this workload. The block-based design provides the greatest initial performance boost at 64MB, which is mostly contributed to the significant cut in off-chip traffic. However, the design fails to deliver considerable further improvements, due to its high and steady miss ratio, as shown in Figure 3.2. The page-based design initially suffers from a considerable performance loss due the excessive off-chip traffic it causes. As the cache capacity increases, it quickly recovers due to fewer misses and decreased pressure on off-chip bandwidth. On the contrary, Footprint Cache shows steady performance improvement across all cache sizes, outperforming the other

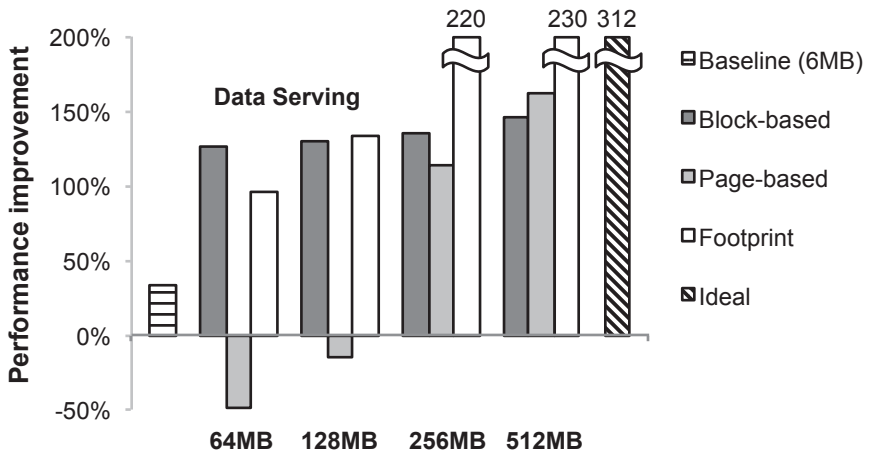

Figure 3.5 – Performance improvement of various designs over the baseline system for Data Serving.

two designs, consistently matching our findings from Figure 3.2 and Figure 3.3. For some of the workloads, we observe a slight advantage of the block-based design over Footprint Cache at smaller cache sizes, due to its superior capacity management.

As the stacked DRAM cache requires on-chip SRAM storage for the tags (under 2MB for the 512MB stacked cache), we also consider a baseline system with additional L2 capacity to compensate for the difference in total on-chip storage. This enhanced baseline provides negligible benefit on scale-out workloads, as expected based on earlier research results [12, 50]. Across all workloads, Footprint Cache is able to deliver 82% of the system performance of an Ideal cache — i.e., a cache that never misses and has no tag overheads (die-stacked main memory).

3.4.4 Sensitivity to Page Size and History Size

Figure 3.6 compares the predictor accuracy assuming various page sizes, showing a fraction of the blocks that are successfully predicted, the blocks that are not predicted (underpredictions), and the blocks that are overpredicted. While for most of the workloads 1KB and 2KB pages are the best options, larger pages might be desirable as they provide further tag storage reduction. Larger pages, however, require larger footprint history, due to an increase in number of PC & offset combinations per instruction. In this work, we find 2KB to be the sweet spot, considering the trade-off between the accuracy and storage overheads.

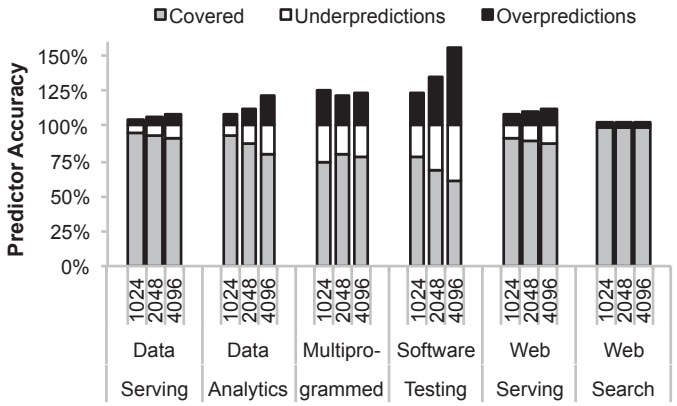

Figure 3.6 – Predictor accuracy sensitivity to the page size, for a 256MB cache with 16K FHT entries.

Because the Footprint Cache prediction mechanism relies on the missing instruction, its history storage requirements are independent of the dataset size. The prediction history, captured by the FHT, contains only the fraction of the application's instruction working set that causes page misses in the DRAM cache. Thus, its size is small and its content is stable. Figure 3.7 illustrates the Footprint Cache hit ratio sensitivity to the number of history entries. In this work we assume 16K FHT entries, which require 144KB of SRAM storage, but other trade-offs are possible with, as Figure 3.7 shows, minimal performance impacts.

3.4.5 Impact of Capacity Optimization

As we saw in Figure 2.6 and Figure 2.7, singleton pages account for a quarter of the pages in the cache, on average. Their elimination allows for a proportionate increase in the effective cache capacity, ultimately resulting in a 10% reduction in the miss rate, on average. The miss rate reduction is in accordance with our observation that miss rates for scale-out workloads follow a power low [21] (the miss rate vs. capacity relationship can be also estimated from Figure 3.2).

3.4.6 Energy Implications

Figure 3.8 compares the three designs in terms of dynamic off-chip DRAM energy. All cache designs use 256MB of DRAM cache. Because the systems differ in performance, and therefore, in the rate at which they access off-chip memory, we present the energy per instruction normalized to the baseline system without a cache. The dynamic energy is broken down into

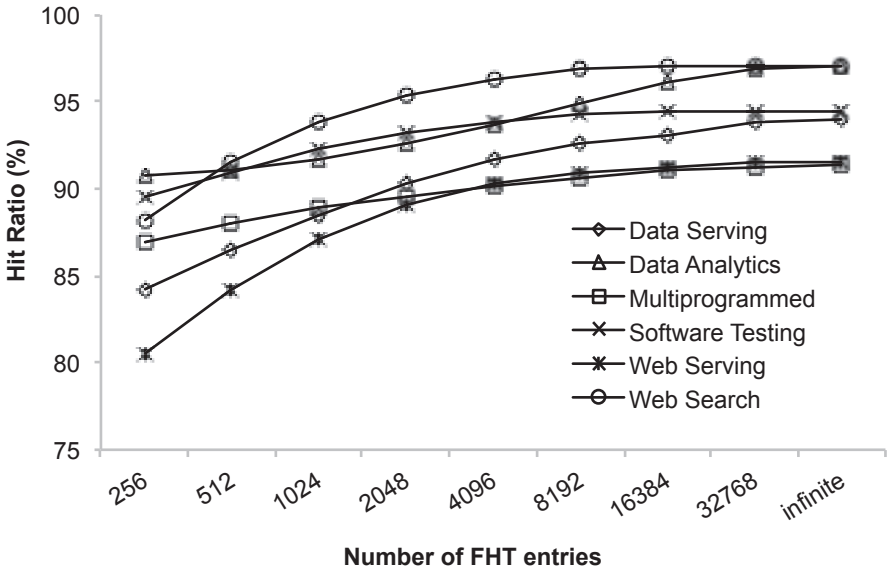

Figure 3.7 – Hit ratio sensitivity to the history size. The DRAM cache capacity is 256MB and the page size is 2KB.

activate/precharge energy, burnt for DRAM row manipulations, and burst energy, spent on reads and writes from an activated DRAM row.

All designs achieve significant energy reduction compared to the baseline system. As expected, the page-based design burns the most burst energy due to its high off-chip traffic per instruction. However, the page-based design exhibits the best DRAM access locality and the highest row-buffer hit ratio, significantly reducing the activate/precharge energy. On the contrary, the block-based design consumes the lowest burst energy due to its low off-chip traffic. However, as almost every read results in a row activation, they exhibit very high activate/precharge energy, which dominates the total dynamic energy.

Footprint Cache delivers the lowest off-chip DRAM energy per instruction. In particular, Footprint Cache is able to reduce both activate/precharge and burst energy thanks to its page organization and its high prediction accuracy, which allows for reducing offchip traffic significantly. Across all workloads, Footprint Cache reduces the total dynamic off-chip DRAM energy of the baseline by 78%, whereas the block-based and page-based designs reduce the energy by 71% and 69%, respectively.

We observe similar trends in stacked DRAM energy consumption. Figure 3.9 plots stacked DRAM energy consumption for various die-stacked designs normalized to the block-based design. Not surprisingly, the savings in activate/precharge energy for the page-based and

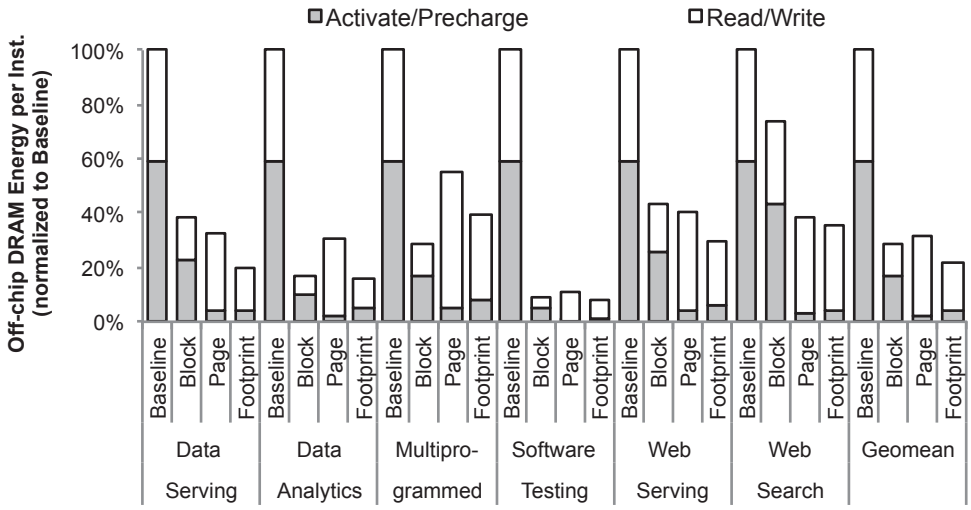

Figure 3.8 – Off-chip DRAM dynamic energy per instruction normalized to the baseline system.

Footprint Cache designs are not as low as in off-chip DRAM, despite the excellent row-buffer locality of cache fills and evictions. The reason is that regular read/write requests (cache hits) experience much fewer row-buffer hits for majority of the workloads. Overall, Footprint Cache reduces the total dynamic DRAM energy by 24% compared to the block-based design, whereas the page-based achieves only a 17% reduction.

3.4.7 Other Page-Based Proposals

We evaluated a recently proposed page-based cache system [30] that tracks the topmost accessed pages, called hot pages, that contribute to 80% of the total accesses. Only pages predicted to be hot are allocated in the cache and fetched at the page granularity. The prediction is based on the previous history of each page's behavior. The idea behind this approach is that only a small fraction of pages contribute to the majority of cache accesses. However, we could not make the same observation with scale-out workloads due to their vast data set, most of which is randomly distributed across memory, without forming a particular working set. Previous work also noted the same problem [48]. Figure 3.10 plots the amount of cache needed to capture a desired fraction of total accesses, assuming a perfect predictor, an ideal cache replacement policy and 4KB pages (4KB was found to be the optimal page size [30]). As we can see, even in the idealized case, to capture 80% of the pages we need caches over 1GB. While the proposed mechanism does not work well for our workloads, we find this work important and believe the idea of page-level filtering has a lot of potential for many applications, if equipped

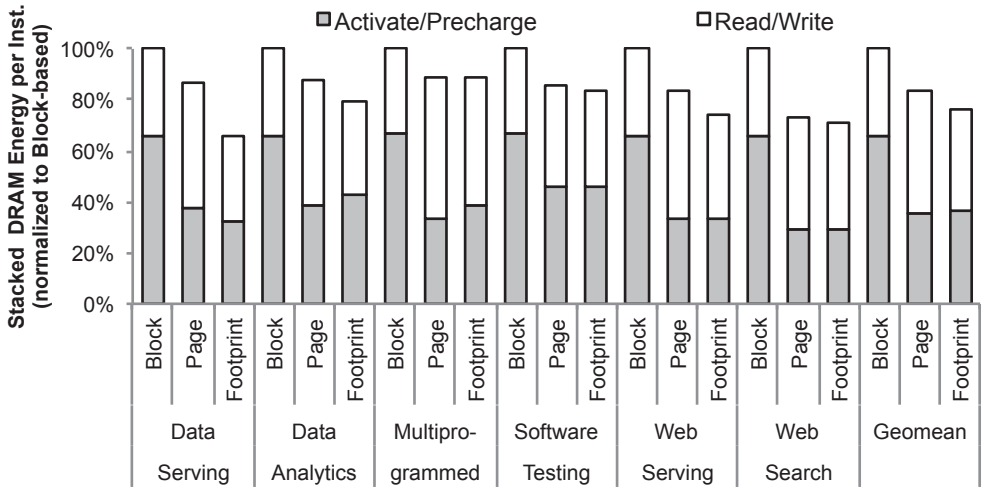

Figure 3.9 – Stacked DRAM dynamic energy per instruction normalized to the block-based design.

with a predictor that is dataset-independent, such as instruction-based ones. In fact, Footprint Cache uses a similar approach to eliminate singleton pages.

3.5 Discussion

3.5.1 Footprint Cache and Coherence

To facilitate the shared memory programming model, contemporary server processors provide hardware-enforced coherence at the chip level. Existing designs enforce coherence at the SRAM LLC level. Given this organization, the addition of the Footprint Cache does not entail any modifications to the underlying coherence protocol and implementation, as it sits below the level at which coherence is enforced.

In systems with multiple sockets, Footprint Cache can easily provide page-level coherence tracking [2, 5, 71] by extending tag entries with per-page coherence bits. Tracking coherence at fine granularity across sockets is not necessary for server workloads as they share little or no data [2, 12, 20].

3.5.2 Knowledge and Transfer of PC

Footprint Cache relies on the knowledge of instructions that cause page misses. Such information is typically not available in the last-level cache. Therefore, our design must extract

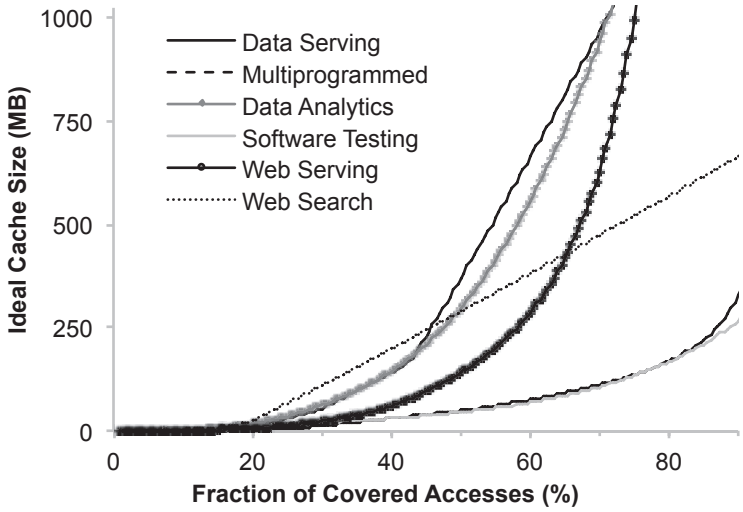

Figure 3.10 – Minimum size of an ideal cache needed to cover a given fraction of cache accesses.

the physical PC out of the pipeline and send it along with all memory requests.This requires carrying the physical PC throughout the pipeline for all memory instructions until the stage where the memory operands are known, and sending it to the memory hierarchy together with memory requests. Because a growing number of reserach proposals rely on such mechanisms and prove the superiority of PC-based speculation over the address-based, we expect future architectures to support this feature.

The PC information needs to be further transfered, along with read/write requests, through the on-chip network [67]. Because Footprint Cache does not track evictions from the higher-level cache, it does not need to store PC information at any cache level. The transfer of PC information via the on-chip network has no performance implications due to the underutilization of the network [66]. Such transfers, however, do have energy implications. We find that PC transfers increase the on-chip network power by 30mW per pod in the worst case, which is a negligible overhead.

In this work we assumed the knowledge of the complete program counter, but in fact this is not necessary. Because a small number of PCs (in the order of several thousands) is stored in the history table, a 16-bit XOR hash of the PC would be enough. Besides reducing the amount of data for transferring the PC information, Section 3.5.3 will show that using a 16-bit XOR hash of the PC can also substantially reduce the amount of SRAM storage dedicated to the Footprint History Table at a negligible accuracy loss.

3.5.3 SRAM Area Overhead

All the designs we discussed, including Footprint Cache, impose a multimegabyte SRAM overhead for tags and other metadata. We assume this area overhead will be compensated by the reduction in the number of off-chip memory channels, for all designs except the page-based, which exacerbates bandwidth demands. Still, as mentioned above, none of the designs we evaluate will gracefully scale to multi-gigabyte capacities, precisely due to the SRAM area overhead.

Regarding Footprint History Table, we saw in Section 3.4.4 that a 16K-entry FHT occupies around 144KB of SRAM storage. Each FHT entry contains a (PC, offset) pair that acts as a tag and a predicted pattern. This storage can be significantly reduced in a number of ways. Firstly, as Figure 3.7 demonstrates, FHT can be shrunk significantly by reducing the number of entries at a small cost in miss ratio. Secondly, we could keep a single pattern per PC along with the offset that corresponds to that pattern. Upon a request with the same PC but different offset, we could than rotate the pattern based on the relative difference between the two offsets [13]. FHT on average keeps two to three different (PC, offset) pairs per PC, and such an optimization could reduce the history size by 2-3x at a small accuracy loss [13]. Thirdly, we could change the fetch policy to all-or-nothing instead of fetching precise footprints [65], in which case there would be no need to store the whole pattern, but only a few flags indicating whether the page should be entirely fetched or not.

The first technique trivially reduces the number of entries in the history table. The second technique focuses on reducing the number of entries by keeping a single offset and approximating the pattern for other offsets. The third technique reduces the amount of space dedicated to patterns. However, the biggest contributor to the FHT storage overhead are FHT's tags, i.e., the (PC, offset) pairs. To tackle the tag overhead, we look at a tagless history solution, in which a (PC, offset) pair is XOR-hashed and the resulting hash is used to index an array of patterns. In this case, a 16K-entry FHT would occupy only 64KB of SRAM for designs with 2KB pages, and only 32KB for designs with 1KB pages. Figure 3.11 shows the effect of not storing the tags in the history table on hit ratio and off-chip traffic, normalized to the baseline solution that maintains full (PC, offset) pairs as tags. Both designs use a 16K-entry FHT, the cache size is 256MB and the page size is 1KB. We see that the relative change in both hit ratio and off-chip traffic is negligible. The 14-bit hash value used to index the tagless history is computed by appending the offset to non-zero bits of the PC[3], and the last 42 bits of the result are XOR-ed

[3] In the SPARC architecture we simulate, instructions are aligned at a 32B boundary, and the last two bits of a program counter are always zero

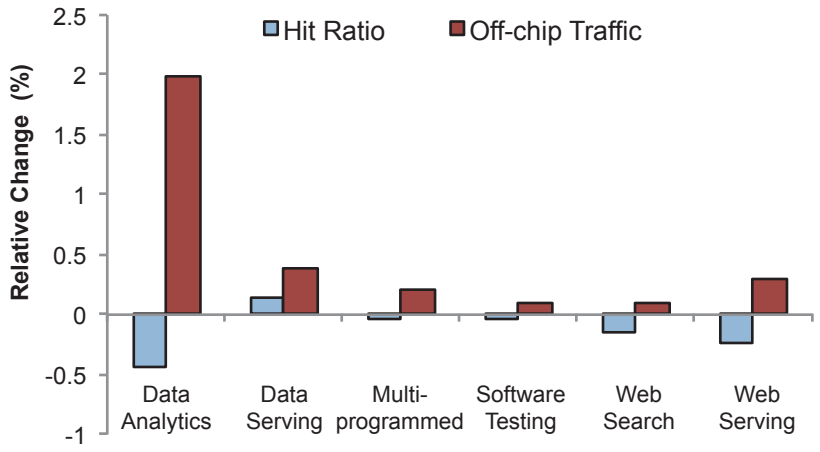

Figure 3.11 – Relative change in hit ratio and off-chip traffic for a 256MB Footprint Cache with 1KB pages using a tagless history table, normalized to the design with complete tags.

into a 14-bit hash. Because XOR is an associative operation, it is possible to compute an XOR hash of the PC in the pipeline and transfer only those 14 bits to the DRAM cache controller, which further applies the XOR operation using the offset bits. As a result, both the history size and the amount of PC information transferred to the DRAM cache controller are substantially reduced.

3.5.4 Other Processor Architectures

We evaluated Footprint Cache in the context of scale-out processors [50, 49]. However, our design is not limited to such an organization. In fact, any many-core chip design that stresses off-chip bandwidth (e.g., Tilera TILE100) would yield similar results. In contrast, processor designs with a handful of large cores (e.g., Intel Westmere) would see less benefit from die-stacked caches as they cannot utilize the available memory bandwidth due to their low degree of on-chip parallelism [12].

3.5.5 Cache Capacity

In this work we covered die-stacked DRAM caches ranging from 64-512MB per pod (up to 3GB per chip). However, the datasets of these workloads are scaled down from hundreds of gigabytes to tens of gigabytes to allow for practical full-system simulation. Because miss rates for server workloads follow a power law [21], which we verified for these workloads,

the observed miss rate curve will shift to the right for larger datasets. This means that the simulated cache sizes correspond to an order of magnitude larger caches in an industrial-strength setup. The all caches we evaluate, however, do have a capacity limitation to 512MB because of the SRAM storage required either for the tags or for various predictors. Scalable DRAM cache solutions with DRAM-based tags are the subject of the next chapter.

3.5.6 Footprint Cache in Non-3D Systems

We evaluated Footprint Cache in the context of die-stacked DRAM. However, nothing in this work is 3D-specific, and our design and conclusions remain valid for other forms of high-bandwidth low-latency on-chip DRAM, such as eDRAM [59] or systems integrated via silicon interposer [10].

3.6 Conclusion

In this chapter we presented Footprint Cache, a cache architecture that combines the best aspects of current die-stacked DRAM cache designs, which fall short of achieving the potential of the die-stacking technology. Footprint Cache fully exploits abundant spatial locality of scale-out applications observed in large DRAM caches, without introducing unnecessary off-chip and on-chip traffic. Footprint Cache is able to achieve the hit ratio of page-based designs and stay within the bandwidth requirements of the block-based ones, while fully preserving on-chip and off-chip DRAM locality. Furthermore, the small tag array overhead makes Footprint Cache practical for implementation. Using full-system, cycle-accurate simulation of scale-out server platforms, we demonstrated that Footprint Cache delivers 57% performance improvement on average, outperforming existing designs, while reducing off-chip DRAM dynamic energy by 78% compared to the baseline system and reducing stacked DRAM dynamic energy by 24% compared to state-of-the-art.

4 Scalable DRAM caches

What allows Footprint Cache to store its tags in SRAM is its page-based organization, which minimizes the storage required for the tags. However, as the technology rapidly enables multi-gigabyte stacked DRAM capacities, even page-based tags quickly consume too much SRAM to be practical. To illustrate, 8GB of stacked DRAM would need 16MB of SRAM in the best case, which is larger than today's last-level caches. Moreover, this storage drastically increases if the cache uses sub-blocking to optimize for off-chip bandwidth, as Footprint Cache does. Furthermore, while the stacked DRAM provides a huge increase in bandwidth compared to conventional DDR channels, the *latency* of the die-stacked DRAM is not substantially better. If a DRAM cache architecture requires accessing the stacked-DRAM or a multi-megabyte SRAM table for tag lookups, then that could add several tens of cycles to the overall cache latency, offsetting any latency advantage of stacked DRAM.

The downside of Footprint Cache is that, as discussed above, the SRAM-based tag array will not gracefully scale to larger stacked DRAM sizes and the tag array imposes additional latency to service a request. In this chapter we examine the approaches to scaling the Footprint Cache design (and page-based designs in general) to multiple gigabytes by efficiently storing its tags in DRAM, while preserving all of its benefits, including high hit rates, low off-chip traffic, and low cache-hit and cache-miss latencies. We build upon a recently proposed block-based Alloy Cache (AC) design [55], which provides an architecture that completely avoids any large SRAM-based tag arrays, and overall provides low latencies on cache hits. Alloy Cache is organized as direct-mapped to avoid searching for the correct way throughout the DRAM-based tags and co-locates each data block with its tags, reading it together with the data block in a single access. However, these advantages come at the cost of relatively low cache hit rates, which are further penalized by the cache's direct-mapped organization, and high miss penalty. To avoid

DRAM cache lookups on cache misses, Alloy Cache employs a miss predictor, sending cache requests to main memory speculatively, if a miss is predicted.

Unison Cache is carefully designed to combine the best traits of both Alloy Cache and Footprint Cache, while avoiding their shortcomings. Tags are directly embedded in the stacked DRAM, like Alloy Cache, to avoid SRAM-based tag arrays. At the same time, Footprint Cache-like large allocation units are used to exploit spatial locality, with the added benefit of reducing the fraction of the stacked DRAM's capacity that must be set aside for the embedded tags. To effectively realize such a design we leverage the following insights:

- In order reduce hit latency Alloy Cache merges ("alloys") each data block and its tag into a single unit and streams both in a single access. However, the primary latency benefit comes from breaking the serialization between the tag and data accesses. Unison Cache instead uses a single tag per page, but overlaps the tag read with the data block read. In doing so, Unison Cache achieves the same hit latency, but also allows for an effective page-based organization with DRAM-based tags.

- By leveraging spatial locality, Unison Cache achieves high hit ratios (often 90% or better). With such a high hit ratio, the miss predictor used by Alloy Cache to reduce miss penalty is not necessary, as a static "always-hit" prediction achieves similar accuracy.

- Direct-mapped organization hurts page-based designs, causing many more conflicts compared to block-based designs. However, we find that direct-mapped organization is not necessary to achieve low hit latency. To reduce the number of conflict misses Unison Cache is organized as a set-associative cache. Instead of serializing tag and data accesses or fetching all the ways in parallel, Unison Cache relies on simple and highly accurate way prediction, increasing neither the cache hit latency nor the amount of transferred data.

The end result is that by carefully leveraging the aforementioned insights, the proposed Unison Cache is able to outperform both Alloy Cache and Footprint Cache designs, approaching the performance of an ideal "latency-optimized" DRAM cache (100% hit rate, 0-cycle tag access). At the same time, Unison Cache does not require SRAM-based tag arrays, which allows Unison Cache to easily scale up to cache sizes of many gigabytes needed by server applications. A summary of the key features of Unison Cache, as well as the prior art, is listed in Table 4.1.

	AC	FC	UC
No SRAM tag overhead	✓	✗	✓
Low hit latency	✓	✗	✓
High hit rate	✗	✓	✓
High effective capacity	✗	✓	✓
Scalability	✓	✗	✓

Table 4.1 – Comparison of Alloy Cache (AC), Footprint Cache (FC), and Unison Cache (UC).

4.1 Block-Based Caches with DRAM-Based Tags

Block-based DRAM caches require several tens or hundreds of MBs for tags. Such a large volume of tag metadata rules out a conventional on-chip, SRAM-based tag array, and forces the tags to be placed in the stacked DRAM along with the data blocks [46, 47, 55]. However, storing tags directly in the DRAM cache can potentially require two DRAM accesses per cache lookup (one for the tag and another for data), thereby doubling the effective DRAM cache access latency in the worst case.

To improve the effective DRAM cache access latency, Loh and Hill proposed organizing each DRAM row as a cache set and co-locating all the ways of a set and their corresponding tags in the same DRAM row [47]. On a DRAM cache request, first, the tags in the beginning of a row are accessed for tag comparison. Upon a tag match, the request for the corresponding data block is issued separately, causing serialization of the tag lookup and data access. However, the accesses to tags and data are scheduled in a way that ensures a row buffer hit for the data block after the tag access, partially reducing the penalty for the second access.

Even though this scheduling optimization reduces the DRAM cache hit latency by exploiting row buffer locality, cache hits suffer from tag lookup and data fetch serialization, while cache misses suffer from high miss latencies due to the tag lookup in the DRAM cache prior to issuing the request to the off-chip main memory. To reduce the DRAM cache miss latency, Loh and Hill propose employing an on-chip SRAM “MissMap” to maintain cache block presence information in a compact form. This way, DRAM cache misses can bypass the high-latency lookups and an off-chip memory request can be issued directly. Unfortunately, this comes at the cost of further increasing the DRAM cache hit latency by adding the MissMap access to the cache lookup path, and the multi-MB MissMap itself will not scale up to support multi-GB DRAM caches.

The state-of-the-art block-based approach, Alloy Cache [55], organizes the DRAM cache as

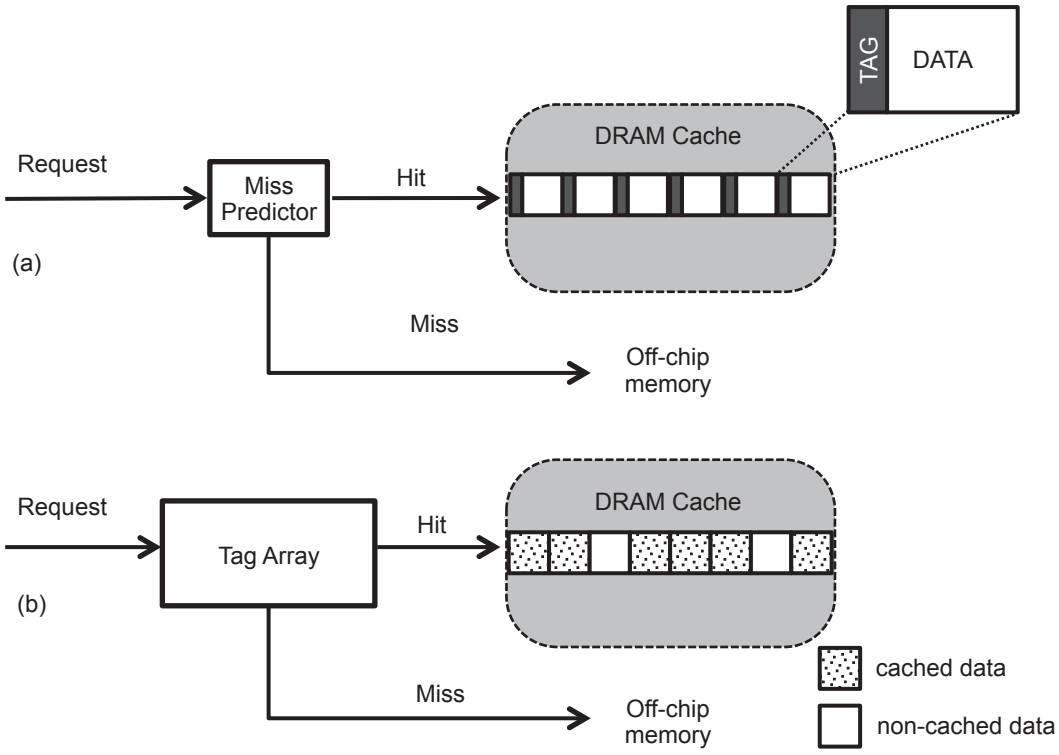

Figure 4.1 – Overview of the (a) Alloy Cache and (b) Footprint Cache designs.

direct-mapped, further reducing the already low hit rate, but compensating for this by greatly improving the cache access latency. AC merges (or "alloys") each single data block with the corresponding tag in unified tag-and-data units (TAD), as shown in Figure 4.1(a). The direct-mapped organization eliminates the need to search for the correct way in the DRAM, allowing AC to stream out a TAD in a single read, thereby breaking the tag-then-data serialization on cache hits and thus significantly reducing the lookup latency compared to Loh and Hill's design.

To minimize the DRAM cache miss latency, AC employs a simple low-latency miss predictor, moving the DRAM cache tag lookup off the critical path when the predictor correctly predicts misses. However, when a cache hit is predicted to be a miss, AC creates extra off-chip traffic by sending an unnecessary fetch request for a block that is already in the cache. When a cache miss is predicted to be a hit, the actual off-chip memory request is delayed by the tag lookup latency.

Alloy Cache is able to effectively mitigate tag-lookup latencies. However, it fails to provide sufficiently high hit rates for server workloads due to its block-based nature, and the lack of

associativity lowers the hit rates even further. The gap in hit rate between block-based and page-based could be bridged through prefetching. However, as we will see in Section 4.2.1, Alloy Cache's distributed tag architecture makes the implementation of spatial prefetchers impractical.

4.2 Unison Cache

The first key insight that leads to an effective design is that while Alloy Cache's tag-and-data (TAD) co-location provides the ability to stream both in a single DRAM access, the primary latency benefit of such an approach comes from breaking the serialization between tag and data accesses rather than from the tag-and-data co-location itself. Unison Cache physically separates tags and data blocks within the DRAM row and uses a single tag per page, as shown in Figure 4.2, but the read operations for both the tag and the individual data block can be overlapped as they are not dependent on each other. While this may require two separate back-to-back read commands to the same row, the reads are not serialized and therefore the latency ends up being the same as for reading a TAD. Maintaining a single centralized tag per page reduces the tag overhead and allows for an efficient implementation of spatial prefetchers, because the presence information for all blocks within a spatial region is kept at one place. For example, such a tag architecture makes the process of tracking page footprints easily implementable and efficient. A data block and the corresponding page tag are always read in parallel (i.e., the tags and data work "in unison").

The second observation is that by leveraging spatial locality, Unison Cache (like Footprint Cache) can achieve very high hit rates (often 90% or better). At this point, we can dispense with Alloy Cache's hit predictor, as a static "always-hit" prediction would achieve accuracy similar to a dynamic hit prediction.

Finally, to avoid the price of the direct-mapped organization, which is particularly high for page-based designs, Unison Cache is organized as set-associative, co-locating all the pages of a set in the same DRAM row. However, instead of serializing tag and data accesses or fetching all the ways at the same time, Unison Cache relies on highly accurate way prediction, increasing neither the cache hit latency nor the amount of transferred data.

In the rest of this section, we describe the Unison Cache design and its operation in detail.

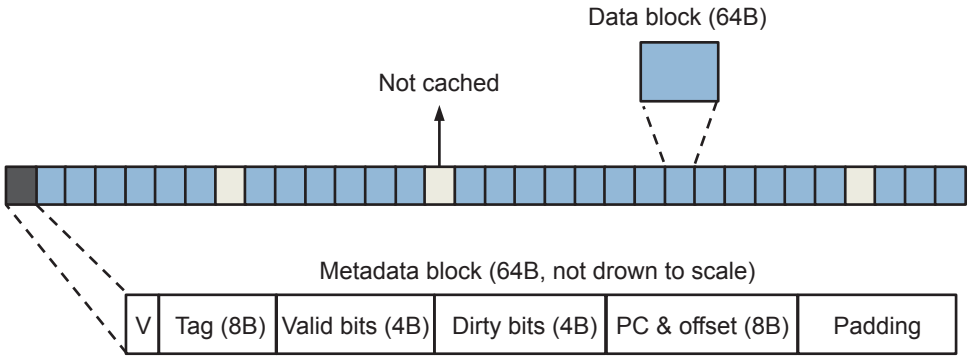

Figure 4.2 – DRAM row content in Unison Cache (not drawn to scale).

Footprint Prediction

Unison Cache learns and fetches page footprints to avoid off-chip bandwidth waste. The footprint of a page comprises all the blocks that are touched between the first access to the page, which happens upon an access to a page that is not in the cache, and the eviction of the page.

Our design leverages FC's footprint predictor [29]. The predictor relies on the correlation between the code and page footprints. This correlation stems from repeated calls to a limited set of functions to access large amounts of data, especially in well-structured object-oriented server software. Repetitive calls to these functions result in repetitive data access patterns (i.e., page footprints) that can be exploited to predict future data accesses upon subsequent calls to the same function. The correlation between code and data access patterns has been heavily exploited for data prefetching [6, 38, 62] and filtering of unused data [29, 37, 39, 70].

The instruction that accesses the first data block in a page has been shown to accurately predict footprints of pages that are later accessed by the same instruction [29, 62]. To account for different alignments of data structure instances in different memory pages, there is also a need to combine the instruction information (i.e., *PC*) with the distance of the first accessed block from the beginning of the page (i.e., *offset*) [29, 62]. Hence, the footprint predictor predicts page footprints based on the (PC, offset) pair that initiates the first access to a page, the *trigger access*. Each footprint prediction table entry consists of a (PC, offset) pair and a bit vector to indicate the page footprint correlated with that pair.

Learning Footprints

To facilitate footprint learning, each page in Unison Cache is augmented with a (PC, offset) pair that corresponds to the first access to the page (triggering miss), as in Footprint Cache. This information is inserted into a DRAM row along with the data when the page is allocated (Figure 4.2). During the page's residency in the cache, each access to a block within a page updates the corresponding valid/dirty bits in the bit vector that belongs to the page's tag to indicate that the block had been demanded (in Section 4.6 we will show how this overhead traffic can be virtually eliminated). To determine the footprint of a page it is necessary to make a distinction between fetched blocks that are actually demanded by the CPU at some point and those that are not (overfetched blocks). To enable such a distinction without extra storage, we modify the semantics of the existing valid and dirty bits and use a different block state encoding scheme, as in Footprint Cache. Upon eviction, the triggering (PC, offset) pair and the footprint bit vector (constructed based on valid and dirty bits) of the evicted page are read from the DRAM row and used to update the footprint prediction table, which associates a footprint to each (PC, offset) pair.

Fetching Footprints

When the requested page is not found in the cache, the footprint prediction table, stored in SRAM, is queried for the (PC, offset) pair that triggered the cache miss. If a match is found, the corresponding footprint is used to determine what blocks will be fetched. In the case of a miss to a block whose page is already allocated in the cache (i.e., footprint *underprediction*), there is no need to initiate footprint prediction and new page fetch. Instead, only a single fetch request for the missing block is sent to memory. However, when the page is evicted, the footprint of the page will indicate that the block was touched during the page's residency and the footprint prediction table is updated accordingly to avoid future underpredictions for the same (PC, offset) pair. Likewise, the footprint prediction might fetch blocks that are not touched during a page's residency in the DRAM cache (i.e., *overpredictions*). Similar to underpredictions, overpredictions are also propagated to the footprint prediction table when a page is evicted to avoid future overpredictions.

Singleton Prediction

In Chapter we showed that a significant fraction of page footprints consists of only a single block. Singleton pages reduce the effective DRAM cache capacity because they allocating

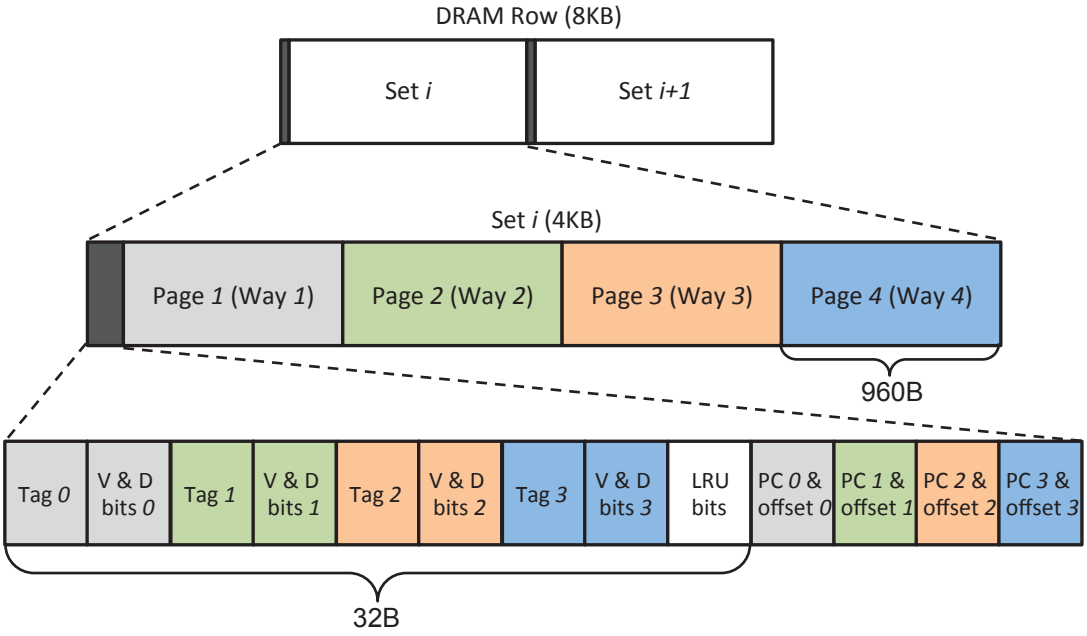

Figure 4.3 – DRAM row organization in the Unison Cache design.

space for an entire page, but accommodate only a single block. Hence, Unison Cache does not allocate a page in the cache if the footprint prediction table predicts the page to be a singleton. The missing block is fetched from memory and simply forwarded to the requestor. However, as singleton pages are not allocated in the cache, it is not possible to correct footprint mispredictions (corrections happen upon page evictions). To track the singleton pages that might become non-singleton later, Unison Cache employs a small singleton table in SRAM as in Footprint Cache [29].

Associativity

Alloy Cache uses direct-mapped organization to quickly locate the requested block in the cache if it is present, without searching through the DRAM tags to find the correct way. Unison Cache inherits the same mechanism to quickly locate the requested page. However, UC is page-based and direct-mapped page-based caches are highly vulnerable to cache conflicts. While zero associativity does not severely affect the hit ratio of block-based DRAM cache designs due to the large number of sets [55], it has a huge impact on page-based designs. We use a simple analytical model to explain this phenomenon.

Let n be the number of 64-byte blocks in a direct-mapped block-based cache, Cb. At the same

time, n represents the number of different sets in the cache. Let pb be the probability that two randomly chosen cache blocks belonging to the same set are in conflict during a particular window of time w. The conflict happens when both of the blocks are requested during the window of time w. We further assume that a direct-mapped page-based cache of the same size, Cp, is organized into pages, each page containing k 64-byte blocks. Under these assumptions, Cp contains n/k entries — i.e., n/k sets. We are interested in computing the probability that two randomly chosen blocks in cache Cp are in conflict during the window of time w. The important observation is that in block-based designs two randomly chosen blocks in Cp will be in conflict if and only if they belong to the same set and if they are both requested in during the window of time w. In contrast, in page-based designs two blocks belonging to the same set will be in conflict not only if the two blocks themselves are needed at the same time, but also if *any* two blocks from the pages they belong to are needed during the window of time w.[1] The probability of conflicts thus grows quadratically with the page size and creates a severe problem despite the large cache size. More precisely, the probability we are looking for can be expressed as:

$$pb \cdot \binom{k}{2} \approx 0.5 \cdot pb \cdot k^2$$

In case of 2KB pages (k=32), this probability would be 500 times higher compared to the same probability in the case of the block-based design Cb. While the number of sets decreases linearly with the page size, the probability of conflicts within each set grows quadratically. In other words, the three orders of magnitude gap in the number of sets between an L1 cache and a DRAM cache is easily bridged through the use of large allocation units, signaling that associativity is as important in page-based DRAM caches as it is in block-based L1 caches. For a 1GB cache and 2KB pages, the probability of conflicts increases by a factor of ~500 in the worst case compared to a block-based direct-mapped cache of the same size.

To reduce page conflicts and achieve higher hit rates, we organize Unison Cache as a set-associative page-based cache. We do not, however, go back to tags-then-data serialization, as it would be highly inefficient; nor do we fetch several ways in parallel, as it would create vast data overfetch and eventually lead to serialization of the fetched ways on the bus, significantly increasing the latency [55]. Instead, we use a simple way predictor that yields an accuracy of over 95% and use this information to fetch the correct way from a DRAM row. We describe the details below.

[1] This is analogous to the false sharing problem.

DRAM Row Organization and Operations

So far we assumed, for simplicity, that the size of a cache page equals to the DRAM row size. In reality, DRAM rows are typically larger than the desirable page size. For the sake of generality, let's assume that the cache is four-way associative, the page size is 1KB, and the DRAM row size is 8KB. In this example, each set is 4KB, and one DRAM row accommodates two whole sets, as shown in Figure 4.3. One of the two sets (half of a DRAM row) is shown in more detail with its four pages. The metadata of each page (valid bit, page tag, valid and dirty bit vectors, replacement policy bits, and (PC, offset) is maintained in the beginning of the row, such that the metadata required to determine the presence of a block is stored first (page tags and bit vectors), whereas (PC, offset) pairs and other metadata for all pages are stored after all the tag information. This placement is chosen for efficiency reasons, so that all the tags from a set can be read together in a single access. For the assumed configuration, the total size of the tag metadata for the four pages is 32B, which can be transferred in two bursts over a 128-bit TSV bus, corresponding to one bus cycle or two CPU cycles in the system we evaluate.[2] The metadata read command is immediately followed by the read command for the data block whose position in the DRAM row is determined by the page offset and by the predicted way; the two read operations are overlapped.

The two cycles that represent an overhead to read the tags leave enough room for way prediction, which is done by the DRAM controller and is not on the critical path. We use a simple tagless way predictor, which is a 2-bit array directly indexed by the 12-bit XOR hash of the page address (16-bit XOR for caches above 4GB). Prior work on way prediction has found that address-based way predictors are the most accurate way predictors for L1 caches [4, 54]. However, such predictors are not an option for L1 caches because the actual address is not known at the time when the prediction has to be made for L1 blocks. We do not have such a constraint here. While the accuracy of address-based way predictors is found to be around 85% for individual blocks [4, 54], our way predictor achieves much higher accuracy (~95%), because it operates at the page level. The abundant spatial locality leads to repeated accesses to the same page; subsequent accesses to the same page result in correct predictions. The predictor's page-based operation also reduces its storage overhead to 1KB (16KB for caches above 4GB). Because all the ways of a set reside in the same DRAM row, way mispredictions, apart from being rare, are also relatively cheap. Due to the DRAM row organization shown in Figure 4.3, the correct way in case of mispredictions is likely to be found in the row buffer, thus the uncommon case is not severely penalized.

[2] For systems with more than 1TB of memory (more than 40 physical address bits), three bursts would be needed to transfer ~48B of tags.

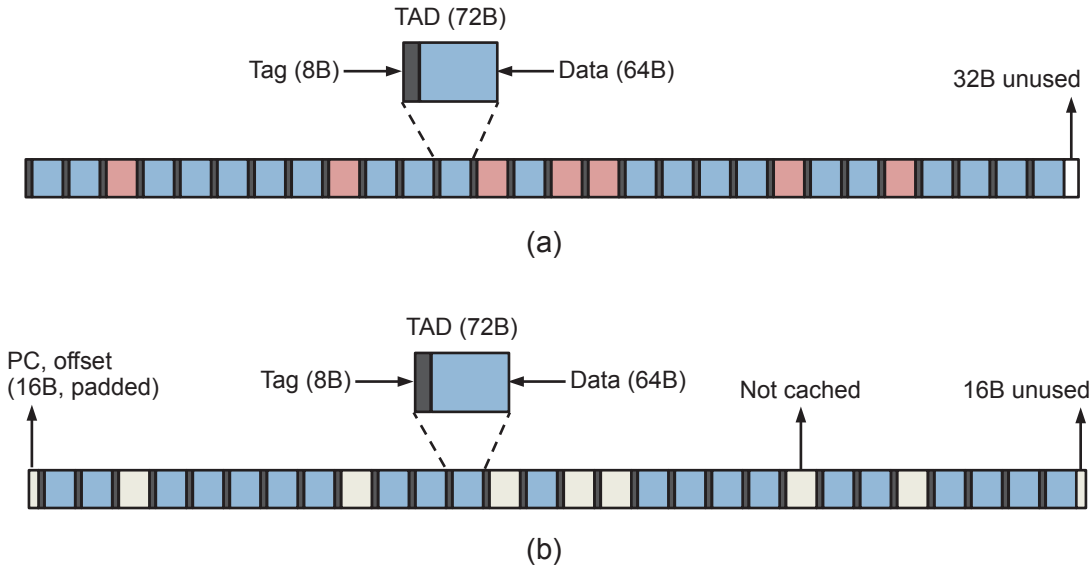

Figure 4.4 – DRAM row organizations for (a) block-based cache with footprint prediction, and (b) page-based cache with tagged blocks.

The (PC, offset) information is stored in the DRAM row upon the page's allocation and it is read only upon its evictions. This information is then used to update an SRAM-based footprint prediction table with the actual footprint of the evicted page, constructed from the page's bit vectors.

In case of cache misses, it is easy to distinguish between triggering misses (the requested page is not in the cache) or regular misses resulting from incorrect footprint prediction (i.e., underprediction), because the page tags for all the ways and the block presence bit vectors are stored in one place. The (PC, offset) information is also stored in the DRAM row upon page allocation and it is read only upon its evictions. This information is then used to update an SRAM-based footprint prediction table along with the actual footprint of the evicted page.

Address mapping

Integrating any kind of metadata into DRAM causes alignment problems, because a fraction of each DRAM row must be reserved for the metadata. In the case of Unison Cache, embedding the tag array into DRAM results in the page size being a non-power-of-two number (e.g., the pages sizes are 960B or 1984B, containing 15 or 31 64-byte blocks, respectively). Such page sizes require specialized logic for address manipulation instead of simply relying on address bits. Designing a general-purpose modulo-computing unit for such address manipulation

would incur high area and latency overheads. However, here we compute modulo with respect to a constant in a specific form (2^n-1), which can be computed with several adders using residue arithmetic [55]. We estimate the calculation to take two cycles and only a few hundred gates, as in AC, and it can be overlapped with last-level SRAM cache accesses.

Non-power-of-two can page sizes, however, cause problems for applications that align data in large power-of-two chunks. For certain applications misalignments can even double the miss ratio. We investigate the consequences of such misalignments in more details and propose an effective solution to integrate metadata into the stacked DRAM without misalignments in section 4.7.2.

	Alloy Cache	**Footprint Cache**	**Unison Cache**
Cache Miss Rate	Medium-High	**Low**	**Low**
Hit Latency	**Predictor + DRAM TAD Read**	SRAM Tag + DRAM Data Read	**Overlapped DRAM Tag/Data Reads**
Miss Latency	**Predictor Lookup**	SRAM Tag Lookup	DRAM Tag Lookup
Associativity	Direct-mapped	**32-way**	4-way (two pages)
64B Blocks per 8KB Row	112	**128**	120-124
SRAM Tag Array @ 8GB	—	~48MB	—
In-DRAM Tag Size @ 8GB	1GB (12.5% of DRAM)	—	256-512MB (3.1-6.2% of DRAM)
Miss-Predictor Size	96B per core, 1.5KB total	—	—
Way Predictor	—	—	1-16KB
Footprint History Table	—	144KB	144KB
Singleton Table	—	3KB	3KB

Table 4.2 – Comparison of key characteristics of different DRAM cache schemes.

4.2.1 Alternative Approaches

In this section we discuss alternative approaches to getting the best of block- and page-based designs. Looking at the two ends of the spectrum, there are two seemingly obvious ways to combine the two designs.

Block-based cache with footprint prediction

One naïve way of combining the two state-of-the-art block- and page-based designs is to start with Alloy Cache's direct-mapped, block-based organization with the tags co-located with data blocks, and then apply footprint prediction as a prefetcher in attempt to exploit spatial locality. Since the footprint prediction mechanism learns and predicts the blocks within pages, such a design would require grouping a number of neighboring blocks into a logical page and fetching and evicting them at the same time. Unlike existing page-based DRAM cache proposals, such a design could theoretically allow multiple pages to co-exist in

the same DRAM row as depicted in Figure 4.4(a). Unfortunately, multiple pages (shown as different shades of gray in the figure) could only co-exist in the same row if their footprints are completely disjoint; an overlap would cause a conflict and require the other page (i.e., its current footprint) to be prematurely evicted, as allocations and evictions happen at the page granularity.

Such a design would introduce major problems due to the mismatch between the cache organization and the footprint prediction mechanism. First, there is no fast lookup mechanism to indicate the presence of a page in the cache. In case of a miss, it is not possible to easily determine whether other blocks of the same page are cached or not. Thus, to identify if a cache miss is a triggering miss (the first miss to a page that initiates footprint prediction and fetching the page's footprint from off-chip memory), the entire DRAM row of the missing cache block needs to be scanned to determine if any block from the same logical page is present in the cache, because the block presence information is spread out over the entire DRAM row. Not finding any block within the page would indicate that the current miss is a trigger access. Such a scan is also needed to identify the footprint of the page that will be evicted as a result of the miss, and update the footprint predictor state accordingly. Unfortunately, scanning all tags in a DRAM row upon each cache miss and block eviction would significantly reduce DRAM cache availability, waste energy, and increase miss latency. Also note that for each page in the cache, we must keep its (PC, offset) pair that caused the initial miss, which are used to update the footprint predictor state upon eviction as in FC [29]. It is not straightforward to augment each DRAM row with the metadata corresponding to each of the variable number of logical pages it contains.

Page-based cache with tagged blocks

Another naïve way of combining the two designs is to start with FC and preserve its page organization, but augment each block in DRAM with its tag in order to stream tag and data blocks together in a single DRAM access, as in Alloy Cache. A DRAM row in such an organization is shown in Figure 4.4(b). As each DRAM row now accommodates a single page, upon a DRAM cache miss it is possible to determine whether or not the miss is the first access to the page that initiates the missing page's footprint fetch. However, this requires writing the correct page tag and resetting the valid bit even for blocks that are not fetched upon page insertions, which means an extra DRAM write for each block that does not belong to the footprint of a newly fetched page. Furthermore, upon page evictions following a miss, there is no simple lookup mechanism to identify the footprint of the evicted page; the entire DRAM row would need to

be scanned to determine the valid blocks within the page. In contrast to the previous design point, the (PC, offset) pair that triggered a page access could be stored at a predetermined position in the corresponding DRAM row and later used to update the footprint prediction table with the correct footprint.

In both naïve design points each data block is co-located with its corresponding tag to minimize latency, leading to a vast amount of replication. The tag replication wastes around 1/8th of the total cache capacity and further reduces the hit ratio. Furthermore, the footprint predictor is partially integrated into DRAM-based tags, which contain various metadata needed for prediction, most importantly the block presence information. Spreading this information throughout a DRAM row causes, as discussed, a variety of problems related to footprint tracking, detecting triggering misses, page evictions, and unnecessary DRAM row scans and writes. Unison Cache avoids these problems by centralizing the tag information for all data blocks within a page and accessing this information in parallel with data blocks to avoid any latency penalty.

4.2.2 Summary and Comparisons

Unison Cache leverages insights and ideas from both the Alloy Cache and the Footprint Cache, but synthesizes and extends them in unique ways to "get the best of both worlds" while sidestepping their pitfalls. Given the many interacting and inter-dependent components, Table 4.2 provides a summary of the key characteristics of the different DRAM cache design approaches to more easily distinguish the contributions and strengths of Unison Cache.

Unison Cache maintains the low miss rate of Footprint Cache (FC), the low hit latency of Alloy Cache (AC), avoids the impractically large SRAM tag arrays of FC, has lower embedded DRAM tag overheads than AC, and has no miss predictor like AC. Assuming an 8GB die-stacked DRAM and 2KB pages, FC would require about 50MB for its SRAM tag array.

On a cache miss, AC has the best latency (assuming the hit-predictor was correct), but in practice both FC and Unison Cache have sufficiently high hit rates that the additional tag-lookup latency for misses has a much smaller impact. FC and Unison Cache often have hit rates in excess of 90%, which is functionally equivalent to having a static hit-predictor with a 90% accuracy.

FC has by far the highest associativity. However, the additional associativity beyond four ways provides rapidly diminishing returns, as discussed in Section 4.4. This is why Unison Cache's comparatively lower four-way set associativity is not a significant constraint.

Like FC, Unison Cache requires some on-chip SRAM resources to implement the footprint predictor structures, but these are fixed sizes and *do not* grow with increasing stacked DRAM capacities.

4.3 Methodology

4.3.1 Simulation Infrastructure

We evaluate Unison Cache through a combination of trace-driven and cycle-level simulation of a 16-core CMP running server workloads. We use the Flexus [68] full-system multiprocessor simulator, which extends the Virtutech Simics functional simulator with OoO cores, on-chip network, and memory hierarchy and models the SPARC v9 ISA. We use DRAMSim2 [58] integrated into Flexus to model both the die-stacked DRAM and the off-chip DRAM, with the parameters listed in Table 4.3.

The trace-driven experiments are based on the memory traces that consist of 30 billion instructions per core, two thirds of which are used for cache warm-up. We evaluate performance through a set of cycle-level experiments, leveraging the SimFlex [68, 69] multiprocessor sampling methodology for server workloads. Our samples are collected over 15 seconds of workload execution. For each measurement point, the cycle-level simulation starts from checkpoints with warmed up architectural state (i.e., caches and branch predictors) and runs for 800K cycles (2M for Data Serving) to warm up the queues and the interconnect state. Then, we collect measurements for the subsequent 400K cycles of the cycle-level simulation. To measure performance, we use the ratio of the number of user instructions to the total number of cycles (including the cycles spent executing the operating system code), as this metric has been shown to accurately reflect overall server throughput [68]. Performance measurements are computed with an average error of less than 2% at a 95% confidence level.

4.3.2 Baseline System Configuration

Our baseline processor is a 16-core CMP design based on the Scale-Out Processor design methodology [50], which seeks to maximize throughput per die area. The chip features a modestly sized last-level cache to capture the instruction working set and shared OS data, which are independent of the core count, and dedicates the rest of the die-area to the cores to maximize throughput. The architectural features are listed in Table 4.3.

CMP Organization	16-core Scale-Out Processor pod
Core	ARM Cortex-A15-like, 3-way OoO @3GHz
L1-I/D caches	64KB, split, 64B blocks 2-cycle load-to-use latency
L2 cache per pod	4MB, unified, 16-way, 64B blocks, 4 banks, 13-cycle hit latency
Interconnect	16x4 crossbar
Off-chip DRAM	16-32GB, one DDR3-1600 (800MHz) channel 8 banks per rank, 8KB row buffer
Stacked DRAM	DDR-like interface (1.6GHz) 4 channels, 8 banks/rank, 8KB row buffer, 128-bit bus width
t_{CAS}-t_{RCD}-t_{RP}-t_{RAS} t_{RC}-t_{WR}-t_{WTR}-t_{RTP} t_{RRD}-t_{FAW}	11-11-11-28 39-12-6-6 5-24

Table 4.3 – Architectural system parameters.

Cache size (B)	128M	256M	512M	1G	2G	4G	8G
Tags (MB)	0.8	1.58	3.12	6.2	12.5	25	50
Latency (cycles)	6	9	11	16	25	36	48

Table 4.4 – Footprint Cache parameters.

4.3.3 DRAM Cache Organizations

Unison Cache

The evaluated design is organized as a four-way set associative cache. Each DRAM row accommodates two sets, each of which contains four pages. Each page contains 15 blocks (960B), and the whole DRAM row accommodates 120 data blocks. We also evaluate a direct-mapped organization of Unison Cache as well as organizations with 1984B pages. The parameters for footprint prediction are the same as in Chapter 4.2.

Footprint Cache

We evaluate the original design with 2KB pages, which we found to be the sweet spot between the accuracy and tag storage overhead. The 8KB DRAM row can accommodate four pages with 128 data blocks. While 1KB pages are a better match for Unison Cache, Footprint Cache

cannot afford that page size as the already high SRAM-based tag storage would double. The aggregate size of the tag storage for various cache sizes is listed in Table 4.4 along with the conservatively estimated latencies. Note that for larger cache sizes Footprint Cache's tag array grows up to ~50MB, which cannot even fit alone in the area of today's chips, but we evaluate these hypothetical designs as reference points.

Alloy Cache

The 8KB row buffer is able to accommodate 112 data blocks. Alloy Cache also employs a miss predictor with a one-cycle latency to bypass the DRAM cache lookup in case of a DRAM cache miss.

4.3.4 Workloads

As a representative set of emerging scale-out server applications that are highly data-intensive and exhibit abundant request-level parallelism, we use the CloudSuite [9] workloads, including Data Analytics, Data Serving, Software Testing, Web Search, and Web Serving [12]. To evaluate multi-gigabyte cache designs, we use a set of analytic queries from the industrial TPC-H benchmark (referred to as TPC-H), running on a modern column-store database engine, MonetDB [33]. While the datasets of other workloads are scaled from hundreds of gigabytes down to 5-20GB (depending on the workload) to allow for practical full-system simulation, the TPC-H dataset is unchanged and exceeds 100GB.

4.4 Evaluation

4.4.1 Predictor Accuracy

The three designs we evaluate rely on various predictors to predict if an access is a hit or miss, to predict page footprints, or to predict the correct way in a set-associative cache. Table 4.5 summarizes the effectiveness of these predictors as well as the extra off-chip traffic generated by some of the predictors due to mispredictions, assuming a 1GB cache (8GB for TPC-H queries). We observed similar trends for other cache sizes, for which we omit the results. For Unison Cache (UC), we show two design points: with 960B and 1984B pages, both 4-way associative. For Alloy Cache (AC), we show the accuracy of the miss predictor — the fraction of misses correctly identified as such. Misses that are wrongly predicted as hits increase miss latency. AC's miss predictor is highly effective achieving over 90% accuracy on our server

		Data Analytics	Data Serving	Software Testing	Web Search	Web Serving	TPC-H Queries	Average Value
AC	MP Accuracy (%)	96.4	90.0	93.2	97.2	91.8	89.0	92.3
	MP Overfetch (%)	7.3	6.4	16.2	13.5	7.9	1.9	8.7
FC	FP Accuracy (%)	92.4	97.7	81.5	98.6	92.3	93.8	92.7
	FP Overfetch (%)	9.2	4.0	24.7	1.6	9.0	6.18	9.1
UC 960B	FP Accuracy (%)	93.1	97.1	84.2	95.5	89.8	84.0	90.6
	FP Overfetch (%)	9.0	3.7	20.6	3.2	12.8	10.7	10
	WP Accuracy (%)	89.6	90.6	92.4	96.6	94.6	95.9	93.3
UC 1984B	FP Accuracy (%)	90.2	95.7	78.2	94.4	83.4	79.9	87.0
	FP Overfetch (%)	11.5	5.4	26.8	4.4	18.9	15.4	13.0
	WP Accuracy (%)	91.1	93.9	96.2	98.1	96.9	96.8	95.5

Table 4.5 – Accuracy of various predictors: Miss Predictor (MP) in Alloy Cache, and Footprint Predictor (FP) in Footprint Cache and Unison Cache, and Way Predictor (WP) in Unison Cache for a 1GB cache (8GB for TPC-H queries).

workloads. The hits that are wrongly identified as misses and thus cause unnecessary off-chip traffic are also shown and are not significant.

For Footprint Cache (FC) and UC, we show the footprint predictor's accuracy — the fraction of a page's footprint that is correctly predicted. We note that this metric is not comparable to AC's accuracy metric. The difference in accuracy for FC and UC stems from the differences in associativity and page size. For most of the workloads, UC's accuracy matches the accuracy of FC. We also note that the UC organization with 960B pages on average provides better prediction accuracy compared to the 1984B organization, which is what we also concluded in the FC study in Chapter 4.2. While FC cannot afford this granularity because of its SRAM-based tag array, UC keeps tags in DRAM and is not restricted to large page sizes.

We also show the overfetch ratios of the two predictors to determine the extra off-chip traffic they generate. AC's miss predictor causes overfetch when it incorrectly predicts a DRAM cache hit to be a miss. Footprint predictor causes overfetch when it fetches blocks that are not accessed prior to a page's eviction. It is important to note that all three designs are highly bandwidth-efficient with small overfetch rates (~10% on average), which are offset by the benefits their predictors provide.

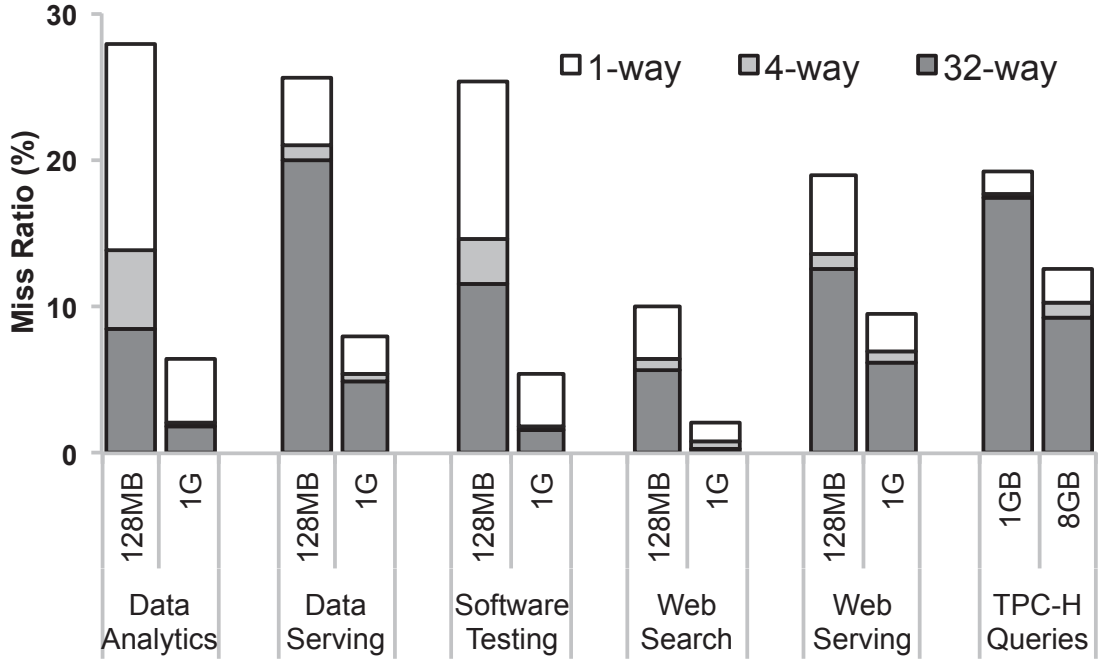

Figure 4.5 – Unison Cache's miss ratio as a function of associativity.

4.4.2 Miss Ratio

As explained in Section 4.2, UC increases the associativity to four by adding only two CPU cycles to the hit latency, which is negligible compared to the ~60 cycles it takes to access DRAM, and without causing data overfetch. Figure 4.5 shows the miss ratio for the UC organization with 960B pages while varying the cache associativity, for both large and small cache sizes. The miss ratios are plotted in a stacked fashion. For example, the dark gray bars show the miss ratios for a 32-way cache, while the sum of dark and light grey bars shows the miss ratios for the 4-way organization. The total height corresponds to the direct-mapped organization. We see that the four-way organization provides a sizable reduction in miss ratio, sometimes by a factor larger than two compared to the direct-mapped organization (the reduction is captured by the white bar). We note that beyond four ways, there is no significant reduction in the hit ratio to compensate for the increased tag lookup latency and reduced accuracy of the way predictor.

Way prediction and associativity have orthogonal effect. While reasonably small associativity halves the miss ratio (Figure 4.5), way prediction enables an effective implementation of associativity by eliminating the latency and bandwidth overheads. In our case, for a 4-way associative cache, way prediction reduces the latency by 12 cycles (needed to transfer extra

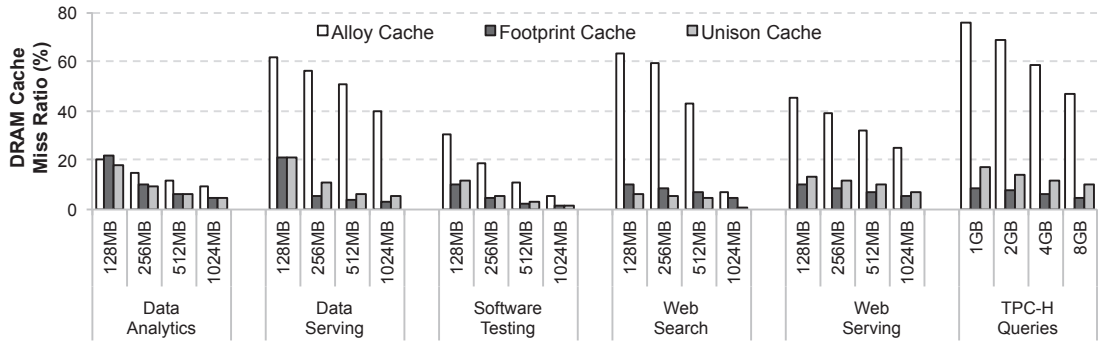

Figure 4.6 – Miss ratio comparison of Alloy Cache, Footprint Cache, and Unison Cache.

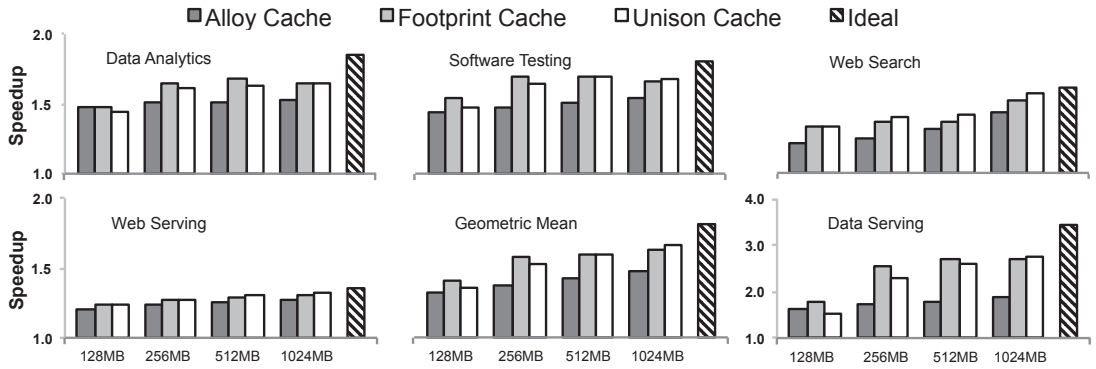

Figure 4.7 – Performance comparison of Alloy, Footprint, and Unison Caches. Note the difference in scale for Data Serving.

ways, 20% of hit latency) and reduces the hit traffic by 4x, as all the ways would otherwise have to be fetched in parallel.

We further compare the three designs with respect to their miss ratios in Figure 4.6 for a range of DRAM cache sizes. As expected, AC has by far the highest miss ratio due to low temporal locality. The exception is Data Analytics, a Map-Reduce workload that exhibits the lowest spatial locality due to its pointer-intensive nature caused by frequent hash table lookups. For this workload, the differences in miss ratio between the designs are less pronounced.

FC and UC, on the other hand, significantly reduce the cache miss ratio by exploiting spatial locality and fetching whole page footprints. The small differences between the miss ratios of FC and UC stem from different page sizes used in the two designs (2KB and 1KB, respectively), the difference in associativity, and a slight difference in the effective cache capacity. Because of the larger page size, FC provides slightly better miss ratios for applications with extremely high spatial locality, such as Web Search. In the case of Data Analytics, UC achieves a better miss

ratio due to the higher footprint prediction accuracy and low spatial locality of this workload, which prefers smaller page sizes.

Because AC is a block-based design, all the cache hits come solely from the temporal reuse. In other words, the hit ratio directly corresponds to the bandwidth savings provided by the cache. It is interesting to note that AC's miss ratio for TPC-H is consistently high, dropping down only for very large cache sizes; caches smaller than 2-4GB hardly provide any hits. This is in line with our intuition that multi-gigabyte caches are indeed required to provide a noticeable reduction in the off-chip traffic for realistic server setups.

4.4.3 Performance

Figure 4.7 compares the performance of the three designs for a range of DRAM cache sizes for all workloads except TPC-H. We also compare the three designs against an ideal DRAM cache that never misses and has no tag overheads, an equivalent to die-stacked main memory.

For small cache sizes, FC performs the best. Compared to AC, it enjoys a much higher hit ratio. The exception is Data Analytics (Map-Reduce), which for the smallest cache size prefers block-based designs due to the lack of spatial locality. As we increase the cache size, the pages stay longer in the cache and their footprints become denser [29], increasing the spatial locality. However, FC's tag array access latency increases with the cache size, increasing both the hit and miss latency and ultimately resulting in diminishing performance returns despite higher hit ratios. In contrast, the cache size affects neither the hit nor the miss latency in case of UC and AC, which is why UC outperforms FC for larger cache sizes.

A more realistic scenario is shown in Figure 4.8, which compares the performance of the three designs for TPC-H queries, for 1-8GB caches. In this case Unison Cache constantly outperforms the hypothetical Footprint Cache design due to its low and constant access latency, whereas the tag array access latency precludes performance improvements for Footprint Cache. Alloy Cache sees steady performance improvements, which are however limited by its low hit ratio.

Overall, Unison Cache provides a 14% performance improvement over Alloy Cache and 2% over the hypothetical Footprint Cache design for a 1GB cache (7% and 6% in case of an 8GB cache for TPC-H queries). We note once again that beyond 256-512MB, Footprint Cache is not a feasible option due to its SRAM-based tag array, which requires up to 50MB for an 8GB design.

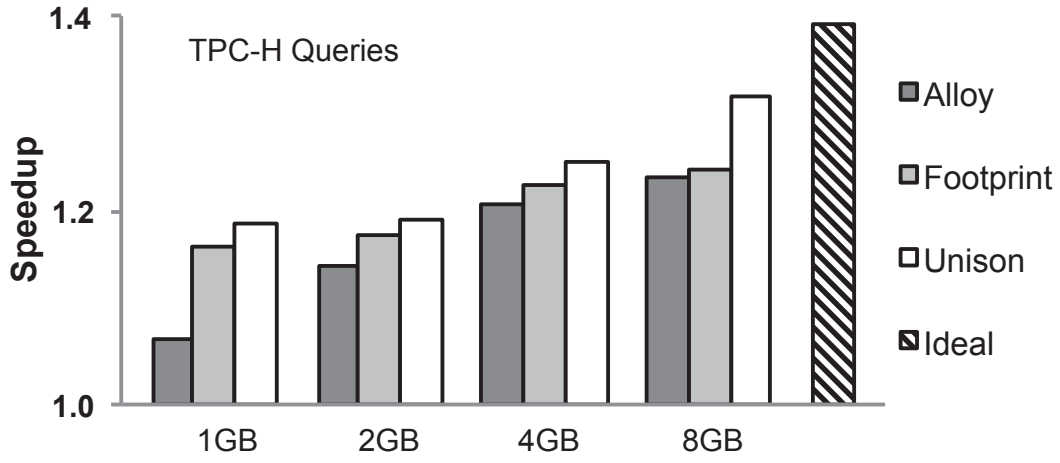

Figure 4.8 – Performance comparison for TPC-H queries.

4.4.4 Energy Considerations

All designs reduce the off-chip main memory energy by reducing the number of accesses to it. However, both UC and FC provide a significant further reduction in energy by reducing the number of DRAM row activations, the most energy-demanding operations, by an order of magnitude [29, 65]. Namely, while cache misses in the case of AC result in random memory accesses, both UC and FC perform off-chip data transfers at the granularity of footprints, which fit in a DRAM row. In case of AC, for almost every block transferred between the cache and memory, a DRAM row needs to be activated both in off-chip DRAM and in the cache, whereas for UC a row activation happens once for the whole footprint (i.e., once per ~10 blocks). Similarly, the DRAM cache energy is reduced due to the cache evictions and fills that happen at the footprint granularity. Data transfers between the die-stacked and off-chip DRAM are, thus, much more energy-efficient in the case of UC and FC. We quantified these benefits in Chapter 4.2, which are around 20-25% of dynamic DRAM energy and to the first order are the same for FC and UC.

4.5 Tag Cache

Besides accurate way prediction, high spatial locality also allows for effective caching of tags in any page-based cache design. The tag array provides two important pieces of information: data presence and data location. Regarding data presence, Unison Cache statically decides

that every access is a hit based on the premise that misses are rare, and therefore does not require data presence information. Regarding data location, Unison Cache employs a simple and accurate way predictor and does not rely on data location information from the tags either. Tag caching is therefore not essential for Unison Cache. Nevertheless, tag caching can improve performance in several ways, by:

- significantly reducing the number of cache probe traffic in case of dirty evictions for non-inclusive page-based DRAM caches. Unison Cache uses cache bypassing for singleton pages, and therefore is a non-inclusive cache; as such, upon dirty evictions it first needs to check the tags if a block is there and if so, where it is exactly, because write-backs cannot be done speculatively using presence prediction, be it static or dynamic (with miss predictors), and way prediction. Accurate tag caching can significantly reduce the number of dirty eviction probes by providing the exact tag information. While way prediction *always* provides *approximate* information, tag caching *frequently* provides *exact information*, which is what eviction probes require.

- reducing the number of cache probes in case of cache misses. Cache misses, although rare, unnecessarily probe the cache and increase the overall cache traffic. Tag cache hits eliminate such probes.

- reducing miss penalty upon tag cache hits. Unison Cache's static speculation that every access is a hit postpones off-chip miss serving by one cache access. Hits in a tag cache eliminate this increase in miss penalty.

- reducing the number of cache updates due to the replacement policy and prediction metadata maintenance upon cache hits. While the vast majority of cache hits are MRU accesses that do not require updates to the LRU bits[3], the first cache hits to any prefetched cache block must be registered in the valid/dirty bits to indicate that the block was demanded, which is required for footprint prediction. Hits in the tag cache essentially coalesce all these updates and write them back at once during tag cache eviction.

Figure 4.9 shows the tag cache hit ratio as a function of the number of tag entries, where each tag entry occupies around 40B of storage. Tag cache hit ratio is independent of the cache capacity. To be able to coalesce LRU updates, each tag cache entry keeps the metadata for one cache set consisting of four pages. This metadata includes four page tags, one for each

[3]Accesses to cache ways that are at the most-recently used position in the recency list do not change the replacement bits.

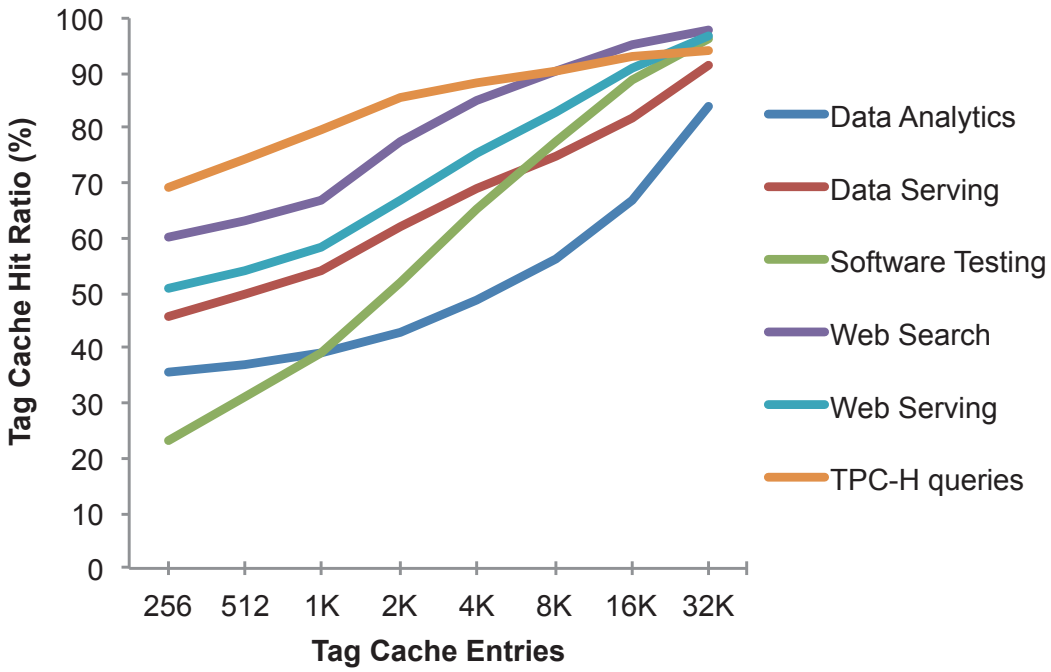

Figure 4.9 – Tag cache hit ratio as a function of the number of tag cache entries. Each entry corresponds to one cache set (four 1KB pages) and requires 40B of storage.

way, four valid/dirty bit vector pairs, and the LRU information for the whole set. As Figure 4.9 shows, the tag cache requires around 1280KB of storage to achieve the accuracy of a way predictor that has only 1KB of storage. The reason is that the tagless way predictor that Unison Cache employs uses only two bits per entry to indicate the predicted cache way, whereas the tag cache stores the complete tag metadata for the entire set. Moreover, accessing 1280KB of SRAM storage requires many CPU cycles, whereas way prediction can be done in a single cycle and off the critical path. Nevertheless, smaller tag caches could still be useful in filtering out many cache accesses and should be used with way prediction, which can facilitate locating the correct way in the case tag cache misses.

Our DRAM cache contains four tiles that are set-interleaved. Tag cache can be easily tiled in the same manner, reducing the size and the access latency of each tile. Assuming a tag cache tile of 80KB that could be probed in three CPU cycles, an 8K-entry tag cache occupying 320KB in total could be practical. Figure 4.10 shows the reduction in cache probes on dirty evictions that an 8K-entry tag cache achieves across our workloads, regardless of the cache capacity, and the reduction in cache probes on cache misses such a tag cache provides for a 256MB cache (2GB for TPC-H queries), as well as the reduction in metadata updates. We see that tag caching can significantly reduce the cache activity and also lower the cache miss

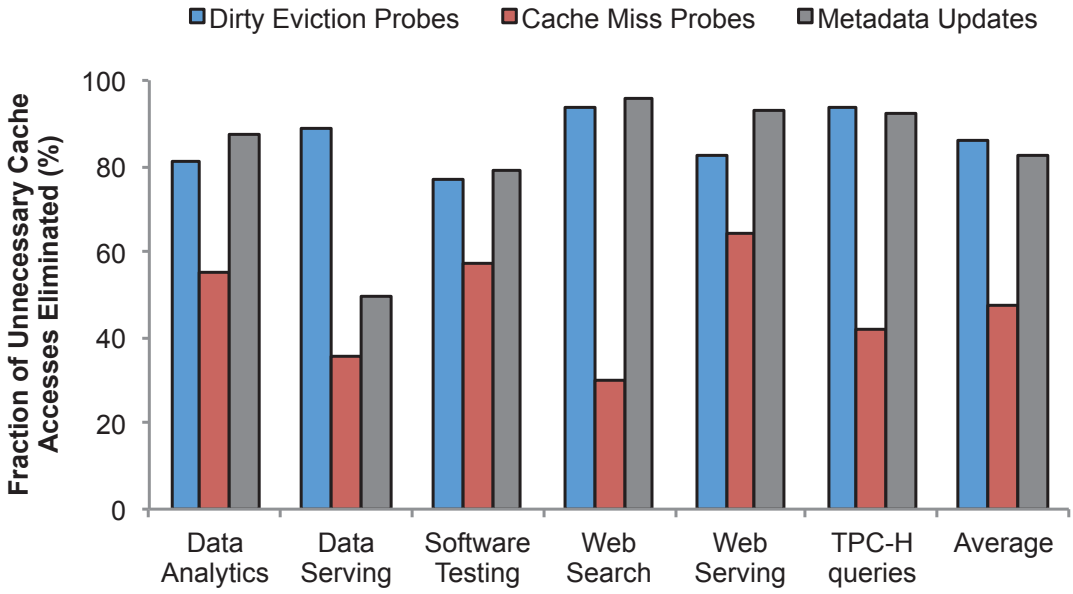

Figure 4.10 – Cache accesses eliminated by a practical tag cache with a three-cycle access latency. The figure shows the fraction of unnecessary probes upon dirty evictions and cache misses that are eliminated, as well as the fraction of LRU/metadata updates upon cache hits that can be coalesced. The cache capacity is 256MB (2GB for TPC-H queries).

penalty, at a three-cycle latency cost paid upon all cache accesses. On average, practical tag caching can eliminate around 85% of unnecessary cache probes upon dirty eviction requests, something less than a half of unnecessary cache probes upon cache misses, and more than 80% of metadata updates. Metadata updates happen on cache hits that are either non-MRU accesses and therefore require updates to the LRU bits, or had not been accessed before and the information that they were demanded have to be registered for footprint prediction. Figure 4.10 shows the fraction of those accesses that can be coalesced by a tag cache. The performance impact of the slightly lower miss penalty is not significant, having in mind that all cache hits are prolonged by three cycles. However, the impact of avoiding unnecessary cache accesses can be significant depending on the cache bandwidth utilization. Besides performance gains, tag caching has obvious implications on energy saving in die-stacked DRAM.

4.5.1 Tag Cache vs. Miss Predictor

In the context of Unison Cache, tag caches could be considered as miss predictors. Because Unison Cache uses way prediction to locate data, the only important information tag caches

provide is a more accurate estimate of data presence in the cache, as compared to Unison Cache's static policy. The important difference between miss predictors and tag caches is that tag caches provide the *exact presence information* upon tag cache hits, which prefetchers can relied upon. Namely, if Unison Cache had an approximate miss predictor, a cache hit that is wrongly predicted as a miss would initiate fetching of the entire page's footprint only to realize that the access was in fact a hit. In the case of tag caching, a tag cache hit always guarantees correctness of the presence information and any cache prefetches can be issued safely. Without a tag cache, Unison Cache must conservatively assume that the access may be a hit and delay miss serving and any prefetching associated with misses. Miss predictors that are not conservative, such as Alloy Cache's instruction-based miss predictor, are not useful at all to DRAM caches that use prefetching.

Tag caching comes at a several-cycle latency cost for all cache accesses. This cost can be partially avoided if tag caching is not used for early miss detection, which is what the tag cache does least effectively anyways, as illustrated in Figure 4.10. The reason why tag caching is less effective for miss detection, as compared to eviction probes and metadata updates, is because cache misses typically happen as the first access to a page — i.e., the triggering miss — which cannot be captured by tag caches. If we let cache accesses bypass the tag cache and if we use the tag cache only for evictions and metadata updates, which are not on the critical path, the main benefits of tag caching can be preserved without any latency penalty in the common case of cache hits. Eliminating the latency cost of tag caching can allow for larger tag caches, which are more effective in eliminating eviction probes and metadata updates.

4.6 Efficient Footprint Tracking through Sampling

Moving tags into DRAM implies a significant increase in die-stacked DRAM traffic coming from various tag probes and updates. Embedding the footprint predictor metadata into DRAM has similar consequences; upon page insertion, the PC & Offset pair need to be written to every DRAM row that contains a fraction of that page, ideally only one; upon the first access to any block its valid and dirty bits must be updated to indicate that the block has been demanded, for the purpose of footprint tracking, which under low data reuse imposes a significant overhead; upon page eviction, the corresponding PC & Offset pair must be read along with the actual footprint of the page and the footprint history needs to be updated.

To virtually eliminate the predictor metadata maintenance overhead, we make the observation that the prediction history does not need to be updated upon every eviction. Namely, page

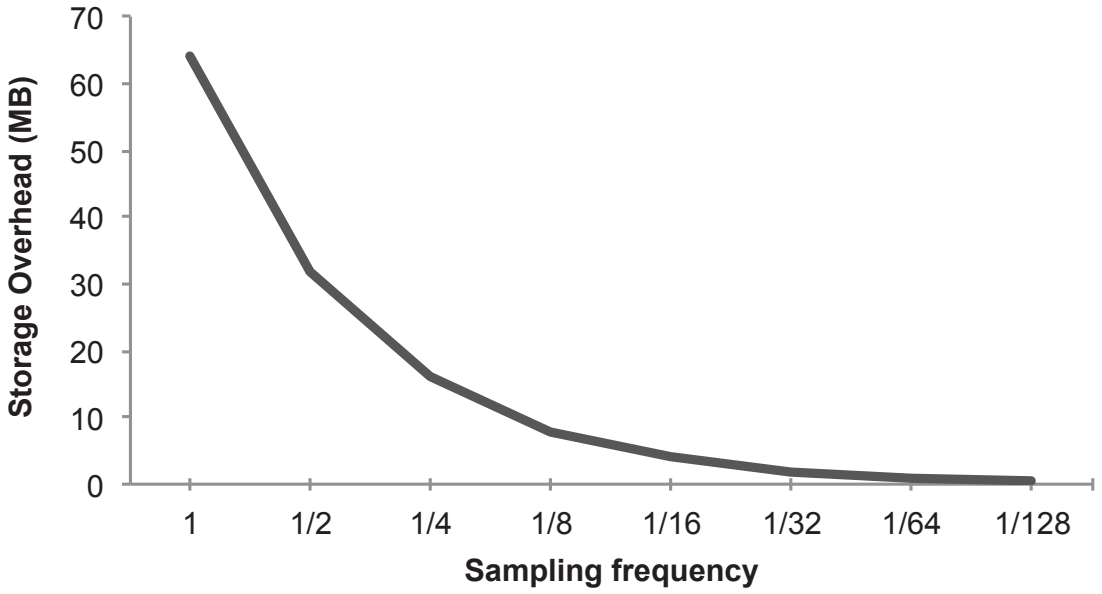

Figure 4.11 – Storage required for PC & offset pairs for an 8GB cache with 1KB pages.

footprint are stable and repetitive, and they are associated to a limited number of PC & offset pairs. As a result, many redundant history updates happen all the time, upon every page eviction, because too many page evictions happen per one PC & offset pair. It is therefore possible to sample the history update process by performing the history updates only for a small fraction of the pages. More precisely, maintaining the PC & Offset pairs in DRAM, tracking the footprints in DRAM upon every cache access and updating the global history upon eviction can be done only for a tiny fraction of the pages, saving bandwidth and energy without *any* accuracy penalty. This can be efficiently implemented through set sampling, where only a number of dedicated sets, for example one out of 64, would generate the prediction history, while all other cache sets would use the history during the page fetch.

Besides reducing the stacked DRAM activity required to maintain the prefetcher metadata by orders of magnitude, the sampling approach can be used to reduce the total storage overhead of keeping the PCs and page offsets in DRAM. A similar idea has been used to reduce the prediction history in dead-block predictors in SRAM caches [35]. While storing the PCs and offsets does not introduce any extra storage overhead in DRAM because of the available unused space in DRAM rows, a significant reduction could allow for storing the PCs and offsets corresponding to the selected history-generating pages in SRAM. For example, tracking only PCs and offsets for every 64th cache set would reduce the storage overhead to only 1MB, as shown in Figure 4.11.

4.7 Page Alignment

In this section we investigate the problems associated with non-power-of-two page sizes and propose a practical way to fully mitigate them.

4.7.1 Effects of Misalignments

Non-power-of-two page size causes problems for Unison Cache in three different ways:

- First, operating system pages are typically 4-8KB, and always aligned to a power-of-two number. Software objects in server applications are typically aligned to 64B to improve on-chip cache locality, but they are also aware of the operating system page size. An obvious example of this awareness are relational databases, where the bufferpool page size is always related to the operating system page size. Let's take an example of MonetDB, which allocates KB objects. To fetch such an object, Unison Cache with 960B pages will always have to bring two objects, which basically doubles the miss ratio. Lets further suppose that the object starts at the beginning of the first Unison Cache page. The last block of the object will be brought into the cache separately, in a different cache page. If it is a scan operation, both the first and the last block will initiate a page access with the same PC, and the same offset (offset 0). Therefore, it will be hard for Unison Cache to distinguish which page is full, and which page is singleton. In this extreme example, the bandwidth efficiency can drop significantly, as well as the predictor accuracy.

- The misalignment between the cache page size and the object size causes a lot of pressure to the footprint history table, because it inflates the number of offsets associated with each PC, and therefore inflates the number of entries that the history table needs to store. For example, with 2KB pages (32 blocks) the number of offsets per PC is typically two. However, with 1984B pages, the number of offsets per PC can go as high as 31. Because the history table is kept in SRAM and has to be small, high pressure on the history table will eventually affect the prediction accuracy.

- Because the cache page size in a non-power-of-two number, it is not enough to keep only some of the address bits as page tags. We either need to keep the whole address as a tag, or do more complex computation to match tags in the cache against an address during tag comparison.

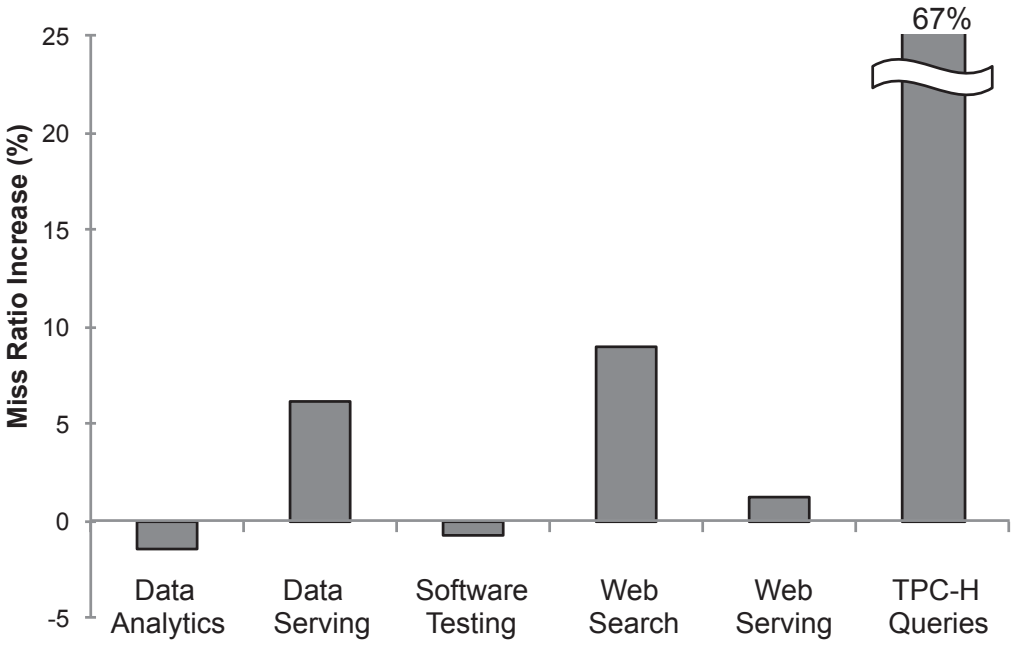

Figure 4.12 – Increase in miss ratio due to misaligned pages.

Figure 4.12 compares the miss ratio of two four-way associative Unison Cache designs. The first design uses 960B pages, whereas the second one uses 1KB pages, assuming that we could somehow efficiently integrate 1KB pages and their metadata. Both designs use a 16KB history table for the footprint predictor. The cache size is 256MB, except for the Data Analytics case, where the cache size is 2GB due to the scale of its dataset (see section 4.3. We show the miss ratio of the first design normalized to the miss ratio of the second design. In other words, the figure illustrates the increase in miss ratio due to the misalignment. All workloads are notably sensitive to the misalignment problem, but TPC-H queries on MonetDB exhibit by far the strongest sensitivity. It miss ratio essentially doubles because of misalignments. This situation suggests a strong need for aligning the pages to a power-of-two number. As expected, the only exceptions are, Data Analytics and Software testing. As explained in chapter, Data Analytics suffers complete absence of spatial locality, whereas Software testing does not even have a dataset, but generates data on the fly. Slightly changing the page size introduces noise that varies across cache sizes.

4.7.2 Mitigating Page Alignment Problems in Unison Cache

Figure 4.13 illustrates a possible data layout of 1KB cache pages in 16 consecutive rows in the stacked DRAM. Note that the page size is exactly 1KB, and not 960B. For simplicity, we assume

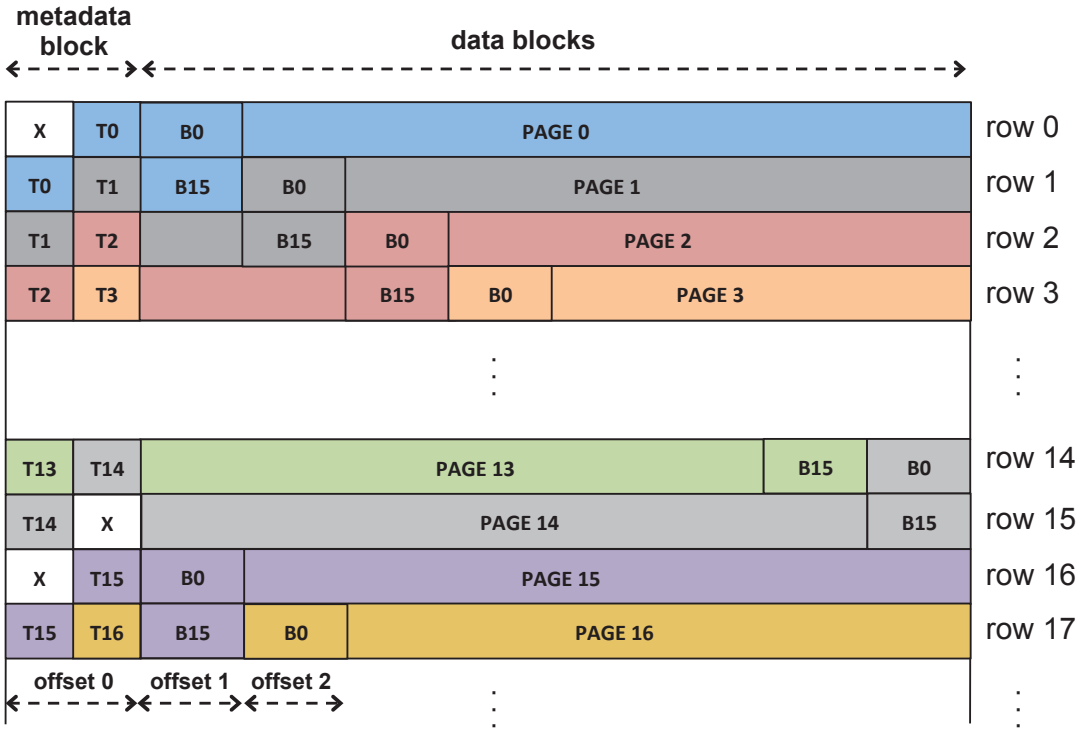

Figure 4.13 – Alternative data layout with 1KB pages.

a direct-mapped cache and 1KB DRAM rows that could accommodate one cache page per row without the page metadata. Because the first 64B in each DRAM row are reserved for metadata for all pages residing in that row, the last cache block of the first page cannot fit and is stored in the subsequent DRAM row. In this example, each cache page spans exactly two DRAM rows, and each DRAM row contains blocks from at most two different pages.

The first block (denoted as *B0*) of *Page 0* in Figure 4.13 is placed in the second block of *Row 0*; we denote its offset as *Offset 1*, while position *Offset 0* is reserved for metadata. The first part of the metadata block contain metadata of the page that ends in the current row (i.e., the parts of the page that could not fit in the previous row), whereas the second part contains metadata of the following page that starts in the current row. Because *Row 0* contains only one page, which starts in that row, the first part of the metadata block is empty, denoted as *X*. The last block of the same page, *B15*, happens to be at the same position in the subsequent row, *Row 1*. The first block of the next page, *Page 1*, occupies the block at position *Offset 2* in *Row 1*, whereas the last block of the same page occupies the same position in the subsequent row. *Page 14* starts at the very end of *Row 14*, and occupies whole *Row 15*. Because there is no page that starts in this row, the second metadata slot in this row is empty, as shown in the figure. The

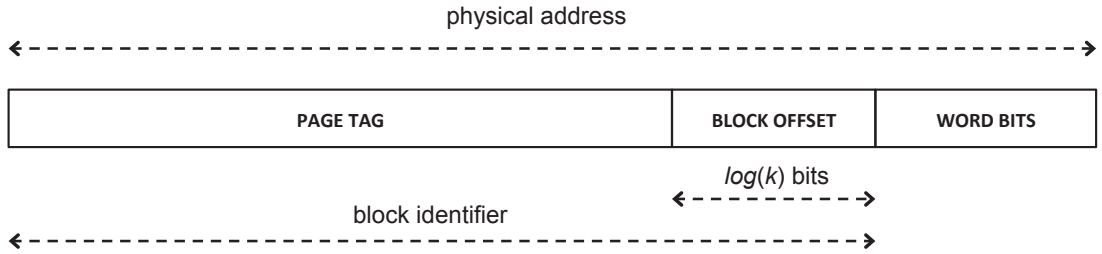

Figure 4.14 – Physical address bits.

layout of *Row 16* is exactly the same as the layout of *Row 0* — i.e., rows 0–15 form a cycle that repeats. Within these 16 rows we are able to accommodate 15 different pages.

While this data layout enables conventional power-of-two page sizes, locating the page upon a request becomes difficult. The target DRAM row cannot be computed only based on the page tag only. In fact, each page spans two different DRAM rows, and the requested block may be in any of them. Furthermore, the physical position of the requested block within the page cannot be determined solely based on the offset. Fortunately, simple circuitry can be used to determine the both the exact DRAM row and the exact position of the requested block within it. The DRAM row where given page starts can be computed as

$$start\_dram\_row = \frac{page\_tag \cdot k}{k-1} = \frac{page\_tag << log(k)}{k-1} \tag{4.1}$$

where *page_tag* represents address bits of without the block offset and the word bits, as illustrated in figure 4.14, k denotes the number of blocks within a page, in this case 16, and << performs the left shift operation. The DRAM row where a given page ends will be simply the next one. The DRAM row where a given block resides can be either the start row or the end row. Regardless of its position, the target DRAM row can be easily determined using the following, even simpler formula:

$$target\_dram\_row = \frac{block\_identifier}{k-1} \tag{4.2}$$

This formula will produce the correct result regardless of the position of the block in its page. The exact position where the requested data block resides within the target DRAM row can be computed as follows:

$$block\_offset = block\_identifier \mod (k-1) + 1 \tag{4.3}$$

The resulting residual in the above formula is incremented by one because of the metadata

block that occupies the first slot in each row. Both the DRAM row location and the block position within it can be computed by a single and fairly simple hardware unit. Division by a fixed small number can be implemented using several adders, producing the required residual as a side effect. Such circuitry is especially simple for numbers in the form $2^n - 1$. For 1KB pages in the above example, the circuitry needed to implement this functionality (i.e., division by 15) consists of four 4-bit adders. We estimate the latency of this computation to take two CPU cycles. This idea can be easily generalized to account for any combination of associativity, cache page size and DRAM row size, as discussed in section 4.2.

4.8 Unison Cache in the Context of Near-Memory Acceleration

This thesis shows that die-stacked caches on average provide a 2-3x reduction in memory traffic of servers chips, postponing the bandwidth wall for a few generations. Even though technological advances will likely allow for larger cache capacities, the rapid data growth and the growth of memory systems hosting the data might offset any benefits. The ultimate solution to the data movement problem lies in moving the computation from the processor closer to the memory, which is becoming possible with the emergence of new DRAM devices that feature a thin layer of logic (e.g., HMC [53, 63]). The idea is to use the available logic layer to offload certain memory-intensive computation and utilize the device's high internal bandwidth, while avoiding off-chip communication. This style of computation is often referred to as near-memory processing, and we refer to these devices as Intelligent Memory Devices (IMD).

The biggest research problem in this context is in finding the exact useful role for the emerging IMDs in server systems, and in their integration with the rest of the system components, and particularly regarding their integration into the virtual memory system and enabling efficient address translation. Address translation could be performed by the IMD itself, if equipped with translation lookaside buffers (TLBs) and page-walker caches. Every computational unit within the logic layer of IMDs must be equipped with a TLB to avoid frequent communication throughout the logic layer only for the purpose of translation. However, placing the TLBs within the IMD causes the problem of TLB coherence, maintenance and inefficient TLB shootdowns. TLBs are also shown to be less and less effective as the memory capacity increases [3, 15, 32].

For a certain class of applications and operations, such as database scans, there is a simple solution to this problem. For such applications, virtual-to-physical address translation can be performed on the processor side, after which a scan request for the entire page is sent to an

IMD for scanning. The observation here is that the translation can be performed once and be valid for each address within the page. The IMD would utilize its high internal bandwidth to efficiently perform the scan operation on the requested page and return only the aggregate result. Unfortunately, this style of computation has very limited applicability, because the server applications rarely scan data sequentially. Instead, they exhibit complex and irregular access patterns as a result of the pointer-intensive data structures they employ [12]. A much more general solution is needed to support address translation for server applications and enable the "pointer is a pointer" semantics.

Virtual memory provides a level of protection and, historically, the illusion of a huge virtual address space for multitasking systems; the latter has been very important for systems in which the memory requirements of all active processes were exceeding the amount of available DRAM by orders of magnitude. General, flexible and fully associative table structures, called page tables, were needed to map any virtual page to any available physical slot. However, in today's datacenters data is almost entirely stored in memory, and the content of memory is much more static; a simpler and more efficient mapping between virtual and physical addresses could be used. An extreme example of such simplification would be direct segmentation [3], where the application tries to allocate most of its memory as a single huge region of memory (segment) and map its virtual address space into a contiguous region of the physical address space. The address translation for that segment is greatly simplified, as the same simple arithmetic is used to compute the physical address of any virtual address belonging to the segment. This technique is applicable to some server applications provided that the whole dataset fits in memory.

In the context of IMDs, Instead of using direct segments, we could employ *direct paging* and provide in-memory address translation. The idea is to keep the traditional page-table based translation for compatibility and flexibility, but avoid any TLB involvement in the translation for the part of memory stored in IMDs. Since datasets mostly fit in memory, one could use a direct-mapped approach to map every virtual page to exactly one possible physical page, which solves the problem of locating the page. The per-page translation metadata would be integrated into DRAM the same way as it is done with tags in Unison Cache [28], and accessed in parallel with data accesses to a page. The proposed system would essentially be a virtual cache, yet having the functionality of main memory. Upon an access, the fetched translation metadata would be consulted to see if the desired page is indeed present. If the entire dataset fit in memory, the vast majority of the time the required page would be found. While the technique of direct segments is not applicable if the dataset exceeds the DRAM capacity by a

single byte, this mechanism would work even for moderately larger datasets. Direct paging is applicable to all cases where direct segments are applicable, but also when direct segments are not. In case the dataset is significantly larger than the available DRAM, a small degree of associativity could be efficiently added to the page mapping process, as it is done in the Unison Cache design, to reduce the incidence of page faults. Unlike TLBs, the way predictor structures in the case of a set-associative solution do not need to be coherent or maintained in software, as they do not impact the translation correctness.

It is important to note that the layout optimization in section 4.7.2 is crucial in applying the Unison Cache idea to in-memory address translation. While in the case of die-stacked caches the page size has only performance implications, in the case of virtual memory the page size has to be aligned to a power-of-two number. Besides virtual memory, this way of metadata integration can be generally used to efficiently tag memory pages in DRAM with arbitrary application-specific metadata, and access the data and tags in unison.

4.9 Summary

This chapter introduced Unison Cache, a practical and scalable stacked DRAM cache design, which brings together the best traits of Footprint Cache and the state-of-the-art block-based design. Unison Cache achieves high hit rates and low DRAM cache access latency, while eliminating impractically large on-chip tag arrays by embedding the tags in the DRAM cache. Cycle-level simulations of scale-out server platforms using Unison Cache show a 14% performance improvement over the state-of-the-art block-based DRAM cache design, stemming from the high hit rates achieved by Unison Cache. Unlike Footprint Cache, Unison Cache requires no dedicated SRAM-based tag storage, enabling scalability to multi-gigabyte stacked DRAM cache sizes.

5 Research Directions for Improving DRAM Cache Efficiency

Research on DRAM caches has been primarily driven by the need for reduction in traffic between the processor and the memory. DRAM caches provide a traffic reduction on the processor side solely through reuse of locally stored copies of data within high capacity on-chip DRAM. What makes the reuse possible is data skew; certain types of data, such as metadata, are more frequently accessed than others; certain objects also happen to be more popular than others. Despite their high capacity, practical DRAM cache sizes are still two orders of magnitude smaller than the off-chip main memory in the following level of the hierarchy and as such cannot accommodate the hot data structures for the majority of the applications [28]. As a result, the amount of temporal reuse in on-chip DRAM caches is fairly low [8, 29].

The abundance of spatial locality and the lack of either temporal reuse or mechanisms to exploit it may lead to cache thrashing. In page-based designs, pages with high spatial locality typically show less temporal reuse and occupy space in the cache for a long time, but are often not useful after they are completely scanned. In block-based designs, most of the data that are inserted into the cache are *dead upon arrival* [8, 29]. It is therefore important to provide DRAM caches with mechanisms that would on one hand minimize the cache space and cache bandwidth resources allocated to data that is not reused, and on the other hand exploit the existing and encourage more reuse among data that are prone to it, and therefore improve the overall cache efficiency.

In this chapter we revisit commonly used techniques for improving cache efficiency in SRAM caches and study their applicability in the context of DRAM caches. We conclude that many of the prior techniques are either ineffective or not directly applicable to DRAM caches. We propose new research directions for improving cache efficiency both in block-based and in page-based DRAM caches.

The rest of this chapter is organized as follows. In Section 5.1 we take a look at the direct and indirect impact of associativity on DRAM caches and practical ways to increase associativity. In Section 5.2 we study adaptive cache insertion policies that protect caches from thrashing, whereas in Section 5.3 we investigate the opportunity to improve the cache behavior through cache bypassing. In Section 5.4 we study the mechanism for timely prediction of dead blocks and pages. Finally, in Section 5.5, we look at the opportunity for prefetching in block-based cache designs.

5.1 Increasing the Associativity in DRAM Caches

The underlying mechanism through which capacity-constrained caches exploit reuse is associativity, which stands for the number of slots into which a new cache entry can be inserted. Another, in certain contexts equivalent definition of associativity is the number different options for selecting the victim entry during cache replacement [60]. Associativity enables control over the placement of data in the cache and over their promotion through recency lists that aim to rank the possible victim options according to the likelihood of their reuse. Unfortunately, practical DRAM cache implementations provide either no associativity at all [8, 55] or very limited associativity [18, 28, 55], which severely limits the cache's ability to identify and keep reusable data in the cache.

Increasing associativity is a trivial technique to reduce conflict misses and support reuse, but comes at a high cost. High associativity in SRAM caches used to be associated with higher access latency. Today, high associativity is expensive primarily from the energy perspective [54]. In L1 caches data and tag lookups are typically performed in parallel due to the strict latency requirements. Looking up all the ways in the tag array and reading out data from every cache way implies substantial energy costs, considering that these operations are performed at every cycle. For that reason highly associative L1 caches usually employ way prediction [54]. Last-level caches typically serialize tag and data accesses to avoid the overhead of reading out all the cache ways, but still look up all the ways in the tag array, although less frequently compared to L1 caches. Regarding DRAM caches, the cost of associativity, as well as its benefits, greatly depend on the cache organization.

5.1.1 Associativity in Block-Based DRAM Caches

To provide efficient support for associativity in block-based DRAM cache designs, all the ways of a set must be placed within the same DRAM row as in the Loh & Hill Cache design [46, 47, 48].

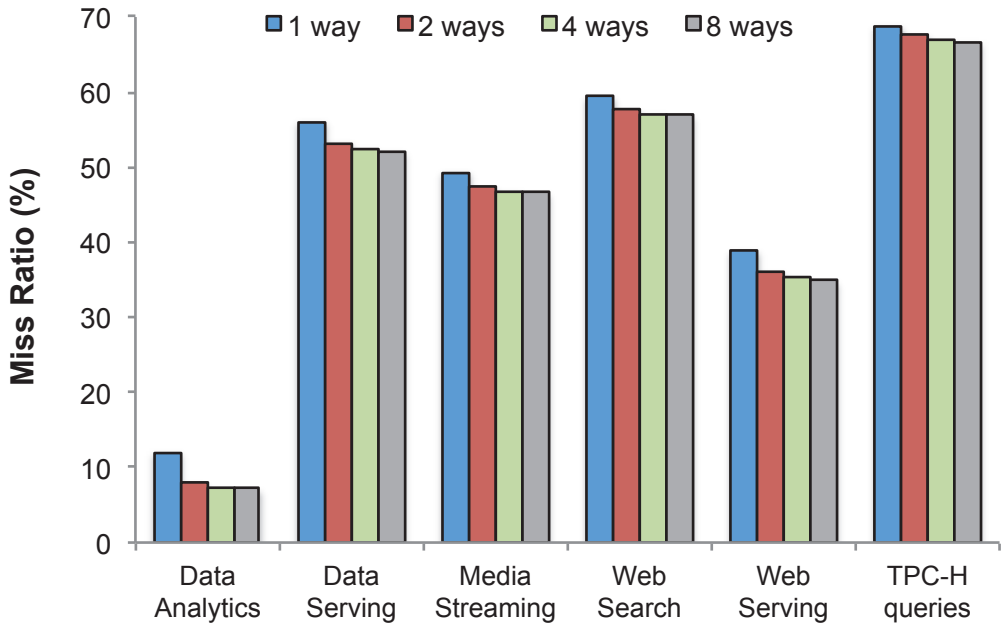

Figure 5.1 – Miss ratio as a function of associativity for a 256MB block-based cache (2GB for TPC-H queries).

In such designs, the beginning of each row is reserved for metadata, and the rest of the blocks correspond to one way each. For example, a 2KB DRAM row could accommodate up to 30 different ways. Unfortunately, this implies that two dependent accesses to the same row are needed: the first access reads the metadata for all the ways (2-3 64-byte blocks) and locates the correct way, whereas the second access fetches the requested block [46]. Unfortunately, this comes at a high latency cost, as two *dependent* die-stacked DRAM accesses may be as costly as an off-chip DRAM access. The two accesses can be merged into one compound access [47, 48] with the help of DRAM controller to guarantee a row-buffer hit for the second access and partially reduce its cost. Unfortunately, the latency penalty is still prohibitively high, because the second access is dependent on the first and the accesses must be therefore serialized.

Fortunately, 30-way associativity that Loh&Hill Cache provides is not in fact needed. Figure 5.1 shows the effect of associativity on block-based caches. We show the miss ratio of two-, four-, and eight-way associative designs normalized to the miss ratio of the direct-mapped design. As expected, the associativity does not play a significant role, partially because of the enormous number of sets and total absence of false conflicts at this level of the hierarchy, but mainly because there is no much sensitivity to cache capacity at this portion of the miss ratio versus

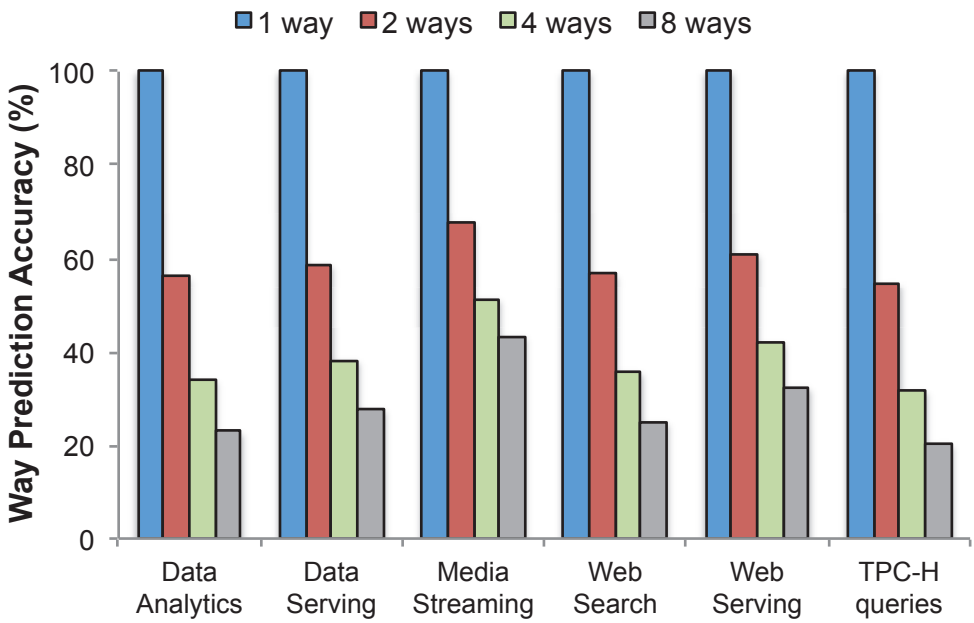

Figure 5.2 – Way prediction accuracy as a function of associativity for a 256MB block-based cache (2GB for TPC-H queries).

capacity curve.[1] Direct-mapped caches are indeed a preferable option [55] for the baseline cache performance. However, associativity still plays an important role in virtually all cache optimization techniques, as we will see in the following sections.

One could still implement a small amount associativity in block-based caches that are organized as Alloy Cache. However, because block-based caches cannot leverage spatial locality, accurate way prediction is not possible. Therefore, associativity in Alloy Cache comes at high cache bandwidth and a non-negligible latency costs. Namely, to fetch a requested block in a single access without way prediction, all the ways have to be read in parallel and serialized on the bus, as we explained in Chapter 4. Figure 5.2 shows the accuracy of an address-based way predictor with 16KB of dedicated storage as we vary the cache associativity. We see that without spatial locality, way prediction is not much better than flipping a coin. The same observation is valid for tag caching in SRAM: only page-based designs can benefit from tag caching, because they leverage spatial locality.

Because there is no opportunity for accurate way prediction, block-based caches must stream together all cache ways, to avoid the high latency penalty. Reading out all the ways together

[1] As we approach the knee of the curve, associativity, and in general most of the techniques that aim to improve cache efficiency, gain more importance.

results in a commensurate bandwidth overhead, which is not an option for die-stacked DRAM because its bandwidth is a highly contended resource [8, 29].

In conclusion, existing block-based DRAM caches have no practical way of supporting associativity and consequently cannot benefit from many standard cache optimization techniques that rely on it. An interesting research direction would be to study way prediction techniques for block-based DRAM caches that could yield better accuracy than way predictors that leverage spatial locality.

5.1.2 Associativity in Page-Based DRAM Caches

In contrast to block-based designs and as evidenced by Figure 4.5, a small amount of associativity is crucial for page-based caches not only for enabling various cache optimizations, but also for the baseline cache performance. In certain cases, a total lack of associativity can double the miss ratio in page-based caches due to *false conflicts*. Such a scenario is very typical in L1 block-based caches due to the small number of sets [22] and a high probability of conflicts.

Page-based caches that keep tags in SRAM, such as CHOP [30] or Footprint Cache [29], can afford arbitrarily high associativity. The tag lookup frequency is substantially lower compared to SRAM caches and so is the power associated with the tag search. Unfortunately, placing the tags in SRAM is not a scalable solution, as it is applicable only to caches of up to at most 512MB. Page-based DRAM caches that keep tags in DRAM must rely on way prediction to avoid reading all the ways in parallel from the die-stacked DRAM [18, 28], which would otherwise incur prohibitive bandwidth and latency costs [28].

The bandwidth cost associated with reading all the ways in parallel[2] is directly proportional to the number of ways, whereas the latency overhead comes from the serialization of the ways during the bus transfer. Fortunately, way prediction in page-based caches is extremely accurate thanks to the spatial locality and repeated accesses to the same page [28]. When using way prediction, the tag metadata for all ways is read along with the data belonging to the predicted way. The advantage of this approach, besides latency and bandwidth savings, is that way mispredictions are not too expensive. Namely, the tags are already read and the correct way is known, while the corresponding DRAM row is already activated and data from the correct way can be read quickly and in at most two trials (in the uncommon case of a way misprediction).

[2] Reading the ways in a serial fashion is not an option, as it would require multiple round-trips to DRAM.

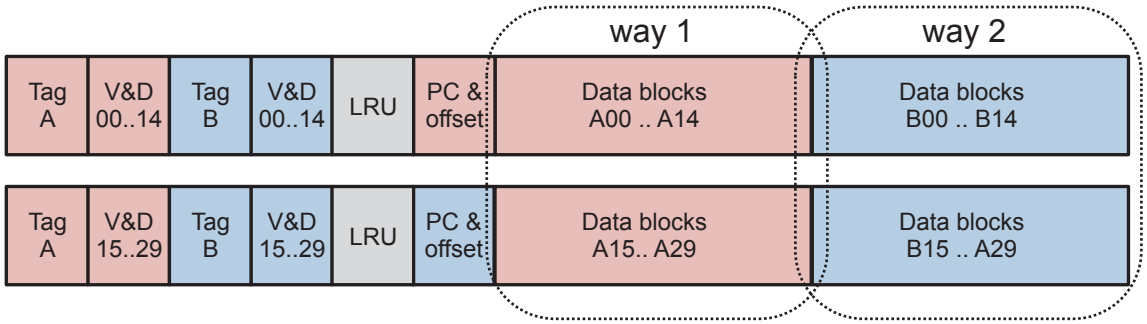

Figure 5.3 – The layout in DRAM of a single cache set in a two-way associative cache with 2KB pages and 2KB DRAM rows. Each cache set consists of two DRAM rows, and each way spans both rows.

An efficient implementation of associativity requires placing all the ways into the same DRAM row. Placing all the ways in the same DRAM row allows certain block-based designs to achieve very high associativity, up to 30 for 2KB rows [46, 47]. However, enabling associativity for page-based designs is severely constrained by the ratio between the DRAM row size and cache page size. The layout depicted in figure 4.3 with associativity of four is only possible because the DRAM row size is 8x larger than the page size. While off-chip DRAM row sizes tend to be quite large (8-16KB) because they must span multiple DRAM chips, on-chip DRAM row sizes are typically smaller because they do not have such a constraint. In a scenario where the on-chip DRAM row size is 2KB and the cache page is 2KB, the associativity would be physically limited to one (i.e., direct-mapped). Reducing the page size could enable higher associativity, but at the expense of lower hit ratio and poor way prediction behavior. To tackle this problem and achieve independence between the DRAM row size, cache page size and associativity, we propose a solution illustrated in 5.3, where we place two 2KB pages of a two-way associative cache in two DRAM rows. The basic idea is still to place multiple ways (logical pages) into the same DRAM row, but to overcome the associativity limitation by splitting each cache page across multiple DRAM rows, as illustrated in Figure 5.3. Each DRAM row holds only a fraction of each page of every cache way, along with the metadata corresponding to only that fraction of the page. Note that the first row contains the tags for both ways, and only a fraction of the valid/dirty bit vectors. Based on the address (i.e., based on the requested block within the page), and the predicted way, the request is forwarded to the appropriate DRAM row.

The prediction metadata — i.e., PC & offset — does not need to be fully present in both DRAM rows, as it is not needed on every access, but only upon evictions. The prediction metadata for one way could be stored in the first row, and the metadata for the other way could be stored in the second row. Because a page spans two rows, both rows need to be read out upon a page

eviction and the correct prediction metadata is read at that time. In contrast, the LRU bits have to be present and correctly maintained in both rows upon every access. However, thanks to the spatial locality, more than 90% of the accesses are MRU accesses, which do not require any updates to the LRU bits. Therefore, maintaining the LRU information in both DRAM rows causes only a negligible activity overhead that could be further reduced by caching of the LRU metadata in a small amount SRAM.

Mapping the cache content into DRAM in the described way allows for complete independence between the DRAM row size, cache page size and associativity, and therefore allows arbitrarily high associativity for page-based designs. An interesting further research direction could be towards enabling adaptive page sizes and variable associativity. It is important to note that it is spatial locality, exposed through larger page sizes, that enables associativity in DRAM caches through accurate way prediction. An important research question that needs to be answered is: can we leverage associativity to improve cache efficiency in page-based DRAM caches?

5.2 Cache Insertion and Promotion Policies

Some of the traditional cache replacement optimizations, such as LRU Insertion Policy (LIP) and its variations [56], aim to protect caches from thrashing. LIP inserts all incoming cache blocks into the LRU position, promoting them to the MRU position only upon the second access — i.e., upon the first reuse. In doing so, LIP conservatively limits the amount of space that blocks with no reuse can occupy in the cache, letting such blocks to compete for a single way in the cache, and leave more space for reusable blocks. Because of the limited reuse in DRAM caches in server applications, LIP is expected to perform well for block-based DRAM cache designs. Although LIP requires associativity and block-based DRAM caches can neither support associativity efficiently nor directly benefit from it, as Figure 5.1 shows, it would be still interesting to see if associativity-based techniques could benefit block-based DRAM caches if they had the necessary support for it and whether enabling associativity in block-based DRAM caches is worth studying.

Figure 5.4 and Figure 5.5 show the increase in hit ratio LIP achieves normalized to the baseline four-way set-associative block-based cache at 256MB for CloudSuite applications. Most of the applications show significant increase in hit ratio. Note the difference in scale among the figures. The vertical distance between the curves is also directly proportional to the off-chip bandwidth savings LIP provides, because the baseline cache is block-based and any hit ratio increase comes solely from temporal reuse. The horizontal distance in the figure directly

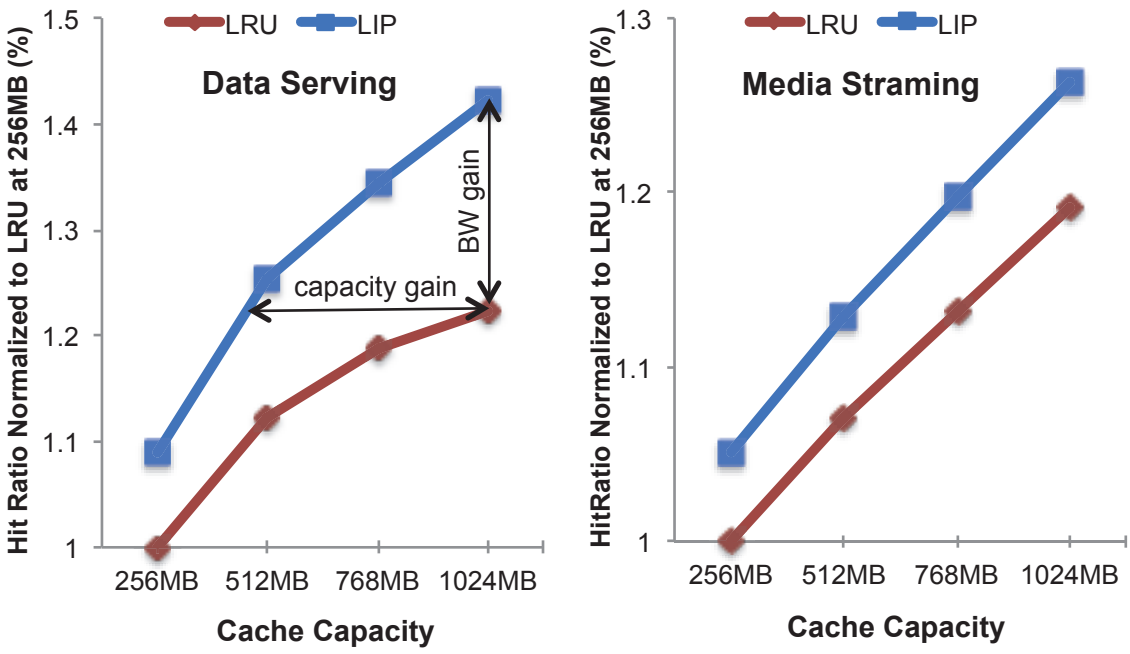

Figure 5.4 – Hit ratio in a four-way associative block-based DRAM cache with the traditional LRU policy and MRU Insertion (LRU) and with LRU Insertion (LIP) for Data Serving (left) and Media Streaming (right). Note the difference in the scale. Increase in hit ratio is directly proportional to the off-chip bandwidth savings (vertical dimension). The horizontal distance in the figure directly corresponds to the difference in effective capacity between the designs.

corresponds to the difference in effective capacity between the designs. For example, in Data Serving a 512MB cache that uses LIP behaves the same as a 1GB cache that does not use LIP. Note that the net effect of LRU Insertion highly depends on the cache size, i.e., it depends on how far we are from the next knee in the miss curve. The point at which the curves cross each other in the case of Web Serving is a starting point after which a block is more likely to be reused than not. We do not show the results for Data Analytics, because the baseline hit ratio is already very high at 256MB and the advantage of LIP compared to LRU is too small to show. The only outliers are TPC-H queries, where LRU consistently performs better than LIP (not shown). The negative effect of LIP on some applications, such as TPC-H queries, could be completely avoided using adaptive mechanisms, set-dueling, that dynamically choose between LRU and LIP.

We also tested a variant of LIP, called Bimodal Insertion Policy (BIP), which with a probability of 1/32 inserts incoming block at the MRU and the rest at the LRU position, with the intent to

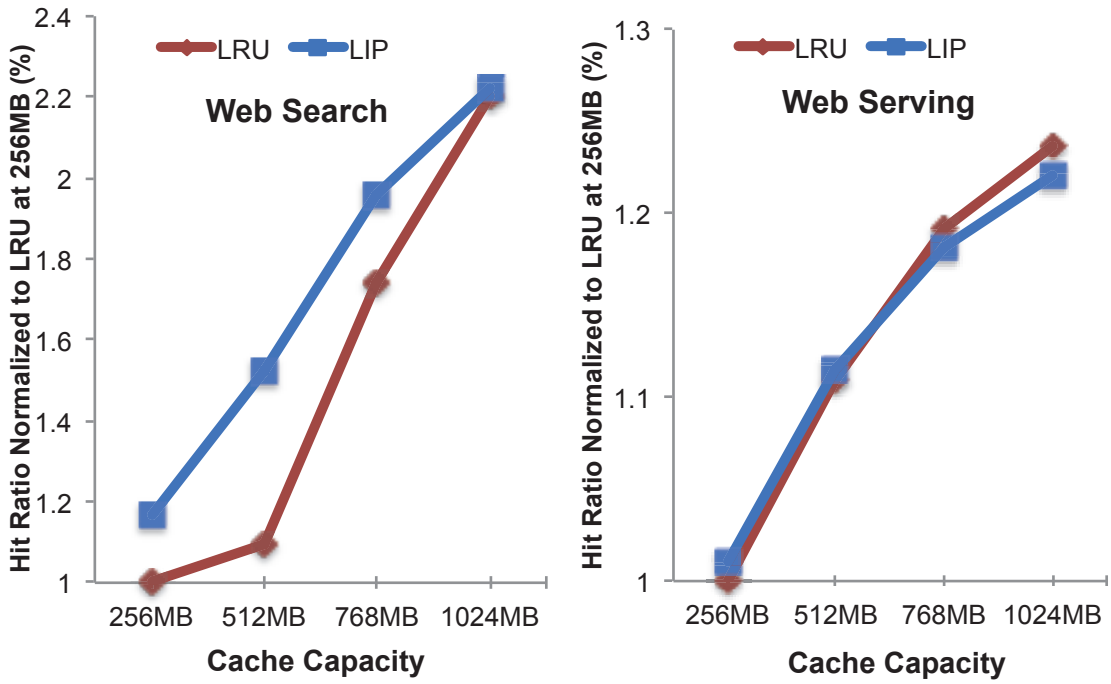

Figure 5.5 – Hit ratio in a four-way associative block-based DRAM cache with the traditional LRU policy and MRU Insertion (LRU) and with LRU Insertion (LIP) for Web Search (left) and Web Serving (right). Note the difference in scale. Increase in hit ratio is directly proportional to the off-chip bandwidth savings (vertical dimension). The horizontal distance in the figure directly corresponds to the difference in effective capacity between the designs. The point in the figure on the right at which the curves cross each other is a starting point after which a block is more likely to be reused than not.

better adapt to the workload needs. We found that, for all of our benchmarks and data points, BIP slightly degrades the cache performance as compared to LIP. The Dynamic Insertion Policy (DIP) that dynamically chooses between the standard LRU and BIP would consequently also be inferior to LIP. The only exception is TPC-H queries, where LIP performs slightly worse than LRU, and BIP partially bridges this gap by cancelling out LIP's effect.

Unfortunately, Figure 5.4 and Figure 5.5 only demonstrate the lost opportunity that block-based DRAM caches could realize if they supported associativity, because LIP fundamentally relies on associativity. In practice, associativity in block-based designs incurs prohibitively high latency and on-chip bandwidth costs that would likely offset any gains in hit ratio and off-chip bandwidth. It is therefore important to study the ways in which associativity in block-based caches can be efficiently increased, for example through PC-based way prediction.

While associativity can be practically implemented in page-based designs, LIP and its variants would not be applicable to such designs for a different reason. Namely, LIP insertion of the whole page upon the first access promotes the page's early eviction, while a different block of the same page is likely to be used soon. Similarly, promotion of a page to the MRU position upon the second access, which is most often an access to a different block within the same page, in fact does not encourage temporal reuse. While such situations do happen due to page reuse — i.e., due to spatial locality — the effective temporal data reuse does not happen. LIP will therefore confuse spatial locality within a page for temporal reuse and promote the page in question, penalizing pages that do exhibit temporal reuse.

Unison Cache's encodings for valid and dirty bits enables distinction between a block being already demanded by the cores and being only present in the cache.[3] It is therefore trivial to detect the first reuse on any block. We also tried promoting the page to the MRU position upon actual temporal reuse of any of its blocks, but we did not see any benefits because of the impact that neighboring blocks have on each other with respect to the replacement policy. In this particular case, such a policy will discourage spatial locality by inserting and keeping the page at the LRU position for a long time until it sees temporal reuse.

5.2.1 Promotion upon Evictions

Every cache policy seeks to promote a cache entry to the most recently used (MRU) position on the premise that the entry will be used again. This always happens upon a cache hit. Unless the cache is implemented as a victim cache, caches rarely promote an entry upon receiving a dirty eviction, because it is assumed that a block evicted from the previous level in the hierarchy will not be required soon. Because of the limited temporal reuse in DRAM caches, this policy works better for block-based caches. However, in page-based caches, a block eviction from a higher-level cache should promote the whole page to the MRU position on the premise that more evictions to the same page are expected to come. Although this approach does not increase the number of cache hits, it has a significant impact on energy, because the die-stacked cache also serves for write coalescing, as explained below.

Assume that a completely scanned page receives an eviction. Our experience with server workloads shows that if there is a single dirty block in a page, highly likely all of the present blocks will eventually become dirty, therefore more evictions to the same page are expected to happen. If the page is not promoted, it may eventually become evicted from the cache before it receives evictions for the rest of its blocks. In such cases, every eviction is sent to the main

[3] A block being only present means that the block has been prefetched, but not yet demanded.

memory separately, most of them incurring an off-chip DRAM row activation. Off-chip DRAM rows are much larger than on-chip DRAM rows, because they span multiple chips that work in lockstep, and each row activation incurs a proportionately higher energy cost. Coalescing all those evictions in the die-stacked DRAM and writing them back to the main memory all together with a single off-chip row activation saves a significant amount of energy and results in more preferable, sequential access patterns. For that reason, in our page-based designs we promote a page to the MRU position upon any received eviction.

5.2.2 Summary

Block-based caches could greatly benefit from the LRU Insertion Policy (LIP) [56], which places incoming blocks at the least-recently used position, and promotes them only upon reuse. However, LIP fundamentally relies on associativity, which existing block-based cache designs cannot provide in a practical way. We strongly encourage research on cache associativity in block-based caches. Further research into page insertion, promotion and replacement policies in page-based DRAM caches would also be interesting to pursue. However, our hypothesis is that research in replacement policies, whether with block-based or page-based caches, could be overshadowed by dead-block and dead-page prediction. Namely, optimizations to insertion/promotion/replacement policies always aim to minimize the residency of the cache content that will not be needed again. The policies in this case will be just a tool towards that goal, which is to evict the dead content. Identifying dead content and its eviction from the cache, in our opinion, is more general and has more potential.

5.3 Cache Bypassing

Cache bypassing is a well-known technique that aims to identify non-reusable cache blocks and avoid storing them in the cache to prevent cache pollution. In the context of DRAM caches, it is an important technique for block-based designs because it does not rely on associativity.

Figure 5.6 shows the fraction of blocks that were reused before their eviction in a direct mapped block-based cache. In this experiment we track only evictions, which excludes the cache resident blocks and includes only blocks that move in and out of the cache. We see that on average 70% of the content that is brought into the cache is *dead upon arrival*. Identifying such blocks and letting them bypass the cache could significantly improve performance.

Cache bypassing can be particularly useful in block-based DRAM caches, because they lack

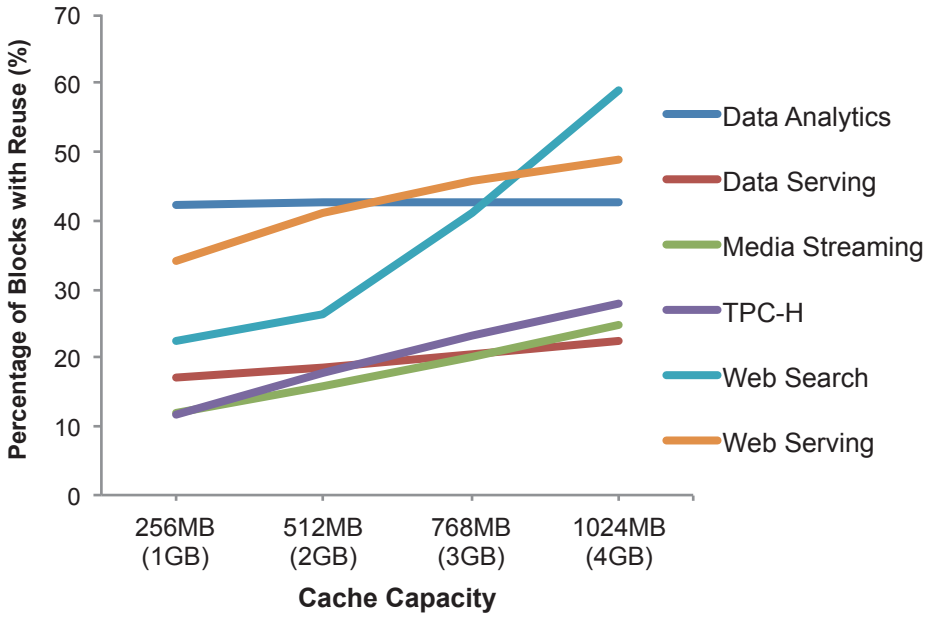

Figure 5.6 – Fraction of evicted blocks that were reused before their eviction for various cache sizes. The cache is organized as direct-mapped block-based cache. The cache size for CloudSuite applications is between 256MB-1024MB, whereas for TPC-H queries the cache size is between 1GB and 4GB. The reuse of all blocks that flow in and out of the cache on average ranges between 23% for the smallest cache size, and 37% for the largest cache size.

associativity needed for the majority of other optimization techniques. A large body of research looked at cache bypassing in conventional block-based SRAM caches, proposing various address-based, PC-based and hybrid reuse predictors to identify blocks that exhibit no reuse. These predictors maintain a global history state used for making predictions, and local state within each cache block used for generation and maintenance of the global history. Address-based and hybrid reuse predictors require the state for global history that is proportional either to the dataset size or to the cache size, and as such are not an option for DRAM caches and server applications. In contrast, PC-based predictors require much less storage for history, because its size is proportional to the application's instruction working set. Because blocks that show no reuse are always accessed once and therefore always by a single instruction, such instructions could serve as a prediction of potential reuse or the absence of it.

Some PC-based predictors require keeping local state consisting of the program counter that initiated the first access to the block and various flags/counters. This state is kept next to the every cache block. Block-based DRAM caches, however, do not have the possibility of keeping such state next to every cache block. The best performing block-based cache, Alloy Cache [55], keeps the cache tag next to each cache block, which together form a Tag-And-Data (TAD) unit,

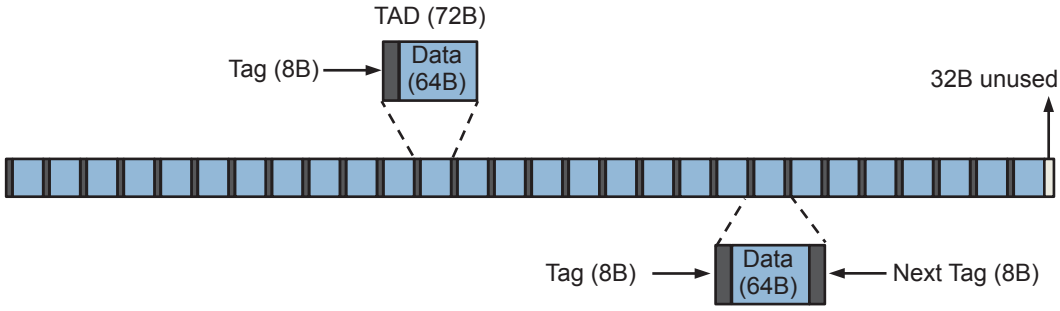

Figure 5.7 – DRAM row layout in Alloy Cache and BEAR.

shown in Figure 5.7. The tag overhead is 8B per 64B of data — 12.5% of the whole cache is dedicated to tags. Because of the irregular address mapping, the whole tag needs to be stored in the cache, except the last six bits corresponding to the byte within a block, which can be used as valid/dirty and coherence bits. Because the remaining one or two bits (depending on the coherence protocol) cannot be used to integrate the program counter or its hashed value — any such state would need much more storage. However, integrating even an extra byte of metadata would require doubling the metadata storage to 25% of the stacked DRAM capacity, which is unacceptable.

5.3.1 Random Cache Bypassing

A recent proposal, called BEAR [8], optimizes Alloy Cache by introducing stateless, random cache bypassing. BEAR builds on the observation that a large number of cache lines experience no reuse due to low temporal locality. Such lines do not save any off-chip bandwidth, but their placement in the cache and its eviction from the cache introduce unnecessary traffic to die-stacked DRAM. To lower the pressure on die-stacked DRAM bandwidth, BEAR caches only randomly selected 10% of the content brought onto the chip. As a side effect, by inserting only 10% of the blocks into the cache, BEAR replaces the cache content at a 10x slower rate and proportionately increases its residency in the cache. Because the content in the cache is more static, the amount of reuse among the data in the cache increases, eventually providing cache hits. To be hit-ratio neutral, BEAR needs to increase the cache reuse by a factor of ten. If the whole dataset is accessed randomly, random bypassing will on average be hit-ratio neutral.

Figure 5.8 plots the average number of accesses per cache block with and without random bypassing. As expected, BEAR significantly increases the reuse among the blocks in the cache. However, the increase in reuse (7.5x) is not proportional to the increase in the average cache residency (10x), and BEAR therefore may experience significant losses in hit ratio. Random

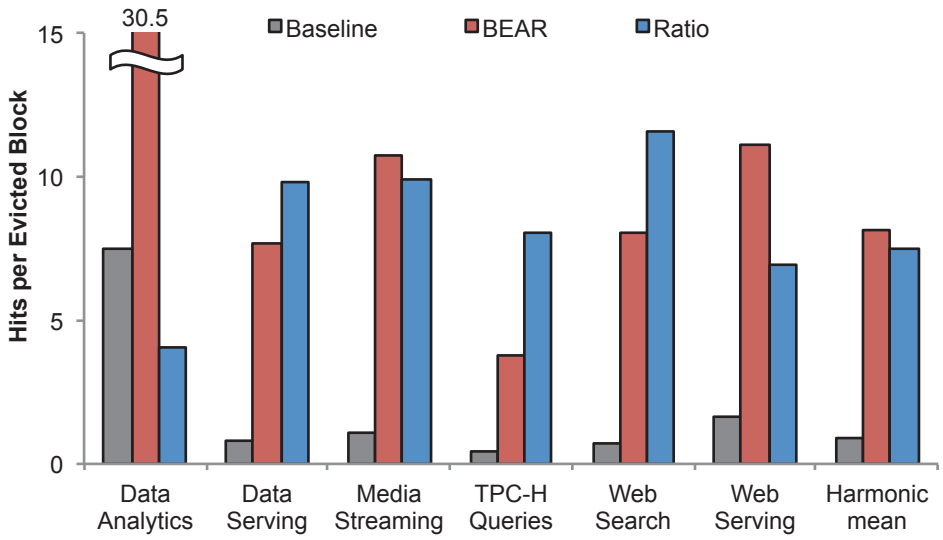

Figure 5.8 – Average number of accesses per block with (BEAR) and without (Baseline) random bypassing for a 256MB cache (2GB for TPC-H Queries), and the ratio between the two.

cache bypassing could however increase the hit ratio in situations when the working set is accessed in a cyclical manner, but the cache is not large enough to capture it all, and therefore thrashes. Because random cache bypassing significantly prolongs the residency of data in the cache, it will provide hits for at least the part of the dataset that it covers. By making cache more static, aggressive random bypassing avoids thrashing.

Besides cache bypassing, BEAR also makes the observation that, due to the width of the bus (16B), each TAD read actually results in an 80-byte transfer. Instead of discarding the extra 8B (*next tag* in Figure 5.7), it understands that these 8B constitute the tag for the next block, that might likely be requested soon, and stores it in a small tag cache to avoid unnecessary DRAM accesses on sequential tag probes that may turn out to be misses.

On one hand BEAR may degrade hit ratio, but on the other hand, BEAR provides a 10x reduction in the number of cache insertions upon cache misses. The overall cache bandwidth savings are much smaller though, because the cache still needs to be probed upon misses. Assuming that a hit ratio is 50%, typical for block-based caches, BEAR saves around 30% in total cache traffic. However, we find that BEAR can increase the off-chip traffic by up to 20% for two reasons: lower hit ratio and lower miss predictor accuracy. Random bypassing may confuse Alloy Cache's PC-based miss predictor and significantly degrade its accuracy. For any hit that was mispredicted as a miss, there is an unnecessary request sent off-chip. Similarly, for any cache miss that was mispredicted as a hit, miss serving is postponed. BEAR's applicability

therefore highly depends on the available off-chip and on-chip bandwidth resources.

5.3.2 PC-Based Cache Bypassing

Cache bypassing significantly prolongs the residency of data inside the cache. It is therefore important to carefully select the data that will be placed in the cache and ensure their high reuse.

It would be interesting to understand if we could make a decision as to what to place in cache that is better than random. To answer that question, we study the correlation between data reuse and instructions that access the data. The intuition behind possible existence of such correlation is that certain types of data, such as metadata, tend to be more frequently accessed than others. It is important to note that within the same type of data, certain objects are more popular than the others, but this type of popularity skew cannot be captured by looking only at the code & data reuse correlation.

An obvious way to leverage the correlation between the code and reuse is to monitor the hit ratio of instructions that access the cache and place into the cache only blocks accessed by instructions that frequently hit. Similar ideas have been proposed in the past for SRAM caches [17, 64]. Because of the large number of instructions that access SRAM caches, previous proposals assume predictors that keep a 2-bit saturating counter or a similar tiny state machine associated with each instruction. The state is used to predict whether an instruction will reuse the block it is currently missing, and the decision whether to place the block in the cache or not is made accordingly. When the instruction accesses the cache, the state is updated according to the outcome (hit or miss). While these solutions may work well for L1 caches, such a small state can hardly predict the overall reuse behavior of an instruction in DRAM caches that hold gigabytes of data. Whether an instruction has recently hit in the cache or not does not tell us much about the importance to cache that type of data. The same instruction may reuse certain data, and not reuse other data. What we would like to know is whether the type of data accessed by a particular instruction is worth caching in general or not.

PC-based approaches may introduce a systematic bias by placing into the cache only blocks accessed by instructions that currently achieve high hit ratio, not allowing other instructions to improve their hit ratio. The gap in hit ratio between the instructions that place data into the cache and those that bypass it can be artificially created and increased only because of the bypass cache policy. The only way an instruction's hit rate history can be changed in such situations is when another piece of code prefetches a data block later accessed by the

instruction. The problem arises because the instruction statistics is created by monitoring the behavior of the cache while it follows a certain policy, which systematically introduces a bias.

To correctly monitor behavior of instructions, we propose an approach whereby only a sample of cache sets, called *monitor sets*, generate instruction hit rate statistics and those cache sets never employ bypassing.[4] The rest of the cache sets, called *follower sets* read the history created by the monitor sets and decide whether to bypass the cache or not based on the instruction's hit rate. The hit rate of instructions is monitored only by the monitor sets, which do not use bypassing. The monitor sets are also the only ones who create the statistics, avoiding the bias introduced by bypassing. Besides avoiding bias, this approach also allows for a practical implementation in block-based DRAM caches that cannot keep any predictor state next to every block. We illustrate its design below.

Practical Implementations

To maintain the precise hit rate behavior, we propose maintaining history table organized as a tagless 64K-entry array with two-byte entries. The array is indexed by a 2-byte XOR hash of the program counter of instructions that miss in the cache. The hash directly points to the two-byte array entry, where the first byte indicates the number of hits the instruction has experienced so far, whereas the second byte counts the number of misses. The history is probed upon every cache miss in the follower sets. A prediction to bypass the cache is made if the computed hit ratio of the instruction is less than a dynamically determined threshold, otherwise the instruction is placed into the cache. The history is updated upon any access to the monitor sets. Depending on whether the access is a hit or a miss, the appropriate counter in the history table is updated. Because the history is updated upon every access to the monitor sets, there is no need to keep any information within the blocks belonging to either the monitor or the follower sets.

Other approaches to PC-based bypassing, such as counter-based bypassing [36], are also applicable. Counter-based bypassing uses a PC-based predictor to estimate the number of accesses for every cache block based on the instruction that brought it into the cache. The approach was proposed in the context of dead-block prediction, where a block would be declared dead after a predicted number of accesses. Blocks predicted to have a single access or fewer than a certain threshold are not stored in the cache. While dead-block prediction is not applicable to direct-mapped block-based caches, counter-based cache bypassing could still be applied.

[4] Note that in direct-mapped block-based caches, one set corresponds to one block

The counter-based approach requires keeping the PC information and the access count within every block in order to maintain the history, which is updated upon every cache eviction. Keeping the PC and a counter next to every cache block, writing them on every insertion and reading them on every eviction, along with the counter increments in DRAM upon every access would result in a prohibitive overhead. Instead, one could use a sampling approach to history maintenance, in which only the last block in each DRAM row maintains the PC of the instruction that brought it into the cache and the access counter. Because this block is the last one in the row, and because there is 32B of unused space, as shown in Figure 5.7, this way of integrating predictor metadata practically comes at no storage cost and no bandwidth costs, because the extra 8B after the last block 8B are read anyways and they do not store the tag for the next block that could be useful. In this implementation, the last block in each row is the history generator block, the only block that updates the history table and creates the reuse history. The rest of the blocks only read the history and make predictions. Similar to the first approach, the generator block should always store every requested block in the cache, to be able to determine which of them will show no reuse.

Our findings indicate that cache bypassing based on instruction hit rates is superior to counter-based bypassing. The reason is that the approach based on the hit rates has a better picture of the overall behavior of each instruction and can make a more informed decision about bypassing. The mechanism we propose for efficient metadata integration for counter-based bypassing could be, however, useful for integrating other predictors into block-based DRAM caches.

Comparison with Random Bypassing

Random cache bypassing was proposed not to improve the cache hit rate but to reduce the number of cache insertions [8]. To make a fair comparison between random bypassing and PC-based bypassing, we dynamically tune the hit ratio threshold an instruction needs to reach for its blocks to be cached so that both designs insert 10% of cache blocks upon insertion. For PC-based bypassing we designate the last block in each row as a monitor block. The monitoring sample thus constitutes 1/28th of the effective cache capacity for a 2KB DRAM row.

Figure 5.9 compares the baseline Alloy Cache design against the designs with random cache bypassing (BEAR) and PC-based bypassing for Data Serving (left) and Media Streaming (right), normalized to the baseline design at 256MB. The designs that use bypassing insert blocks into the cache at the same rate and therefore achieve the same on-chip bandwidth reduction.

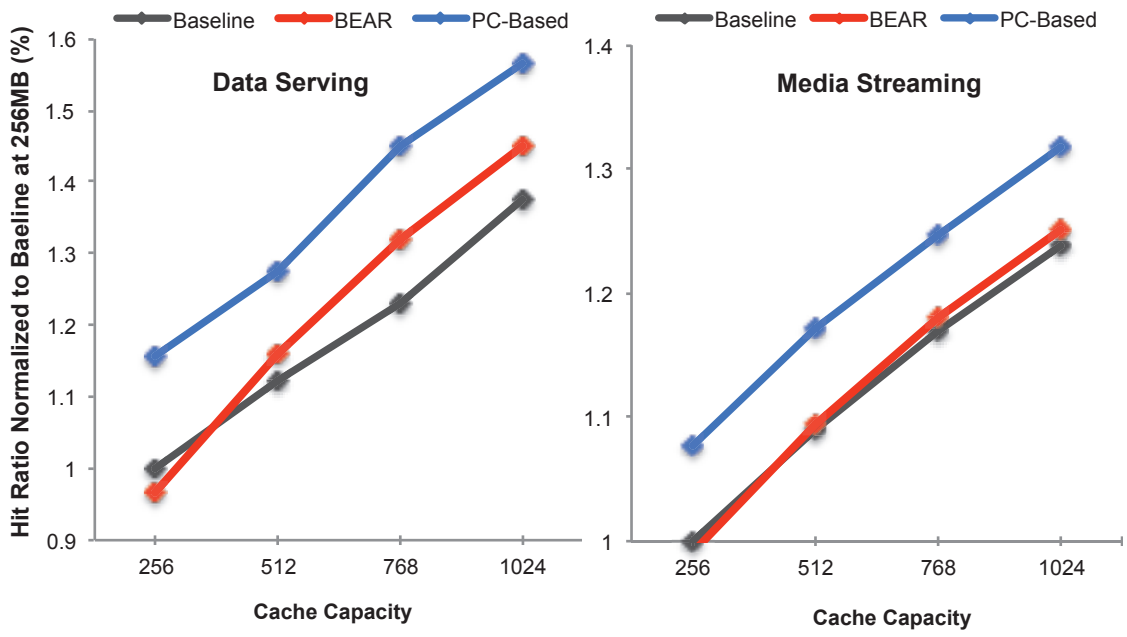

Figure 5.9 – Hit ratio of a direct-mapped Alloy Cache (baseline), Alloy Cache with random cache bypassing (BEAR) and PC-based bypassing for Data Serving (left) and Media Streaming (right), normalized to the baseline design at 256MB. Note the difference in the scale. Increase in hit ratio is directly proportional to the off-chip bandwidth savings (vertical dimension). The horizontal distance in the figure directly corresponds to the difference in effective capacity between the designs

While BEAR reduces the hit ratio compared to the baseline, PC-based bypassing actually increases the hit ratio significantly. For Data Serving, a 256MB cache with PC-based bypassing performs better than the baseline cache with 512MB, while significantly reducing the number of cache insertion.

Figure 5.10 shows the results for Web Search (left) and Web Serving (right), normalized to the baseline design at 256MB. Web Search has a working set of around 800MB that is uniformly accessed. As discussed in section 5.3.1, applications that uniformly access their data benefit from random bypassing. For such applications, PC-based bypassing performs marginally better than random bypassing. For cache sizes above 800MB, all designs capture the working set and show no difference in behavior. In contrast, bypassing hurts Web Serving. Random bypassing drastically reduces the hit rate, while PC-based bypassing performs only slightly worse than the baseline.

Figure 5.11 shows the results for Data Analytics (left) and TPC-H queries (right), normalized to

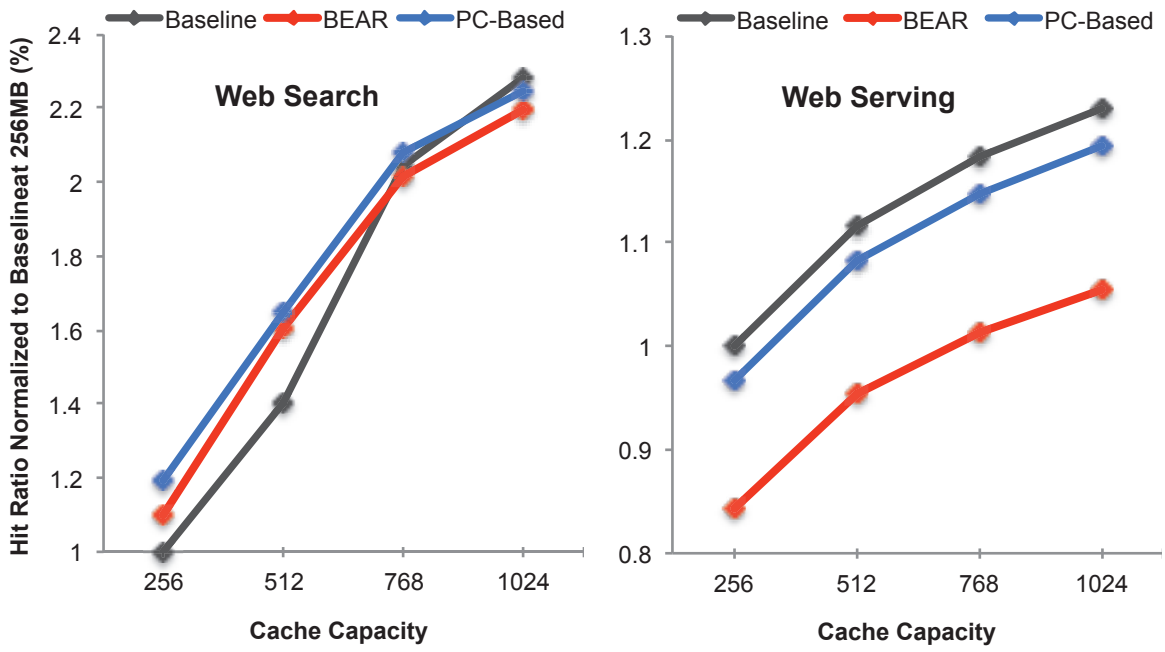

Figure 5.10 – Hit ratio of a direct-mapped Alloy Cache (baseline), Alloy Cache with random block bypassing (BEAR), and PC-based bypassing for Web Search (left) and Web Serving (right), normalized to the baseline design at 256MB. Note the difference in the scale. The horizontal distance in the figure directly corresponds to the difference in effective capacity between the designs

the baseline design at 256MB. Data Analytics' exhibit very high reuse, as shown in Figure 5.8, and does not benefit from bypassing. Data Analytics is a Hadoop MapReduce application that slowly streams through vast amounts of data that is heavily reused. A mix of TPC-H queries behave very similarly in that respect except that their data structures greatly exceed the cache capacity and the level of reuse is much lower. Because cache bypassing makes the cache content much more static by slowing down the rate at which cache content is replaced, cache bypassing, especially random bypassing, consistently degrades the performance of these two applications.

Summary

Figure 5.9, Figure 5.10, and Figure 5.11 demonstrate two important things. First, aggressive cache bypassing does not necessarily have to degrade the cache hit rate. Instead, hit rates can be significantly improved through better selection of the blocks for insertion. Second, the correlation between the code and data reuse exists in server applications. Note that we

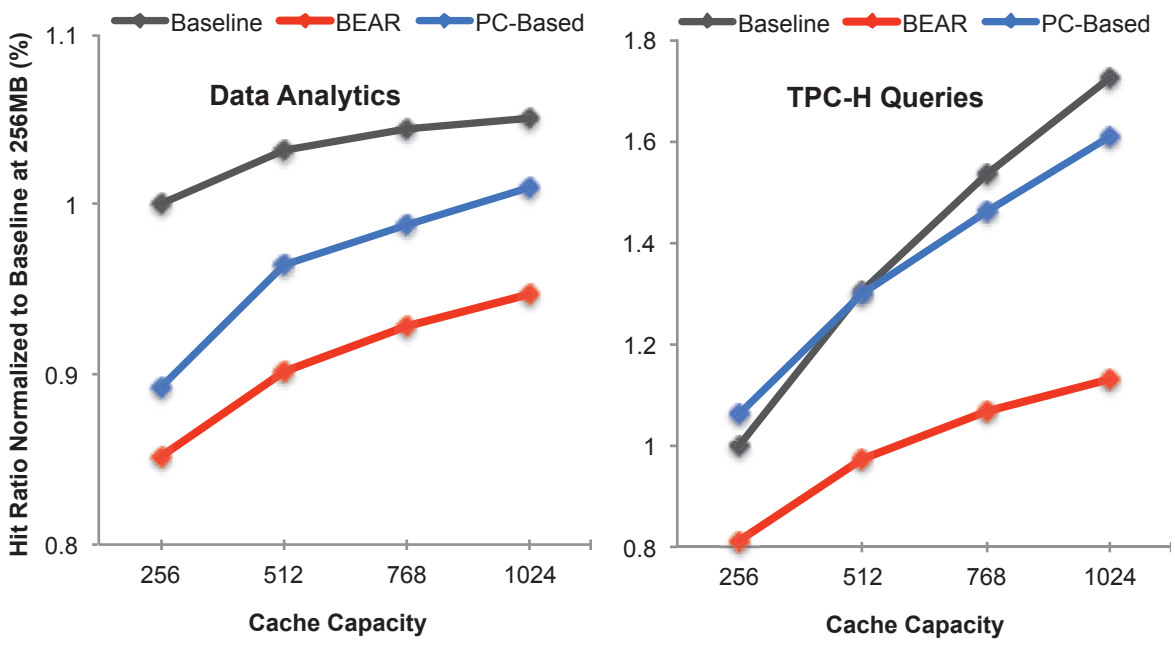

Figure 5.11 – Hit ratio of a direct-mapped Alloy Cache (baseline), Alloy Cache with random block bypassing (BEAR), and PC-based bypassing for Data Analytics, normalized to the baseline design at 256MB (left) and TPC-H queries, normalized to the 1GB baseline (right). Note the difference in the scale. The horizontal distance in the figure directly corresponds to the difference in effective capacity between the designs.

designed the PC-based scheme to directly leverage the skew among the hit rates of different instructions. Its significant advantage over random bypassing at the same bypass frequency shows the extent of the correlation. Leveraging such correlation could be a fruitful research direction.

Block-based DRAM caches are direct-mapped and have no choice when selecting a victim block for eviction. The question whether to place incoming block A into the cache could be thus turned into a question whether to keep existing block B in the cache or not; or more generally, should we replace block A by block B. Designs that could intelligently compare the two blocks to decide on replacement would probably yield the best result, but they require certain metadata about every block residing in the cache. Integration of such metadata into DRAM would be a challenge, but could provide a much better control over the cache content. We strongly encourage further research into this topic.

5.3.3 Cache Bypassing in Page-Based Designs

Similarly to block-based caches, cache bypassing could be applied to pages that exhibit no reuse. In fact, a large fraction of pages are only scanned once and never reused. From the off-chip bandwidth perspective, caching these pages does not contribute to any reduction in off-chip bandwidth, and causes more traffic to and from the die-stacked DRAM. However, caching those pages is important for two reasons: hit ratio and energy.

Regarding the hit ratio, it is spatial locality that gives page-based designs a significant advantage over block-based designs. Letting these pages bypass the cache would significantly lower the hit ratio and the overall performance. As for the energy, we already mentioned in Section 5.2.1 that die-stacked DRAM also serves as a large prefetch buffer that coalesces reads from and writes to the off-chip DRAM. While caching these pages may slightly increase the energy in die-stacked DRAM, it provides more significant savings in the off-chip DRAM. Therefore, cache bypassing for page-based pages is not a reasonable option not only due to the cost in hit ratio, but also due to the loss in opportunity for energy savings in off-chip DRAM.

5.4 Dead-Block and Dead-Page Prediction

Dead-block prediction is a well-established approach to improving cache efficiency [23, 35, 36, 40, 44]. As the name says, such techniques seek to *timely* identify cache blocks that will not be referenced again before they become evicted. Accurate dead-block prediction can be useful in cache replacement, to improve the LRU algorithm by replacing dead blocks first, or in coherence protocol optimizations, where a dead block can be self-invalidated early. The other possibility is to repurpose dead blocks and use them as a prefetch buffer to prefetch data into. In the context of DRAM caches we consider dead-block prediction as a replacement optimization that increases the effective cache capacity for data that exhibit reuse.

Dead-block prediction could be accurately performed in block-based DRAM caches, but the lack of cache associativity severely limits its usability. Namely, after a dead-block prediction, dead blocks are either moved to the LRU position to promote their eviction, or the replacement algorithm takes into account the existence of dead blocks and evicts them before the block that currently sits at the LRU position. Unfortunately, such optimizations do not apply to direct-mapped caches. Blocks predicted to be dead-upon-arrival could be used for bypassing, which we covered in Section5.3.

In set-associative page-based designs, *dead-page prediction* has the potential to limit the

cache residency of pages that show low reuse. We see two practical approaches to dead-page predictions:

- Counter-based. Similar to counter-based dead-block prediction, counting the number of accesses to a page and correlating it with the code that fetched the page into the cache could serve as an estimate of the number of future accesses by the same code. Unlike counter-based dead-block prediction which tries to predict only temporal reuse, counter-based dead-page prediction tries to predict both temporal and spatial reuse. The approach requires storing the counter within every page, and updating it upon every access, without the possibility to sample. However, the updating overhead can be virtually removed with practical tag caching, as we explained in Section 4.5.
- Pages that stream through the cache, fully scanned but never reused, are common in certain applications, such as Media Streaming, TPC-H queries and Web Search. The moment when a page becomes fully scanned, i.e., when every block in the page becomes demanded, is easy to identify in the Unison Cache design and requires no additional state.

Trace-based approaches, which mark a block as dead once it has been accessed by a certain sequence of instructions [40], could also be generalized to page-based designs. A page would be announced dead if the signature containing all instructions that have accessed the page so far matches a signature in the history of sequences that led to dead pages in the past. Unlike the counter-based approach, which uses only the PC of the instruction that brought a block into the cache to make predictions, the trace-based approach can distinguish between pages that are brought into the cache by the same instruction but are referenced by different instruction sequences [44]. However, sequences that lead to dead pages in page-based designs can be quite long, even in the case of pages that are just scanned once. These sequences are at least an order of magnitude longer compared to sequences that lead to death of a block in SRAM caches. The explosion of the history and the complexity of history lookups make trace-based designs impractical in the context of page-based DRAM caches.

Time-based approaches have also been proposed for dead-block prediction in SRAM caches [23], and could be generalized to dead-page prediction in page-based DRAM caches. However, the residency of pages in a DRAM cache is orders of magnitude longer compared to the residency of blocks in SRAM caches. Eventual temporal reuse happens between different server requests after long and unpredictable intervals, so we expect that time-based approaches are less accurate in estimating the reuse. Time-based approaches could be useful for pages that

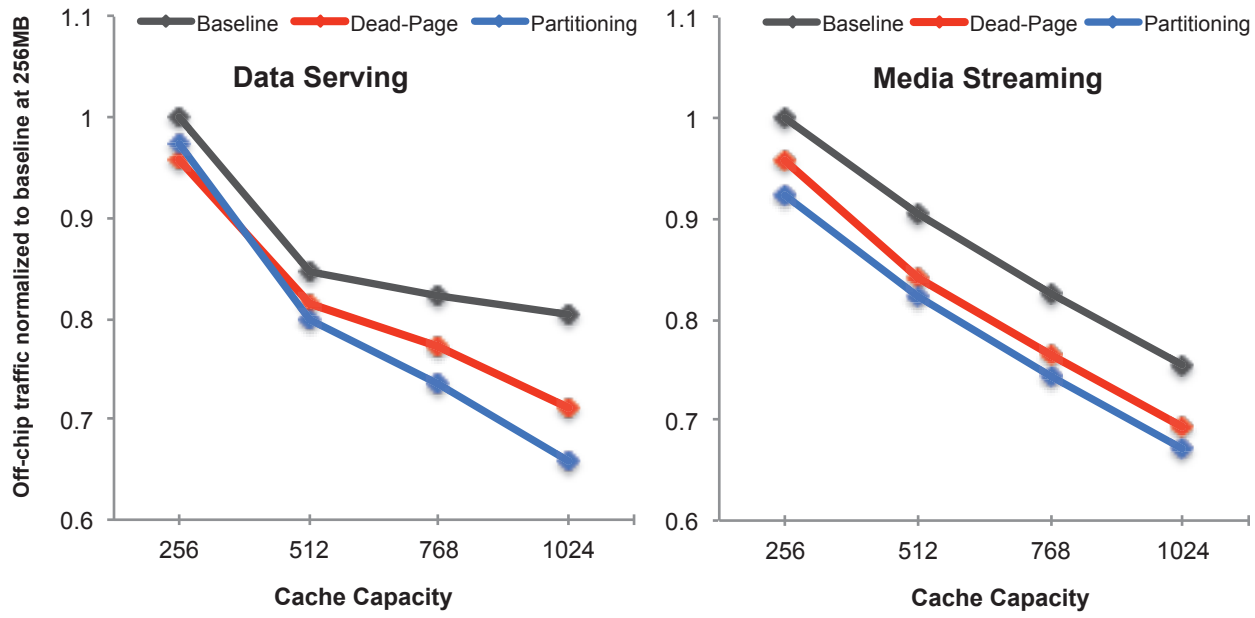

Figure 5.12 – Off-chip traffic in Data Serving and Media Streaming in the baseline four-way Unison Cache, Unison Cache with counter-based dead-page prediction and Unison Cache with cache partitioning. The results are normalized to the off-chip traffic of the baseline design at 256MB. The horizontal distance in the figure directly corresponds to the difference in effective capacity between the designs.

are sequentially accessed once, but such predictions could be made by other, less complex mechanisms.

Another way to limit the time a low-reuse page spends in the cache is by physically limiting the cache space allocated to such pages. Similar to cache partitioning into a spatial and a temporal cache [17], one could way-partition a page-based cache into a part with high reuse, and a part with low reuse, which do not interfere with each other. As a practical example, a four-way associative Unison Cache could dedicate three ways to pages with high reuse and run the LRU replacement algorithm among the three ways, while the remaining way could be used as a smaller direct-mapped cache for pages with high spatial locality and low reuse.

5.4.1 Quantitative Comparison

We compare a baseline four-way associative Unison Cache with our vanilla implementations of counter-based dead-page prediction and cache partitioning.

For the counter-based approach, we maintain a counter per page counting the total number of accesses to the page, including evictions (as mentioned in section 5.2.1, eviction hits are important accesses in page-based caches). Upon a page-eviction, we the read the counter from the page's metadata and update the Footprint History Table (FHT) used for footprint predictions, whose entries are also augmented to store an access counter and a confidence bit indicating the counter's stability. Upon a miss, we probe the FHT to read the expected page footprint but also the expected number of accesses to the incoming page. If the confidence flag indicates that the counter is stable, we store that information in the inserted page as well. On every cache access, including evictions, we increment the access counter in a page and check if the counter has reached the expected number of accesses. Once the number of access has reached the expected value, we move the page to the LRU position to promote its eviction if the confidence bit was set upon the page's insertion.

For the cache partitioning scheme, we dedicate one way to the pages with low reuse and the other three to pages with high reuse. The bigger partition runs internally runs the LRU algorithm. We measure the reuse in a page as a ratio between the number of accesses and the number of touched blocks (the size of the footprint). This approach requires keeping a small counter next to each page to update the history correctly. The reuse history is associated with the instruction that brought the page into the cache. Note that the space and activity overhead the counter and its maintenance introduce can be significantly reduced by sampling. A page is inserted into the smaller partition if its expected reuse is lower than a threshold. In this study we statically set the threshold to two.

Figure 5.12 compares the two approaches against the baseline for Data Serving and Media Streaming in terms of off-chip traffic.[5] With the parameters we selected for our vanilla designs, only these two applications showed a significant improvement, while the others were less sensitive. However, in these two applications both techniques show significant potential to increase the effective capacity. A 512MB cache with cache partitioning performs better than a baseline cache at 1GB for Data Serving. Note that cache partitioning performs slightly better than dead-page prediction in our setup.

This experiment confirms the correlation between the code and data reuse, and shows that the correlation can be used to limit the time and space allocated to pages with low reuse. We strongly encourage further research into both dead-page prediction and cache partitioning.

[5] For page-based designs, hit ratio is not a representative metric of reuse.

5.5 Prefetching

Page-based designs implicitely rely on spatial prefetching to boost their hit ratio. Prefetching could significantly benefit block-based designs as well. However, as explained in Section 4.2.1, the lack of centralized and reliable information about the presence of neighboring blocks in the cache severely limits the applicability of spatial prefetchers in practical block-based DRAM caches, such as Alloy Cache.

Alloy Cache could still implement an effective next-line prefetcher of degree one. Upon every access, regardless of whether it is predicted to be a hit or a miss, Alloy Cache probes the cache and fetches the tag and the data, but also the tag for the next block as a side effect. At that point, if the tag for the neighboring block does not correspond to the next block address, a prefetch request can be sent to memory. However, if the first access was predicted to be a miss, the two requests, the miss and the prefetch, would be sent off-chip separately and the prefetch request may not hit in the off-chip row buffer. Because the two blocks will also arrive separately, they likely won't be stored in on-chip DRAM with a single row activation. In contrast, in page-based designs a whole page (or its footprint) is read from memory with one off-chip DRAM row activation and stored in the cache with one on-chip DRAM row activation. While a limited form of prefetching is still possible in block-based designs, prefetching in page-based designs is much more efficient.

Because practical block-based DRAM caches achieve low hit rates, designing an effective prefetcher for them is of great importance. We therefore highly encourage further research on effective prefetchers for block-based designs.

6 Related Work

6.1 Die-Stacked DRAM

Die stacked DRAM has been recognized as a powerful technology to improve DRAM latency, bandwidth, and capacity. Many researches have tried to exploit the advantages die-stacking provides and address the challenges it imposes [16, 51], assuming die-stacked DRAM in form of a main memory [19, 25, 34, 43, 45] or a large cache [8, 18, 24, 30, 46, 47, 48, 55, 72]. Die-stacked DRAM has been also studied in the context of heterogeneous memory systems, where it is employed as a hardware- or software-managed extension to off-chip main memory [7, 11, 41, 52]. In the context of server processors, practical on-chip die-stacked DRAM capacities are insufficient to meet the memory needs of modern servers; it is virtually impossible to fit all the main memory distributed across multiple multi-chip DRAM modules onto a single processor chip. Such a constraint forces the architects to use the on-chip stacked DRAM as a hardware-managed cache [30, 46, 47, 48, 55, 72] or as a software-managed cache or scratchpad [11, 41]. Managing die-stacked DRAM in software is a preferable option in custom designs where hardware and software evolve together, such as embedded systems. In contrast, deep, diverse and rapidly changing software stacks in server systems rely on general-purpose processors and operating systems, mandating non-intrusive hardware-based solutions.

In this thesis we extensively covered block-based and page-based DRAM cache proposals. A *bimodal* DRAM cache, which supports both granularities has also been proposed as a compromise between the two [18]. Bimodal Cache places the tags and other metadata into a dedicated DRAM bank, which is accessed in parallel with the data bank. The problem with such an approach is that it is not possible to precisely balance the bandwidth and, more importantly, the latency between the data and metadata accesses. For example, the effective

access latency of a cache access would be the bigger of the tag access and data access latency. Tag caching could, however, reduce the imbalance by making metadata accesses less frequent. In this thesis we show that, besides data caching, accesses related to prediction metadata can be drastically reduced through sampling.

ATCache is a recent proposal that leverages tag caching in the context of block-based DRAM caches [24]. However, because of the absence of spatial locality, tag caching is much less effective in block-based designs. To mitigate the poor performance of tag caching in block-based designs, ATCache prefetches tags for nearby cache sets to improve the tag cache hit ratio.

Jiang et al. were the first to argue for page-based DRAM caches to leverage spatial locality [30]. Before them, Woo et al. explored the spatial locality of desktop applications, concluding that large cache blocks in L2 caches boost performance better than conventional prefetchers, if supported with a high-density TSV bus connected to die-stacked main memory [25]. They also argued that fetching larger regions from DRAM in the open-page mode can save a substantial amount of power by minimizing the number of row activations. While most of the researchers agree that large cache blocks are beneficial for overall performance for systems that are not bandwidth-constrained [14, 25, 30], some proposed filtering of unused data. Lin et al. proposed filtering of unused data coming from aggressive prefetchers [42].

Page-migration between off-chip and on-chip DRAM is a hardware-software solution that maintains most frequently accessed OS pages in the die-stacked DRAM [11, 41]. The tag array functionality is offloaded to TLBs, whereas the page replacement is done in software. The latter allows for greater flexibility regarding page replacement but makes it longer compared to hardware-based solutions. The main disadvantage of this technique is that the granularity of transfers is an OS page, which is not an optimal unit as we have shown in this thesis. Systems that employ super pages to minimize the address translation overhead are further penalized by page-migration.

6.2 Spatial Prefetchers

Instruction-based predictors are used extensively in data prefetching [6, 62], dead-block prediction [40], last-write prediction [67], traffic reduction in networks-on-chip [37], and on-chip and off-chip fetch granularity speculation [39, 65, 70]. Our footprint predictor builds upon previous work on spatial memory streaming [62], which estimates the footprints of spatial regions, based on the first instruction that accesses a region, and prefetches them into

SRAM caches. The opportunity for spatial streaming in SRAM caches is remarkably lower though, because most of the spatial locality in the application is revealed in large DRAM caches.

The idea of Footprint Cache is conceptually similar to the work of Kumar and Wilkerson [38], who used a similar predictor based on spatial footprints to predict and fetch only useful words within an L1 cache block, and store such words in a decoupled sectored cache [61]. Their predictor, though, relies on the missing instruction and the full missing address, requiring larger history storage and covering only previously accessed data.

6.3 Cache Bypassing

Cache bypassing has been widely used to improve SRAM cache efficiency [17, 27, 31, 36, 57, 64]. Tyson et al. proposed bypassing based on the recent hit rate of the missing load/store instruction [64]. Johnson et al. proposed bypassing based on the reference frequency of the data being referenced [31] but put bypassed blocks in a separate buffer parallel to the cache. Jalminger and Stenstrom proposed bypassing based on the reuse distance of the missing block [27]. Gonzalez et al. proposed to bypass L1 data cache blocks with low temporal locality [17].

In the context of DRAM caches, a recent proposal, called BEAR [8], optimizes Alloy Cache by introducing random cache bypassing. BEAR builds on the observation that a large number of cache lines experience no reuse due to low temporal locality. Such lines do not save any off-chip bandwidth, but their placement in the cache and its eviction from the cache introduce unnecessary traffic to die-stacked DRAM. To lower the pressure on die-stacked DRAM bandwidth, BEAR caches only randomly selected 10% of the content brought onto the chip. We find that random cache bypassing can significantly degrade the cache hit rate and confuse PC-based predictors. In contrast, we demonstrate that PC-based cache bypassing can not only save on-chip DRAM bandwidth, but also improve the cache hit rate.

6.4 Way Prediction

Way prediction is widely employed in SRAM caches to allow for an energy-efficient implementation of high associativity. Prior work on way prediction has found that address-based way predictors are the most accurate way predictors for L1 caches [4, 54], mostly because they can leverage spatial locality. However, such predictors are not an option for L1 caches because

the actual address is not known at the time when the prediction has to be made for L1 blocks. We do not have such a constraint in DRAM caches. While the accuracy of address-based way predictors is found to be around 85% for individual blocks [4, 54], Unison Cache's way predictor achieves much higher accuracy (~95%), because it operates at the page level. The abundant spatial locality leads to repeated accesses to the same page; subsequent accesses to the same page result in correct predictions.

6.5 Dead-Block Prediction

Dead-block prediction is a well-established approach to improving cache efficiency [23, 35, 36, 40, 44]. Accurate dead-block prediction can be useful in cache replacement, to improve the LRU algorithm by replacing dead blocks first, or in coherence protocol optimizations, where a dead block can be self-invalidated early. The other possibility is to repurpose dead blocks and use them as a prefetch buffer to prefetch data into. In the context of DRAM caches we considered dead-block prediction as a replacement optimization that increases the effective cache capacity for data that exhibit reuse, and are therefore applicable only to set-associative caches, which in practice implies page-based designs.

Lai et al. were the first to propose the concept of dead-block prediction and proposed a trace-based approach to identifying dead blocks [40]. Trace-based approaches, which mark a block as dead once it has been accessed by a certain sequence of instructions, are not practical for page-based DRAM caches because of the amount of state they require and the complexity of state lookups. We also believe that time-based approaches [23] are less applicable to DRAM caches due to the large variation in cache residency of pages in multi-gigabyte caches and their complex reuse patterns. In contrast, we demonstrated that counter-based approaches [36] to dead-page prediction have a great potential to improve DRAM cache efficiency in page-based designs.

7 Conclusions

Die-stacked DRAM has been advocated as a promising technology to break the memory bandwidth wall and improve memory latency and density. It delivers several times more internal bandwidth compared to off-chip memory due to dense on-chip TSV buses, as well as lower access latency due to reduction in physical distances enabled by die stacking. Recent advances in die-stacking technologies have made it possible to tightly integrate a sizeable amount of DRAM in the same chip as the processor. Having die-stacked DRAM on the chip could virtually eliminate the memory bandwidth wall by exposing all of its internal bandwidth at lower access latency. The latency advantage that die-stacked on-chip DRAM provides over conventional off-chip DRAM is particularly important in server applications, which are known for being memory-bound.

Technological constraints, however, limit the on-chip stacked DRAM capacity to levels that are orders of magnitude lower than what modern server applications demand. It is impossible to fit all the main memory distributed across multiple multi-chip DRAM modules onto a single processor chip. Such a constraint forces the architects to use the on-chip stacked DRAM as a hardware-managed cache or as a software-managed cache or scratchpad. Managing die-stacked DRAM in software is a preferable option in custom designs where hardware and software evolve together, such as embedded systems. In contrast, deep, diverse and rapidly changing software stacks in server systems rely on general-purpose processors and operating systems, mandating non-intrusive hardware-based solutions.

This thesis investigated the use of on-chip die-stacked DRAM as a hardware-managed cache in processor chips for datacenters with the purpose of reducing memory traffic on the processor side and improving memory latency. We provided a detailed characterization of real-world server software stacks with respect to DRAM caches in order to gain the critical insights

that lead to appropriate cache designs. We demonstrated the potential of die-stacked DRAM caches to reduce memory traffic and improve memory latency in server systems, and proposed effective, scalable and energy-efficient designs that realize that potential.

In this thesis we argued that effective and efficient DRAM cache designs for servers must leverage spatial locality and must do so in a bandwidth- and capacity-efficient manner. Using analytic models, trace-driven and cycle-accurate full-system simulation of modern, real-world server workloads, this thesis demonstrated that:

- High capacity on-chip DRAM caches expose abundant spatial locality of server applications and their modest temporal locality. As a consequence, DRAM caches that manage and fetch data at a coarser granularity exhibit overall superior properties compared to caches that do fine-grain management. These properties include higher hit rates, smaller tag storage, and higher energy efficiency. However, their naïve employment results in excessive data overfetch and capacity waste that can offsets any benefits of DRAM caches.

- If the cache is organized as page-based, page footprints — i.e., the set of blocks that are touched while the page is in the cache — are highly predictable using well-established code-correlation techniques. Predicting page footprints can eliminate most of the bandwidth overhead and capacity waste that page-based caches suffer from. We demonstrated such a design, called Footprint Cache.

- Fetching whole page footprints at once and writing them back together to the main memory greatly improves the energy efficiency in off-chip DRAM by reducing the number of DRAM row activations by an order of magnitude as compared to fetching the same set of blocks separately.

- In contrast to block-based caches, page-based caches need a modest amount of associativity to avoid frequent conflicts. Associativity can be efficiently implemented through way prediction, which is highly accurate for and only for page-based designs. We demonstrated an efficient implementation of arbitrarily high associativity for page-based designs.

- It is possible to build a scalable, associative, low-latency page-based cache design with DRAM-based tags that achieves high hit rates and high bandwidth efficiency. We demonstrated such an implementation, called Unison Cache, in this thesis.

- Although associativity is not crucial for the baseline cache performance in block-based DRAM caches, its absence disables many standard cache optimization techniques that block-based caches could otherwise greatly benefit from.
- There is a correlation between the code and data reuse. In the absence of associativity, block-based DRAM caches could leverage this correlation and perform cache bypassing not only to reduce the cache activity but also to increase the hit rate. Page-based caches can leverage the correlation between the code and data reuse to employ dead-page prediction and increase cache efficiency.

This thesis showed that die-stacked caches on average provide a 2-3x reduction in memory traffic of servers chips, postponing the bandwidth wall for a few generations. Even though technological advances will likely allow for larger cache capacities, the rapid data growth and the growth of memory systems hosting the data might offset any benefits. We believe that the ultimate solution to the data movement problem lies in moving the computation from the processor closer to the memory, which is becoming possible with the emergence of new DRAM devices that feature a thin layer of logic. This style of computation is often referred to as near-memory processing, and we refer to these devices as Intelligent Memory Devices (IMD).

The biggest research problem in this context is in finding the exact useful role for the emerging IMDs in server systems, and in their integration with the rest of the system components, and particularly regarding their integration into the virtual memory system and enabling efficient address translation. In this thesis we also demonstrated how Unison Cache's idea of metadata integration can be extended to facilitate the integration of IMDs into the virtual memory system.

Bibliography

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: where does time go? In *Proceedings of the 25th International Conference on Very Large Data Bases*, Sept. 1999.

[2] M. Alisafaee. Spatiotemporal coherence tracking. In *Proceedings of the 45th International Symposium on Microarchitecture*, Dec. 2012.

[3] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift. Efficient virtual memory for big memory servers. In *Proceedings of the 40th International Symposium on Computer Architecture*, June 2013.

[4] B. Calder, D. Grunwald, and J. Emer. Predictive sequential associative cache. In *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture*, Feb. 1996.

[5] J. F. Cantin, L. M. H., and J. E. Smith. Improving multiprocessors performance with coarse-grain coherence tracking. In *Proceedings of the 32nd International Symposium on Computer Architecture*, May 2005.

[6] C. F. Chen, S.-H. Yang, B. Falsafi, and A. Moshovos. Accurate and complexity-effective spatial pattern prediction. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, feb 2004.

[7] C. Chou, A. Jaleel, and M. K. Qureshi. Cameo:a two-level memory organization with capacity of main memory and flexibility of hardware-managed cache. In *Proceedings of the 47th International Symposium on Microarchitecture*, Dec. 2014.

[8] C.-C. Chou, A. Jaleel, and M. K. Qureshi. BEAR: Techniques for mitigating bandwidth bloat in gigascale dram caches. In *Proceedings of the 42nd International Symposium on Computer Architecture*, June 2015.

[9] CloudSuite benchmarks. http://parsa.epfl.ch/cloudsuite.

[10] Y. Deng and W. P. Maly. Interconnect characteristics of 2.5-d system integration scheme. In *Proceedings of the 2001 International Symposium on Physical Design*, ISPD '01, pages 171–175, New York, NY, USA, 2001. ACM.

[11] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi. Simple but effective heterogeneous main memory with on-chip memory controller support. In *Proceedings of the 2010 International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2010.

[12] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2012.

[13] M. Ferdman, S. Somogyi, and B. Falsafi. Spatial memory streaming with rotated patterns. In *the 1st JILP Data Prefetching Championship*, Feb. 2009.

[14] P. A. Franaszek, L. A. Lastras-Montan, S. R. Kunkel, and A. C. Sawdey. Victim management in a cache hierarchy. *IBM Journal of Research and Development*, 50(4/5):507–523, jul-sep 2006.

[15] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift. Efficient memory virtualization: Reducing dimensionality of nested page walks. In *Proceedings of the 47th International Symposium on Microarchitecture*, Dec. 2014.

[16] M. Ghosh and H.-H. S. Lee. Smart refresh: An enhanced memory controller design for reducing energy in conventional and 3d die-stacked drams. In *Proceedings of the 40th International Symposium on Microarchitecture*, Dec. 2007.

[17] A. Gonzalez, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *Proceedings of the 9th International Conference on Supercomputing*, 1995.

[18] N. Gulur, R. Govindarajan, R. Manikantan, and M. Mehendale. Bi-modal dram cache: Improving hit rate, hit latency and bandwidth. In *Proceedings of the 47th International Symposium on Microarchitecture*, Dec. 2014.

[19] A. Gutierrez, M. Cieslak, B. Giridhar, R. G. Dreslinski, L. Ceze, and T. Mudge. Integrated 3d-stacked server designs for increasing physical density of key-value stores. In *Proceedings*

of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, Mar. 2014.

[20] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th International Symposium on Computer Architecture*, June 2009.

[21] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(4):6–15, July-August 2011.

[22] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 5th edition, 2011.

[23] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the memory system: predicting and optimizing memory behavior. In *Proceedings of the 41st International Symposium on Microarchitecture*, May 2002.

[24] C.-C. Huang and V. Nagarajan. Atcache: Reducing dram cache latency via a small sram tag cache. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, Aug. 2014.

[25] D. Hyuk Woo, N. H. Seong, D. L. Lewis, and H.-H. S. Lee. An optimized 3d-stacked memory architecture by exploiting excessive, high-density tsv bandwidth. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture*, Jan. 2010.

[26] ITRS. http://www.itrs.net/Links/2011ITRS/Home2011.htm.

[27] J. Jalminger and P. P. Stenstroom. A novel approach to cache block reuse prediction. In *Proceedings of the 2003 International Conference on Parallel Processing*, Oct. 2003.

[28] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi. Unison cache: A scalable and effective die-stacked dram cache. In *Proceedings of the 47th International Symposium on Microarchitecture*, Dec. 2014.

[29] D. Jevdjic, S. Volos, and B. Falsafi. Die-stacked dram caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache. In *Proceedings of the 40th International Symposium on Computer Architecture*, June 2013.

[30] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian. Chop: Adaptive filter-based dram caching for cmp server platforms.

In *Proceedings of the 16th International Symposium on High Performance Computer Architecture*, Jan. 2010.

[31] T. L. Johnson, D. A. Connors, M. C. Merten, and W. W. Hwu. Run-time cache bypassing. *IEEE Transactions on Computers*, 48(12):1338–1354, Dec. 1999.

[32] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Unsal. Redundant memory mappings for fast access to large memories. In *Proceedings of the 42th International Symposium on Computer Architecture*, June 2015.

[33] M. L. Kersten, S. Idreos, S. Manegold, and E. Liarou. The Researcher's Guide To The Data Deluge: Querying A Scientific Database In Just A Few Seconds. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2011.

[34] T. Kgil, S. D'Souza, A. Saidi, N. Binkert, R. Dreslinski, T. Mudge, S. Reinhardt, and K. Flautner. PicoServer: using 3D stacking technology to enable a compact energy efficient chip multiprocessor. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2006.

[35] S. Khan, Y. Tian, and D. A. Jimenez. Sampling dead block prediction for last-level caches. In *Proceedings of the 43rd International Symposium on Microarchitecture*, Dec. 2010.

[36] M. Kharbutli and Y. Solihin. Counter-based cache replacement algorithms. In *Proceedings of the 23rd International Conference on Computer Design*, Oct. 2005.

[37] H. Kim, P. Ghoshal, B. Grot, P. V. Gratz, and D. A. Jimenez. Reducing network-on-chip energy consumption through spatial locality speculation. In *Proceedings of the 5th International Symposium on Networks-on-Chip*, May 2011.

[38] S. Kumar and C. Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

[39] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon. Amoebacache: Adaptive blocks for eliminating waste in the memory hierarchy. In *Proceedings of the 45th International Symposium on Microarchitecture*, 2012.

[40] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proceedings of the 28th International Symposium on Computer Architecture*, June 2001.

[41] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Lee. A fully associative, tagless dram cache. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 211–222, June 2015.

[42] W.-F. Lin, S. K. Reinhardt, D. Burger, and T. R. Puzak. Filtering superfluous prefetches using density vectors. In *Proceedings of the 19th International Conference on Computer Design*, Sept. 2001.

[43] C. Liu, I. Ganusov, and M. Burtscher. Bridging the processor-memory performance gap with 3D IC technology. *IEEE Design & Test of Computers*, Nov-Dec 2005.

[44] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proceedings of the 41st International Symposium on Microarchitecture*, Dec. 2008.

[45] G. H. Loh. 3d-stacked memory architectures for multi-core processors. In *Proceedings of the 35th International Symposium on Computer Architecture*, June 2008.

[46] G. H. Loh. Extending the effectiveness of 3d-stacked dram caches with an adaptive multi-queue policy. In *Proceedings of the 42nd International Symposium on Microarchitecture*, Dec. 2009.

[47] G. H. Loh and M. D. Hill. Efficiently enabling conventional block sizes for very large die-stacked dram caches. In *Proceedings of the 44th International Symposium on Microarchitecture*, Dec. 2011.

[48] G. H. Loh and M. D. Hill. Supporting very large dram caches with compound access scheduling and missmaps. *IEEE Micro*, 32(3):70 –78, may-jun 2012.

[49] P. Lotfi-Kamran, B. Grot, and B. Falsafi. NOC-Out: microarchitecting a scale-out processor. In *Proceedings of the 45th International Symposium on Microarchitecture*, 2012.

[50] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi. Scale-out processors. In *Proceedings of the 39th International Symposium on Computer Architecture*, June 2012.

[51] N. Madan, L. Zhao, N. Muralimanohar, A. Udipiu, R. Balasubramonian, R. Iyer, S. Makineni, and D. Newell. Optimizing communication and capacity in a 3d stacked reconfigurable cache hierarchy. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture*, feb 2009.

[52] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan. Enabling efficient and scalable hybrid memories using fine-granularity dram cache management. In *Computer Architecture Letters*, Feb. 2012.

[53] Micron's Hybrid Memory Cube Earns High Praise in Next-Generation Supercomputer. http://investors.micron.com/releasedetail.cfm?ReleaseID=805283.

[54] M. Powell, A. Agarwal, T. Vijaykumar, B. Falsafi, and K. Roy. Reducing set-associative cache energy via way-prediction and selective direct-mapping. In *Proceedings of the 34th International Symposium on Microarchitecture*, Dec. 2001.

[55] M. Qureshi and G. H. Loh. Fundamental latency trade-offs in architecting DRAM caches. In *Proceedings of the 45th International Symposium on Microarchitecture*, Dec. 2012.

[56] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th International Symposium on Computer Architecture*, May 2007.

[57] J. A. Rivers, E. S. Tam, G. Tyson, E. S. Davidson, and M. Farrens. Utilizing resue information in data cache management. In *Proceedings of the 12th International Conference on Supercomputing*, 1998.

[58] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, 10(1):16 –19, jan.-jun 2011.

[59] V. Salapura, J. Brunheroto, F. Redigolo, and A. Gara. Exploiting edram bandwidth with data prefetching. In *Proceedings of the International Conference on Computer Design*, 2007.

[60] D. Sanchez and C. Kozyrakis. The zcache: Decoupling ways and associativity. In *Proceedings of the 43th International Symposium on Microarchitecture*, Dec. 2010.

[61] A. Seznec. Decoupled sectored caches: conciliating low tag implementation cost and low miss ratio. In *Proceedings of the 21st International Symposium on Computer Architecture*, Apr. 1994.

[62] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial memory streaming. In *Proceedings of the 33rd International Symposium on Computer Architecture*, June 2006.

[63] The Hybrid Memory Cube. http://www.hybridmemorycube.org.

[64] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *Proceedings of the 28th International Symposium on Microarchitecture*, Dec. 1995.

[65] S. Volos, J. Picorel, B. Grot, and B. Falsafi. BuMP: Bulk memory access prediction and streaming. In *Proceedings of the 47th International Symposium on Microarchitecture*, Dec. 2014.

[66] S. Volos, C. Seiculescu, B. Grot, N. Khosro Pour, B. Falsafi, and G. De Micheli. Ccnoc: Specializing on-chip interconnects for energy efficiency in cache-coherent servers. In *Proceedings of the 6th International Symposium on Networks-on-Chip*, May 2012.

[67] Z. Wang, S. M. Khan, and D. A. Jimenez. Improving writeback efficiency with decoupled last write prediction. In *Proceedings of the 39th International Symposium on Computer Architecture*, June 2012.

[68] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. Simflex: Statistical sampling of computer system simulation. *IEEE Micro*, 26:18–31, July 2006.

[69] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003.

[70] D. H. Yoon, M. K. Jeong, M. Sullivan, and M. Erez. The dynamic granularity memory system. In *Proceedings of the 39th International Symposium on Computer Architecture*, June 2012.

[71] J. Zebchuk, B. Falsafi, and A. Moshovos. Multi-grain coherence directories. In *Proceedings of the 46th International Symposium on Microarchitecture*, Dec. 2013.

[72] L. Zhao, R. Iyer, R. Illikkal, and D. Newell. Exploring dram cache architectures for cmp server platforms. In *Proceedings of the 25th International Conference on Computer Design*, Oct. 2007.

DJORDJE JEVDJIC

Curriculum Vitae

CONTACT INFORMATION

Ecole Polytechnique Fédérale de Lausanne (EPFL)
Parallel Systems Architecture lab (PARSA)
INJ 238, Station 14
1015 Lausanne,
Switzerland

Web: http://parsa.epfl.ch/~jevdjic
E-mail: djordje.jevdjic@epfl.ch
Tel: +41 21 693 13 76

RESEARCH INTERESTS

- **Broad:** Computer architecture and computer systems.
- **Current focus:** Memory systems, system architectures for datacenters, near-memory processing, die-stacked DRAM, hardware specialization/heterogeneity, server design evaluation methodology

EDUCATION

Ecole Polytechnique Fédérale de Lausanne (EPFL) — Lausanne, Switzerland
Ph.D., Computer Science, GPA: 5.87/6.0 — Sep 2015
- Thesis: "DRAM Caches for Servers"
- Advisor: Prof. Babak Falsafi

University of Belgrade — Belgrade, Serbia
M.S., Computer Science, GPA: 10.0/10.0 — Apr 2010

University of Belgrade — Belgrade, Serbia
B.S. (Valedictorian), Electrical and Computer Engineering, GPA: 10.0/10.0 — Sep 2007

HONORS AND AWARDS

- Early Postdoc.Mobility Fellowship by the Swiss National Science Foundation for 2015-2017
- IEEE Micro Top Picks from Computer Architecture Conferences of 2013, "A Case for Specialized Processors for Scale-Out Workloads", 2014.
- Intel Doctoral Student Honor Fellowship for 2013/14.
- Best Paper Award at ASLPOS-XXVII for "Clearing the Clouds: A Study of Emerging Scale-Out Workloads on Modern Hardware", 2012.
- Serbian State Scholarship Foundation Fellowship, 2006-2013.
- Distinguished Student Award by the Serbian Association of Professors and Scientists (2007).
- Award for the Best Student of the Generation, School of Electrical and Computer Engineering, University of Belgrade (2007).

PROFESSIONAL EXPERIENCE

- Research and Teaching Assistant, *EPFL* (2009-Present).
- Teaching Assistant, *University of Belgrade* (2008-2009).
- Data Collection & Labeling Associate, *Microsoft Development Center Serbia* (Winter 2007/08).
- Intern, *Barcelona Computing Center* (Fall 2007).

REFERRED CONFERENCE PUBLICATIONS

- D. Jevdjic, C. Kaynak, G. Loh, and B. Falsafi. Unison Cache: A Scalable and Effective DRAM Cache. In *International Symposium on Microarchitecture (MICRO)*, Dec 2014.
- D. Jevdjic, S. Volos, and B. Falsafi. Die-Stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache. In *International Symposium on Computer Architecture (ISCA)*, Jun 2013.
- P. Tozun, I. Pandis, C. Kaynak, D. Jevdjic, and A. Ailamaki. From A to E: Analyzing TPC's OLTP Benchmarks - the Obsolete, the Ubiquitous, the Unexplored. In *International Conference on Extending Database Technology (EDBT)*, Mar 2013.
- D. Milojevic, S. Idgunji, D. Jevdjic, E. Ozer, P. Lotfi-Kamran, A. Panteli, A. Prodromou, C. Nicopoulos, D. Hardy, B. Falsafi, and Y. Sazeides. Thermal Characterization of Cloud Workloads on a Power-Efficient Server-on-Chip. In *International Conference on Computer Design (ICCD)*, Sep 2012.
- P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer and B. Falsafi. Scale-Out Processors. In *International Symposium on Computer Architecture (ISCA)*, Jun 2012.
- M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. Popescu, A. Ailamaki, and B. Falsafi. Clearing the Clouds: A Study of Emerging Scale-Out Workloads on Modern Hardware. In *International Conference on Architectural Support for Operating Systems and Programming Languages (ASPLOS)*, Mar 2012.

JOURNAL ARTICLES

- M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. Popescu, A. Ailamaki, and B. Falsafi. A Case for Specialized Processors for Scale-Out Workloads. In *IEEE Micro Top Picks,* May/Jun 2014 (original at ISCA 2012).
- M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. Popescu, A. Ailamaki, and B. Falsafi. Quantifying the Mismatch Between Emerging Scale-Out Applications and Modern Processors. In *ACM Transaction on Computer Systems,* Nov 2012.

TALKS

- "Unison Cache: A Scalable and Effective DRAM Cache", MICRO 2014, Cambridge, UK.
- "Die-Stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache", ISCA 2013, Tel Aviv, Israel.
- "DRAM Caches for Servers", EcoCloud Annual Event, May 2013, Lausanne, Switzerland.

TEACHING ASSISTANTSHIPS

- *Principles of Computer Systems* (graduate), EPFL (Fall '14).
- *Introduction to Multicore Architectures* (undergraduate), EPFL (Spring '12).
- *Theoretical Computer Science* (undergraduate), EPFL (Spring '11).
- *Advanced Multicore Architectures* (graduate), EPFL, (Fall '10).
- *Computer Systems Performance Analysis* (undergraduate), Univ. of Belgrade (Spring '08, Spring '09).
- *Algorithms and Data Structures* (undergraduate), Univ. of Belgrade (Spring '08, Spring '09).
- *Internet Application Programming* (undergraduate), Univ. of Belgrade (Spring '08, Fall '08, Spring '09).

- *Processor Microarchitecture* (undergraduate), Univ. of Belgrade (Spring '08, Fall '08, Spring '09).
- *Foundations of Programming* (undergraduate), Univ. of Belgrade (Fall '08).

PROFESSIONAL SERVICE

- External reviewer for: ACM TOCS'15. IEEE TVLSI'15, ISCA'15, HPCA'15, MemForum'14, IISWC'14, ASPLOS'14, HPCA'14, HPCA'13, IISWC'11, DATE'11, ICS'11.
- Co-architect of *CloudSuite*, a benchmark suite for scale-out applications.
- Co-developer of *Flexus,* a full-system multi-processor simulation framework, 2010 - present
- *CloudSuite on Flexus* tutorial (aka *Rigorous and Practical Server Design Evaluation*):
  - EPFL, with C. Kaynak, O. Kocberber, J. Picorel, and S. Volos, Feb 2015,
  - ISCA'13, with A. Daglis and C. Kaynak, Jun 2013.
- ACM SIGARCH and IEEE student member.

MISCELLANEOUS

- Languages: English (professional fluency), French (intermediate), Serbian/Croatian (native)
- Hobbies: music (playing guitar, singing in a jazz choir), sports, philosophy, politics