Performance Enhancement of Tiled Multicore Processors using Prefetching and NoC Packet Compression

Dipika Deb

Performance Enhancement of Tiled Multicore Processors using Prefetching and NoC Packet Compression

Thesis submitted in partial fulfilment
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science and Engineering

by

Dipika Deb

Under the supervision of

Dr. John Jose

Department of Computer Science and Engineering
Indian Institute of Technology Guwahati

May 2022

To my Grandmother

- whose unconditional love, support and blessings helped me to sail through the journey.

Declaration

I hereby certify that

a. The work contained in this thesis is original and has been done by myself and the general supervision of my supervisor.

b. The work has not been submitted to any other Institute for any degree or diploma.

c. Whenever I have used materials (data, theoretical analysis, results) from other sources, I have given due credit by citing them in the text of the thesis and giving their details in the references. Elaborate sentences used verbatim from published work have been clearly identified and quoted.

d. No part of this thesis can be considered plagiarism to the best of my knowledge and understanding and take complete responsibility if any complaint arises.

Date : **17** / **05** / **2022** **Dipika Deb**

Place: Guwahati, India

भारतीय प्रौद्योगिकी संस्थान गुवाहाटी
गुवाहाटी - 781039
Indian Institute of Technology Guwahati
Guwahati - 781039

Dr. John Jose
Associate Professor

: +91 361 258 3256
: +91 361 269 2787
: johnjose@iitg.ac.in

Department of Computer Science and Engineering

Thesis Certificate

This is to certify that the thesis entitled **"Peformance Enhancement of Tiled Multicore Processors using Prefetching and NoC Packet Compression"** being submitted by **Dipika Deb** to the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati, is a record of bonafide research work carried out by her under my supervision and is worthy of consideration for the award of the degree of Doctor of Philosophy of the Institute.

To the best of my knowledge, no part of the work reported in this thesis has been presented for the award of any degree at any other institution.

Date : **17** / **05** / **2022**

Place: Guwahati, India

Dr. John Jose
(Thesis Supervisor)

Acknowledgments

It has been a long journey in IIT Guwahati since my M.Tech days. I met many people throughout the journey and encountered many challenges that helped to shape my personal and professional life. First, I would like to express my sincere gratitude towards my Ph.D. supervisor, Dr. John Jose, for providing me the opportunity to work with him. Throughout the Ph.D. duration, his substantial guidance and support helped to sail this wave smoothly. All the discussions, insightful suggestions, and support throughout the duration cannot be expressed in words. His approach towards his scholars and the discussion on various issues helped to mould my personality and bring the best out of me.

I also thank my Doctoral Committee members, Prof. Jatindra Kumar Deka, Dr. Aryabartta Sahu, and Dr. Tony Jacob, for providing valuable feedback regarding my thesis work. I am also thankful to Prof. Hemangee K. Kapoor. Her expertise and guidance throughout my M.Tech helped me to gain a much deeper understanding on the topic of multicore architecture. I am personally grateful to all the professors of Department of Computer Science and Engineering for their compassionate teaching, and all the staff of the department Raktajit Pathak, Monojit Bhattacharjee, Gauri Deori, Pranjit Talukdar, Nanu Alan Kachari, Bhriguraj Borah, Hemanta Kumar Nath, Nava Kumar Boro, and Prabin Bharali for providing such a wonderful academic environment, necessary help, and computing resources.

I am incredibly thankful to Prof. Maurizio Palesi from the University of Catania, Italy, for his guidance and valuable and timely input on most of my thesis-related contributions. His expertise in the topic of multicore architecture has been invaluable to me. I would also like to mention Prof. Prabhat Mishra from the University of Florida, USA, whom I met for the first time at VLSID 2019 conference in New Delhi, India. He is such zeal of a person. The motivation that he provided on his visit to IIT Guwahati made me think a notch higher in publication aspects. I am sincerely grateful to Prof. Mishra for his valuable suggestions. During my Ph.D. I also got an opportunity to work with Prof. Chun-Lung Hsu, Information and Communications Research Laboratories (ICL), ITRI, Taiwan. Throughout my stay in Taiwan, Prof. Hsu provided invaluable support and ensured that my stay in a foreign land was comfortable. I genuinely thank Prof. Hsu for all the guidance he provided.

No stay is wonderful without a group of friends, and thanking them is just not enough. Being the junior most in the group, I have always been pampered, cared for, and supported by Mayank Agarwal, Shilpa Budhkar, Nayanatara Kotoky, Nilkanta Sahu, Shashi Shekhar Jha, Basant Subba, Hema Kumar Yarnagula, and Sukarn Agarwal. They are not just my seniors or friends; they are my family. I also thank Multi-Core ARchitecture and Systems (MARS) lab members wholeheartedly, with a special mention to Abhijit Das, Sivakumar S, Manjari

Saha, Manju R, Amit Puri, and Tathagata Barik. Four of us (Abhijit, Siva, Manjari, and I) were part of the MARS lab during its inception. We shared many discussions, fun-filled talks, tours, and a fruitful chat regarding work helped me to grow in various aspects. These guys never made me feel that I stayed away from home. I would also like to thank my special friend and sister, Shrutidhara Sarma, who is the constant go-to person through all ups and downs in my life. We were roommates since B.Tech days at Tezpur University, but our bond grew stronger every day. I dedicate this thesis to all of them.

All of this became possible by the love, support, and encouragement from my family. I am indebted to my grandmother Nihar Bala Deb (Daji), my parents Dilip Deb and Dipali Deb, for their constant support to make me what I am today. Daji plays a vital role in my life. With all her sacrifices and immense love, she stood by me for every decision I made. She holds a special place in my heart, and thanking her is just not enough. My parents made countless sacrifices to fulfill my dreams, and I'll forever stay indebted to them. Finally, the immense love, patience, support, and encouragement provided by my soul mate Shirshendu and my wonderful son, Rudra, is beyond imaginable. I am exceedingly thankful to both of them for being a part of my life.

Guwahati, May 2022 **Dipika Deb**

Abstract

The well-known memory wall problem is created because of the disparity between the processor speed and main memory speed, restricting a system from achieving the maximum performance benefit. In a multicore system, the performance is closely linked to how fast a cache miss is served, i.e., Average Memory Access Time (AMAT). However, in Tiled Chip MultiProcessors (TCMP), the inbuilt Network on Chip (NoC) plays an important role in determining AMAT. This is because the last level cache is shared and distributed among the tiles present in the system. In such systems, very often, the role of the underlying communication network gets unnoticed. Thus, cache misses experience additional delay apart from the conventional memory access latencies, which makes the block access time non-uniform. The additional delay is the network latency incurred to transfer the cache miss request and reply packet to the requesting tile. The non-uniform memory access latency across the tiles makes it unpredictable to estimate AMAT. Prefetching and NoC packet compression are the two techniques that can be used to reduce AMAT in TCMP. However, none of the existing techniques considers the on-chip communication overhead of TCMP.

Considering the limitations of existing prefetching techniques, in this thesis, we propose efficient prefetching strategies that are aware of the underlying TCMP architecture. It identifies the false positive cases of prefetching that results in generating useless prefetch requests. These conditions prevail only on TCMP architectures due to its shared and distributed last level cache. The useless prefetch requests, thus generated, causes cache pollution which further results in generating unwanted NoC traffic. It further congests the network, thereby increasing the packet transfer rate in NoC. We notice that useless prefetch requests increases AMAT, hampering the system performance. We also cannot ignore the fact that cache pollution can be caused by useful prefetches by evicting important demand blocks from the cache. Hence, to reduce cache pollution we propose mechanisms for throttling useless prefetches and efficient strategies for placement of prefetch blocks. In order to reduce the on-chip communication latency, a novel packet compression technique is also proposed that operates at a smaller granularity of data to achieve better compression ratio. Experimental analysis shows that the proposed prefetching and compression techniques perform better than the existing techniques. Thus, both the techniques combined reduces the on-chip communication cost that directly improves AMAT for TCMP architectures.

Table of Contents

List of Figures

Page

List of Tables

Page

List of Acronyms

Acronym	Expansion
CMP	Chip MultiProcessor
TCMP	Tiled based Chip MultiProcessor
NoC	Network on Chip
LLC	Last Level Cache
CPI	Cycle Per Instructions
AMAT	Average Memory Access Time
MPKI	Miss Per Kilo Instructions
IPC	Instructions Per Cycles
NUCA	Non Uniform Cache Access
SNUCA	Static Non Uniform Cache Access
DNUCA	Dynamic Non Uniform Cache Access
NST	Network Stall Time
NVP	Near Vicinity Prefetching
ECAP	Energy Efficient Caching of Prefetch Blocks in TCMP
ZPP	Zero Pollution Prefetcher
COPE	Coordinated Prefetching for Efficient resource sharing
WS	Weighted Speedup
UCA	Unified Cache Access
PAggr	Prefetch Aggressiveness
TrAL	Tracker Access Latency
VC	Virtual Channel
OoO	Out of Order

CHAPTER

1

Introduction

The continuous effort towards scaling the transistor technology to lower dimensions enabled the semiconductor industry to densely pack them in a given chip area. Gordon Moore, who predicted that the number of transistors on a chip can be doubled after every eighteen months [1], has made it possible. However, with the same supply voltage, simultaneous switching of these transistors generate an enormous increase in the power density, leading to the *Power Wall Problem* [2]. The increase in power density contributes to the on-chip temperature that further slows down the transistor switching rate, affecting the overall speed of the processing core. Therefore, the paradigm shifted to multicore systems known as Chip MultiProcessor (CMP), where multiple cores run in lower frequencies, providing better parallelism and reduced power consumption with enhanced system performance [3, 4, 5].

Tiled CMP (TCMP) [6, 7, 8, 9] are the next generation CMPs where the cores are organized as multiple tiles connected with an underlying network called Network-on-Chip (NoC) [10, 11]. Figure 1.1 shows a 16 core TCMP where each tile consists of an OoO core, a private L1 cache, and a slice of L2 cache (bank). In this thesis, we use L2 as the Last Level Cache (LLC). The L2 cache is equally shared among the tiles, and a block is mapped to a unique L2 bank known as its home-bank. Within each tile, the core runs an application. For each L1 cache miss that gets transmitted as a request packet through underlying NoC, the requested block is fetched from its home-bank as reply packets. Therefore, the time required to access an L2 block depends on how far its home-bank is from the requesting tile. Also, the speed at which a core is scaling is not proportionate to main memory scaling [12] leading to the famous *Memory Wall Problem* [12]. This restricted

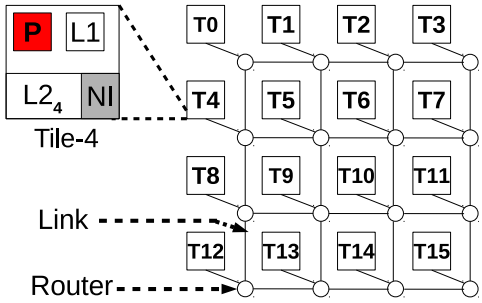

Figure 1.1: An example of 16 cores organized as 4x4 TCMP.

the system performance that is closely linked to how fast a cache miss is served to reduce the Average Memory Access Time (AMAT).

$$AMAT_{TCMP} = Average\ Block\ Access\ Time + Average\ Network\ Stall\ Time \quad (1.1)$$

AMAT in TCMP ($AMAT_{TCMP}$) experiences additional delay apart from the block access time. The additional delay is the communication latency required to fetch the block from another tile. Since the underlying network is responsible for serving the cache miss and the requested block to a tile, the role of NoC is central to the system performance. In this thesis, we redefine AMAT for TCMP architectures using Equation 1.1 where *Average Block Access Time* is calculated conventionally, including the hit time and miss rate in different cache levels and *Average Network Stall Time* (NST) is the time required by the network to serve an L1 cache miss.

In TCMP, AMAT can be reduced by reducing both the parameters in Equation 1.1. The first parameter is directly related with the number of cache hits. Prefetching is used to increase cache hits without any major architectural changes in the cache memory [13]. Also, cache misses can cause an increase in NoC traffic, as well as congestion in the LLC read/write buffer and DRAM bandwidth. As a result, the interference produced by cache misses in TCMP may effect the performance of applications running in other tiles. Hence, one of the primary goals of this thesis is to propose an efficient TCMP-friendly prefetcher to increase the number of cache hits. The second parameter in this equation can be reduced by reducing the number of packets communicating through the NoC. Increase in packet count increases the network load. As a result, packets encounter additional delays in the intermediate routers when compared to a zero load network, thereby increasing NST. Hence, if packet count reduces, NST can be reduced. It can be achieved using a cost effective on-chip packet compression technique. Hence, the second goal of the thesis is to propose an

efficient packet compression technique for TCMP. The amount of on-chip communication in TCMP can also be reduced by increasing the number of L1 hits.

1.1 Prefetching

Prefetching [14, 15] hides the long memory access latency by speculatively fetching blocks to on-chip caches for future use. For this purpose, a prefetch engine continuously monitors the memory access pattern of applications running in the cores. Though predicting simple access patterns is easy, complex access patterns hardly show any repetitive pattern, demanding sophisticated prefetchers. For prefetching to be effective, it requires how accurate its prediction mechanism is (prefetch accuracy), how timely a prefetch block reaches the requesting core (prefetch timeliness) and how much of the cache misses are covered by the prefetcher (prefetch coverage). However, using naive prefetching may degrade the system performance and unnecessarily consume network bandwidth.

Ideally, a prefetcher must completely predict an application's future memory access pattern, which is not possible practically. This results in fetching some useless prefetch blocks along with the useful ones that unnecessarily consume network bandwidth and may replace frequently used demand blocks from the cache, causing cache pollution. Prefetching also increases the frequency at which the processor issues memory requests. Hence, the memory system and the underlying network must match the bandwidth. Since prefetch requests are extra requests that the network handles apart from the demand requests, they may congest the network and increase the packet transfer rate in the network. Thus, achieving the maximum benefit of prefetching requires addressing several challenges; reducing cache pollution, increasing prefetch timeliness, reducing useless prefetch requests, and reducing network traffic caused by prefetching. However, in TCMP, cache prefetching involves new challenges, to be discussed in Section 1.3.

1.2 Packet Compression

In a TCMP architecture, upon encountering a cache miss, applications running on a tile demand an increased on-chip network bandwidth that can cater to intensive tile-to-tile communication to fetch the cache blocks from next levels of memory. Due to the strict area and power budgets, the bandwidth offered by NoC is very limited. Multiple such applications generating significant network traffic for transferring blocks across cache hierarchies can

increase congestion on the shared resources such as LLC and NoC. An increase in network congestion increases the packet transmission latency, thereby increasing AMAT.

Packet compression exploits the data redundancy within network packets, shrinking their size. Reduction in packet size reduces network traffic, wastage of link bandwidth, and dynamic power consumption in NoC. This leads to the reduction in NST and hence AMAT decreases. However, packet compression requires an efficient compressor and decompressor units because the de/compression latency contributes to $AMAT_{TCMP}$. Therefore, on-chip packet compression techniques need to be simple and lightweight. Delta compression [16, 17, 18, 19, 20] is a popular compression technique used due to its simplicity, less hardware overhead, power efficiency, and lower de/compression latency. Existing packet compression techniques [17, 21, 22] are unable to reduce the repetition/redundancies within network packets efficiently that is further discuused in the next section.

1.3 Thesis Motivation

Prefetching is an active area of research for several years [15]. Prefetchers such as stream [23], next line, stride, etc., are simple and cannot predict complex memory access patterns. Recent state-of-the-art prefetchers like PPF [24], and SPP [25] can predict both simple and complex patterns in a non-TCMP architecture (unified LLC). These prefetchers require continuous monitoring and training to make prefetching efficient. The prefetch engine in the state-of-the-art complex prefetchers uses the LLC cache accesses (hits, misses, and block eviction) as feedbacks to train the prefetchers. In TCMP, such feedback is sent from the L2 bank to the tiles as NoC packets. Hence, feedbacks are additional packets routed across the tiles along with the important packets, i.e., cache miss and cache replies. This results in NoC congestion, increasing the average packet latency in the network. Thus, the distributed LLC in TCMP makes the prefetcher training cumbersome.

Experimentally we observed that in a 4x4 TCMP, nearly 33% of L2 blocks are evicted, and 63% hits are experienced in each L2 bank. Similarly, for a 8x8 TCMP, around 47% of L2 blocks are evicted, and nearly 79% of hits are observed in the L2 banks. Thus, there is a significant increase in feedback packets in the network that increase the packet latency for state-of-the-art prefetchers [24, 25, 26, 27, 28]. This explains the unsuitability of recent prefetchers for TCMPs and demands revisiting prefetching for TCMP architectures.

Also, when prefetching is enabled, the network has to route extra packets. Since network bandwidth and NoC power consumption [5, 29, 30, 31] have become a major

concern in designing TCMPs, packet compression [17, 21, 22, 32, 33, 34] provides an alternative to reduce network power and prefetcher-triggered network traffic by exploiting data redundancy within the packets. However, state-of-the-art on-chip packet compression techniques [17, 21, 22, 33] cannot compress packets efficiently using delta compression. Also, the compressor and decompressor units used by these techniques are power-hungry and consume a significant area. This demands revisiting on-chip packet compression for TCMPs to be discussed further in Chapter 7.

1.4 Thesis Contribution

The research challenges, as described in Section 1.3, are dealt with in the rest of the thesis. The first two contribution: *Energy Efficient Caching of Prefetch Blocks (ECAP)* [35, 36], and *Zero Pollution Prefetcher (ZPP)* [37] targets to reduce prefetcher caused cache pollution in TCMPs, in particular, by using efficient prefetch block placement strategies. The third contribution is *Coordinated Prefetching for Efficient resource sharing (COPE)* [38] that identifies useless prefetch requests for TCMPs and throttles them. Finally, the fourth contribution is a novel packet compression technique, *FlitZip* [39] that reduces network traffic such that blocks are transferred faster from their home-bank to the requesting tile. The proposed techniques are implemented in gem5 [40] and are extensively evaluated using real workloads from PARSEC [41] and SPEC CPU 2006 benchmark suite [42]. The hardware overhead of each contribution is also quoted in each chapter. We present a summary of each contribution as follows.

1.4.1 A Dynamic Caching Strategy for Prefetch Blocks in TCMPs

In TCMPs, applications running in each tile have varying memory footprints. Some applications have larger memory footprints (heavy) and some have smaller memory footprints (light). Light applications may waste its cache space while for heavy applications, the limited cache size may be insufficient to hold the working set size. Thus, the heavy applications generate more cache misses than the lighter ones. Also, prefetching in heavy applications may evict useful blocks from the cache, thereby causing cache pollution. Such evicted blocks are requested again, contributing to more cache misses. Since in TCMP, every cache miss generates network packets to fetch the desired cache block, the speculative nature of a prefetcher and an inefficient caching strategy for prefetch blocks can trigger frequent cache block replacements, thereby increasing network load which affects $AMAT_{TCMP}$.

Considering this, ECAP [36] is proposed. It uses a dynamic caching strategy for prefetch blocks where heavy applications can utilize the less used sets of light applications in the neighboring tiles to place its prefetch blocks. Thus, by virtually increasing the cache space for heavy applications, we can reduce cache pollution for such applications. ECAP also uses a novel prefetch-aware replacement policy known as Confidence Aware Replacement Policy (CARP) that helps in caching the prefetch blocks in the neighboring tiles for a longer duration. When compared with baseline (conventional block mapping technique), ECAP reduces AMAT by 19%. Though ECAP reduces cache pollution to some extent, it cannot completely remove prefetcher-caused pollution. Hence, the thesis proposes another technique, *ZPP*, that can eradicate prefetcher-caused cache pollution completely.

1.4.2 ZPP: A Dynamic Technique to Eradicate Cache Pollution

Distributed caches uses coherence protocol to keep blocks coherent across all cache levels. In this thesis, we use the directory-based MESI-CMP protocol [43] where an L1 cache block can be in any of the four stable states: Modified (M), Exclusive (E), Shared(S), and Invalid (I). The M or E states are indicates that the L2 cache may have a stale copy of the block and an updated copy of such blocks is present in some private L1 cache. Any request to such blocks is served from the L1 cache to maintain data coherent across the system. Only the metadata of such blocks is essential to store in LLC. We exploit these LLC blocks to place prefetch blocks using the proposed ZPP [37] technique.

ZPP avoids cache pollution by placing prefetch blocks in the data portion of the stale block location in the LLC bank of local tile (local bank). It keeps the metadata of such blocks intact in local L2 bank. When compared with baseline (BOP [44]), ZPP (best variant) reduces AMAT by nearly 28%.

1.4.3 COPE: Identifying and Throttling Unwanted Prefetches in TCMPs

Existing prefetch techniques extensively use prefetch accuracy parameter to throttle useless prefetch requests [14, 15]. The conventional prefetch accuracy parameter when used in TCMPs may falsely make an inaccurate prefetcher to generate requests which can harm the system performance. Similarly, there may also arise a scenario where an accurate prefetcher is falsely throttled, considering it as inaccurate. These problems are called over-estimation and under-estimation issues in TCMPs. Also, the prefetch accuracy parameter may create a false feedback loop that can mislead a prefetcher in generating more prefetch requests

than necessary. To solve this problem, COPE [38] is proposed.

COPE uses two parameters: Individual Prefetch Accuracy (IPA) and Inter-Tile Communication (ITC) frequency to solve the under-estimation, over-estimation and false feedback loop in TCMP. When compared with baseline (stream prefetcher [45]), COPE reduces AMAT by 24%.

1.4.4 FlitZip: Effective Packet Compression for NoC Packets

Network latency has a major contribution in determining $AMAT_{TCMP}$. Packet compression [17, 21, 22, 32, 33, 34] reduces data redundancy within the packets, thereby shrinking its size. Reduction in packet size reduces network stall time which in turn reduces $AMAT_{TCMP}$. Considering this FlitZip [39] is proposed that can efficiently reduce data redundancies within packets compared to existing techniques [17, 21, 22].

FlitZip searches for data patterns within a flit and upon finding a pattern, each flit is represented as a set of differences, di from an optimal base, B. If the size of di is less than a byte, the flit is compressible. Thus, it performs flit-wise compression, as the name suggests *FlitZip*. When compared with baseline (NoΔ [17]), FlitZip reduces AMAT by 32%.

1.5 Thesis Organization

The chapter wise organization of the thesis is as follows.

Chapter 2 provides a detailed description of the architecture used in the thesis. Then it describes the concept of prefetching and recent prefetching techniques proposed in the literature. It is followed by a summary of various prefetcher proposed in the literature categorized by their type and the underlying architecture used. The chapter also describes the state-of-the-art packet compression techniques for NoC.

Chapter 3 presents the experimental framework followed by the description of simulators, application workloads, benchmarks and evaluation parameters used in the thesis.

Chapter 4 presents ECAP, a novel prefetch block placement strategy to reduce cache pollution in TCMPs. It shows the experimental framework used to place prefetch blocks in less used cache locations to reduce cache pollution. The chapter also presents a confidence-aware replacement policy to make efficient use of the cache space.

Chapter 5 presents ZPP which shows that though the existing prefetching techniques can reduce cache pollution to some extent but none of them can completely remove it. Thus, the chapter proposes an efficient prefetch block placement strategy that can eradicate

prefetcher-caused cache pollution completely from on-chip caches in TCMP architectures.

Chapter 6 shows how the conventional parameters used to throttle useless prefetch requests can mislead a prefetcher in TCMP based architectures in generating further prefetch requests. The chapter proposes COPE which shows that the necessary and sufficient conditions required to throttle prefetch requests in TCMPs are different from the conventional architectures. Hence, the chapter presents a novel technique to identify useless prefetch requests generated in TCMPs and throttle them accordingly.

Chapter 7 discusses the shortcomings of state-of-the-art packet compression techniques in NoC. In the light of the above, it presents a novel packet compression technique, FlitZip that can efficiently reduce the redundancies within packets, thereby obtaining a better compression ratio than the existing techniques.

Chapter 8 finally highlights a summary of all the contributions presented in the thesis and discusses a few future possible research directions.

CHAPTER

2

Background and Literature Survey

CMPs have been dominating the present-day computing world, where multiple cores are packed together on the same chip and are attached with a common shared communication backbone to the main memory. The cores that are employed in CMPs are OoO processors that can execute instructions in parallel, thereby increasing the system throughput. However, with the increase in core count, the bus-based interconnect is not scalable, resulting in the design shift to NoC based TCMP where multiple cores are connected using short wires that are further connected with additional controllers known as routers.

Since the thesis focuses on improving performance aspects in TCMP, from this point onward, we discuss the necessary background required for the reader to understand the subsequent chapters. The chapter is organized as follows. Section 2.1 provides a detailed background on the baseline TCMP architecture followed by a literature survey on existing prefetching and packet compression techniques in Section 2.2 and Section 2.3, respectively. Finally, we conclude the chapter in Section 2.4.

2.1 Baseline TCMP Architecture

In this thesis, we use a TCMP architecture where each tile consists of an OoO core, a private split L1 cache (I cache and D cache), and a slice of L2 cache. We have used L2 (inclusive) as LLC. The LLC is equally shared among the tiles, and a block is mapped to a unique tile using a few MSB bits from the set-index portion (bank-index bits) of its address known as the *home-bank* of the block. Such a distributed cache organization is known as Non Uniform Cache Access (NUCA). If a cache block is statically mapped to its home-bank, the

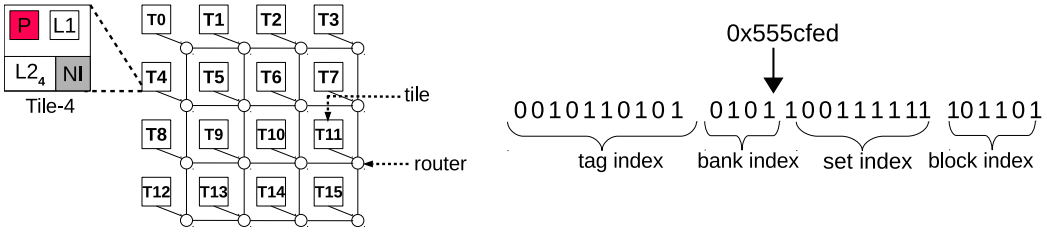

Figure 2.1: An example showing block mapping in 4x4 TCMP.

cache organization is known as Static NUCA (SNUCA) [6, 7]. However, if a cache block is dynamically mapped to a set of L2 banks, the cache architecture is known as Dynamic NUCA (DNUCA) [46]. The tiles are interconnected with each other using NoC. Figure 2.1 shows a 16 core TCMP that is organized as 4x4 mesh. Each tile consists of a 32KB, 4-way set associative L1 cache, and a 4MB, 8-way set associative L2 cache. The block size is 64B for both the caches. Let us assume that an application running in tile 11 encounters L1 cache miss at address 0x555$cfed$ as shown on the right hand side of the figure. From the address, the bank-index bits is determined as $(0101)_2$. Hence, tile 5 is the home-bank of the block. Thus, the L1 cache miss request has to be sent to tile 5 to bring the missed block. We discuss each of the component of tiles in details as follows.

In this thesis, we use TCMP with SNUCA as the baseline architecture and L2 as LLC.

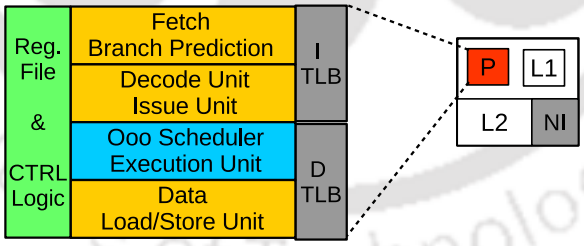

Figure 2.2: Internal organization of processing cores in TCMP.

2.1.1 Processing Cores:

Figure 2.2 shows a processing core that consists of instruction pipeline, functional units, register file, control logic, L1-I cache Translation Lookaside Buffer (I TLB), and L1-D cache TLB (D TLB). The instruction pipeline fetches instructions from I-cache and access respective data from D-cache using I TLB and D TLB. The control logic oversees the

smooth execution of instruction over various pipeline stages consisting of fetch unit, branch predictor, decode unit, issue unit, OoO scheduler, execution unit and load/store unit. D TLB and I TLB translates the virtual address to physical address for program execution.

2.1.2 Cache

Cache [47, 48] is a small and fast on-chip memory made of SRAM cells that takes the advantage of Locality of Reference. The principle of locality of reference statesÂăthat both instructions and data are frequently reused (temporal locality) andÂăthe adjacent locations of a referenced instruction/data have a higher probability of rereferencing soon (spatial locality). As a result, data required by processing cores is fetched on a block-level granularity to take advantage of cache. Caches benefit programmes with predictable and regular memory access patterns. Data-intensive applications or applications that has irregular/complex memory access pattern with large Working Set Size (WSS) does not fit into the cache. This leads to frequent cache misses. On the other hand, large on-chip caches are not a design choice because they increases the cache access time. As a result, the cache size limits the number of blocks that it can store. Despite the introduction of many cache hierarchies (L1, L2, *ldots*, Ln), the limited on-chip area and power budget limit the amount of cache levels that may be added on a die. Modern processors have two or three-level caches. In a two-level cache hierarchy, the L1 cache is private to the core, while the L2 cache serves as the LLC and it is shared by all cores [3, 6]. Similarly, L1 and L2 of the three-level cache hierarchy are private for the core, whereas L3 is LLC.

Figure 2.3 shows the organisation of a two-level cache used in the thesis. Together with a cache block (Data), a few more fields $<$*Valid(V), Dirty(D), Tag, State, LRU*$>$ are stored in L1 cache. Kindly note that in this thesis, the cache is physically tagged and physically index (PIPT) using the tag bits obtained from a block address. The V-bit indicates if the L1 location contains a valid block with the address stored in Tag field. The D-bit specifies whether the block is dirty, and the LRU bits are used to choose a victim during block replacement using the LRU replacement policy [49]. We use the MESI CMP-based directory coherence protocol [50] to keep data consistent across all caches. A block can be in one of the four states: Modified (M), Exclusive (E), Shared (S), or Invalid (I). So, the *State* field can be M/E/S/I. With each block (Data), the L2 cache also contains $<$*V, D, Tag, Sharer, State, LRU*$>$ information. The fields *V, D, Tag, LRU, State*, and *Data* are same to those of the L1 cache. The *Sharer* field keeps track of the L1 caches that have the same copy of

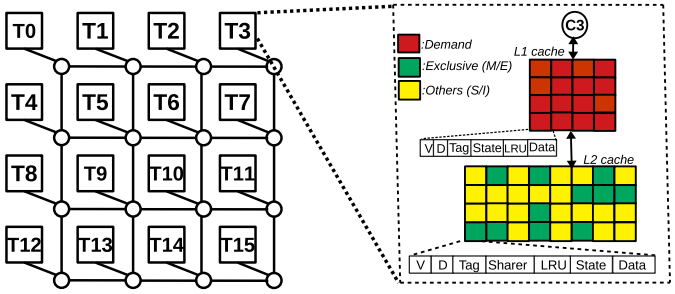

Figure 2.3: Two-level cache organization in TCMP

a block. Hence, it is maintained as a bit-vector with the same number of elements as the system's L1 cache. If a bit is set, a copy of the block is stored in the associated L1 cache. The E and M (exclusive) states of a block indicate that only one L1 cache owns it and has read and write permissions. Because the private L1 cache has an updated copy of the block, L2 blocks are stale. So the exclusive block owner fulfils RD/WR requests. According to the state S, there are many copies of the block with RD-only permission in various L1 caches (S/I are non-exclusive states).

2.1.3 Network on Chip

NoC consists of routers and bi-directional links that communicate the request (cache misses) and reply (data) as packets destined to a tile. A request packet is represented using one head flit and the reply packet is further divided into multiple flits as head, body, and tail; with each flit equivalent to the link bandwidth. The packet's control information is contained in the head flit, and the cache block contents in the body flits and tail flit. Figure 2.4 shows the internal structure of router where each router contains multiple input and output ports through which an incoming and outgoing packet can travel to north, south, east, and west directions. There also exists a local port through which the local tile's request and reply packet travels. Each input port contains multiple buffers known as Virtual Channels (VC) [51, 52], Route Computation (RC) unit, VC Allocator (VA), and Switch Allocator (SA). RC determines the output port that a packet takes in the current router to reach its destination. VA determines the VC number in the downstream router. SA resolves port contention and assigns output port through which flits travel out of a router and link carries flit from one router to another.

The three main aspects of NoC are topology, routing, and flow control. Topology is the arrangement of tiles in the network, routing determines the path through which a packet

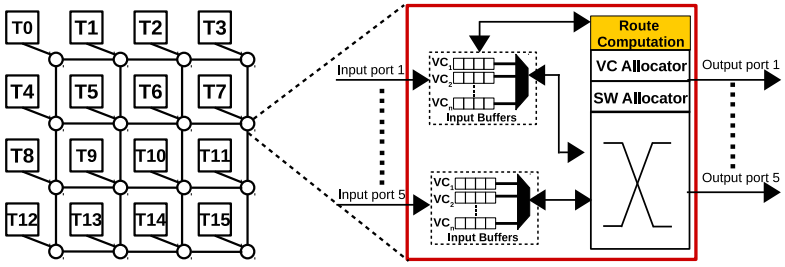

Figure 2.4: Internal structure of router.

travels between a pair of tiles, and flow control is a mechanism through which a packet allocates resources in the network while traveling from the source tile to a destination tile. To make routing efficient, routing algorithms determine the shortest path between a pair of tiles. Various types of routing algorithms are used, such as deterministic, adaptive, store and forward, wormhole, and cut-through. Out of these categories, we use a deterministic XY routing where a packet first aligns itself with the destination tile in X-direction and then traverses along the Y-direction to reach its destination. We also use a wormhole routing with virtual cut-through where the head flit reserves a VC at each router, and all the flits use the same path as the head flit.

2.1.4 Communication in TCMP

The tiles in TCMP are interconnected with NoC using a Network Interface (NI). NI consists of an inject queue and eject queue used to transfer cache blocks to and from the processor. Whenever the processor requests a block, it is first searched in L1 cache. If is present in L1 cache, it is termed as cache hit; otherwise, a miss (demand miss). Upon encountering an L1 cache miss, the request is sent to L2 cache. Since L2 cache is shared and distributed among the tiles, an L1 cache miss is sent as a request packet to the home-bank of the block (which can be a distant tile) through NI. For this purpose, the request packet is queued in the inject queue. Upon finding a free VC, the request packet is injected into the network that travels to its home-bank. The home-bank forwards the requested cache block as reply packet (divided into multiple body flits) destined to the requesting tile. On reaching the requesting tile, the reply packet is queued at the eject queue in NI, which is placed in L1 cache for subsequent processing. Therefore, the access time of a cache block depends on the round trip time of the cache miss request and the reply packets.

The time required by the request and reply packet to travel between the tiles is known as network stall time (NST). NST is heavily dependent upon the underlying network's condition.

If the network is congested, it may delay the packet transfer rate, thereby increasing NST. Since $AMAT_{TCMP}$ is dependent upon NST, it makes the cache access time non-uniform. The non-uniform cache access time makes the system unpredictable to estimate the CPU execution time, thereby hampering system performance. Prefetching and on-chip packet compression are the two techniques that are employed to improve $AMAT_{TCMP}$. In the Equation 1.1, prefetching reduces the block access time and NST by fetching cache blocks ahead of its use. However, there are hardly any prefetchers that take into account the underlying communication overhead of NoC. Thus, in this thesis, we propose an efficient TCMP-friendly prefetcher to improve $AMAT_{TCMP}$. Also, packet compression helps in fetching cache blocks in a cost-effective way by reducing its size. Reducing packet size reduces network traffic, thereby improving average NST. Therefore, the thesis also aims to propose an efficient on-chip packet compression technique to improve $AMAT_{TCMP}$. The next two sections describe state-of-the-art prefetching and packet compression techniques.

2.2 Prefetching

Prefetching predicts the future blocks that might be used by an application and avoids cache misses by fetching such from main memory to on-chip caches before they are actually referenced. It is first used commercially in Intel 8086 microprocessors [53]. Since then almost all high performance processors uses several prefetching techniques at different cache levels [9, 14, 15, 54, 55, 56, 57]. In this section, we present a discussion on various prefetching techniques that have been proposed in the literature. It also provides some insights into the unsuitability of the existing techniques for TCMP architectures in Section 2.2.3.

In this thesis, we focus on data prefetching only.

Prefetching can be done in hardware [25, 24, 58, 59, 60, 61, 62, 63, 64, 65, 66] as well as in software [67, 68]. Software prefetching requires inserting *fetch* instructions in the program either by a programmer or a compiler before run time. The fetch instructions are placed within loops used for large calculations, as loops have predictable array referencing patterns. Software prefetching cannot prefetch block dynamically at run-time. Hardware prefetching requires a prefetch engine placed on-chip and observes the memory access pattern of applications running in the cores. Thus, it proceeds parallel to the processor computation.

Upon observing a pattern, the hardware prefetcher dynamically issues prefetch requests for the predicted blocks at run time.

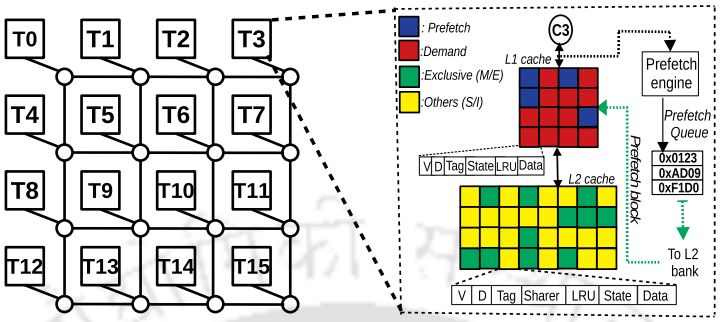

Figure 2.5: Prefetch enabled cache organization.

The memory access pattern observed by a hardware prefetcher can be simple or complex. Simple data patterns are easy to predict, while complex data patterns are irregular and do not follow any patterns. Such access patterns can be further categorized as temporal and spatial [15]. Temporal patterns have jumbled memory patterns where a few memory requests reoccur within a short time interval. Spatial patterns have a correlation across an address. Such patterns need not reoccur again. Upon predicting a pattern, the prefetcher generates requests to fetch the predicted blocks to be referenced in the future. Such blocks, upon reaching the requested tile, are placed in the conventional location. For example, an L1 prefetcher prefetches blocks at the L1 cache. Figure 2.5 shows the contents of L1 cache when an L1 prefetcher is enabled in a TCMP system. As shown in the figure, each tile has an L1 prefetcher (prefetch engine) that generates request queued at the prefetch queue. Upon finding an empty MSHR entry, requests are issued from the prefetch queue which are sent to its respective home-bank (L2 bank) through NoC (shown as green dotted lines in the figure). A prefetch block upon reaching the requesting tile is placed in the L1 cache. Thus, the L1 cache contains two types of blocks: Demand blocks and Prefetch blocks. Demand blocks are fetched when the processor explicitly requests them, and prefetch blocks are fetched far ahead of their use.

Some of the parameters that quantify a prefetcher are distance, degree, aggressiveness, coverage, accuracy, timeliness, and cache pollution. Prefetch distance refers to how far ahead from the miss stream a prefetcher should start fetching cache blocks, and prefetch aggressiveness decides the number of requests a prefetcher should generate at once. Prefetch coverage is defined as the number of cache miss that a prefetcher covers. Prefetch accuracy is calculated as the ratio of useful prefetch blocks to the total number of prefetch blocks.

Prefetching also demands extra care such that the prefetched data is fetched neither too early nor too late. It is known as prefetch timeliness [14, 15]. If a prefetch block is fetched too early, there may arise two problems: (a) The prefetch block may evict a useful or frequently used block from the cache. The evicted block may be re-requested by the application, which results in fetching the block again. Thus, prefetching may cause *cache pollution* which is a serious concern in design a better prefetcher [14, 15], and (b) The prefetch block might get replaced by another block before it is referenced. If the prefetch block is fetched too late, the processor may stall, waiting for the block to be fetched. Hence, prefetch timeliness is also an essential parameter for designing a prefetcher.

2.2.1 Hardware Data Prefetcher

The simplest form of hardware prefetcher is the sequential prefetcher or Next-Line Prefetcher (NLP) [15]. In NLP, when a miss occurs for cache block B, along with the block B, block B+1 is also prefetched. Stream prefetcher [23, 45, 69], on the other hand, upon encountering a cache miss prefetches the next K sequential cache blocks and stores them in a stream buffer (FIFO) instead of cache to avoid cache pollution. Whenever a cache miss results in a stream buffer hit, the prefetch block is placed in the L1 cache. As prefetches are used from the buffer, new prefetches are brought in to keep the buffer full. When the required block is not present in the buffer, the entire buffer is flushed. Another technique, called as stride prefetching [14] observes the common stride in the memory accesses. Suppose, an observed memory access pattern is *A, A+n, A+ 2n, A+ 3n,* then n is the stride and a prefetch is issued for (A+4n), and so on. Thus, stride prefetcher brings cache blocks from a particular pre-defined prefetch distance with an appropriate prefetch aggressiveness.

Recent prefetchers such as Best Offset Prefetcher (BOP) [44] uses a best-offset learning algorithm where the miss addresses are being checked for 52 different offset list. Due to the learning stage, the time taken by BOP to decide which block to prefetch is longer but accurate. Also, the misses observed during the longer learning stage in BOP are demand misses and not covered by the prefetcher. This results in less prefetch coverage. Access Map Pattern Matching (AMPM) [70] uses the conventional prefetch accuracy parameter and BOP [44] uses an approximation algorithm to throttle prefetches. AMPM uses an indirect method to calculate prefetch accuracy where the access pattern map is decoupled from the cache hierarchy and core. Based on the conventional prefetch accuracy calculation, the prefetch degree is adjusted. Since the conventional method produces false positive cases of

prefetching [38], AMPM fails to deliver the performance in a TCMP architecture.

There are recent lookahead prefetchers [25, 24, 26, 27, 28] that observes changes in block addresses called delta between the cache lines and stores them along with the block address as signatures in a table. The table is later used for prefetching blocks. VLDP uses multiple prediction tables to predict future deltas. For prefetching, a VLDP module per core is used. Each module consists of a Data History Buffer (DHB) per page, a Data Prediction Table (DPT), and an Offset Prediction Table (OPT). DHB tracks the delta history of recently accessed pages and is used by DPT to predict future memory requests. OPT's are used if there are fewer accesses to a page. Multiple DPTs are used to predict future memory requests based on delta history and accuracy. After learning some access patterns on a page, VLDP applies the same patterns to other pages using a Global Prediction Table.

Signature Path Prefetcher (SPP) [25] predicts both sequential and complex memory access patterns across pages with reasonable confidence. It uses two tables: Signature Table (ST), and Pattern Table (PT), and a prefetch engine. ST stores 12-bit signatures by hashing previous deltas in the page, and the last block accessed is used to update the signature. PT is used to predict the next delta to be fetched and maps signature with prediction using a probability of correctness. If the probability crosses a certain threshold, the engine issues prefetch requests, and it is also used as a lookahead candidate to the prefetch engine. The prefetch engine recursively predicts future blocks and multiplies probabilities to get confidence value, determining whether to prefetch a block or throttle it.

Perceptron-based Prefetch Filtering (PPF) [24] introduces an additional module that can be used with preexisting prefetching hardware to increase the coverage while maintaining its accuracy. It uses a perceptron model and consists of a list of weight tables for each feature, a prefetch table and a reject table. The features are selected using Pearson's correlation factor and cross-correlation. The perceptron model operates in three steps: Inferencing, Recording, and Retrieving. A prefetch request is generated based on the weights of various perceptrons in the inference stage. If the summation of all perceptron weights is above a threshold, the prefetch request is generated otherwise, rejected. In recording stage, the generated and rejected prefetch requests are recorded in a Prefetch Table and Reject Table, respectively. In the retrieving stage, perceptrons are continuously trained by updating the associated weights. The weights are updated according to the perceptron learning rule by communicating every LLC access and block eviction to the prefetch engines.

Kill-the-PC cache management policy (KPC) [26] works on the fact that existing

replacement policies and prefetching algorithms, when used together, negatively impact performance since they do not work well together. KPC has two parts: a prefetching component, KCP-P, and a replacement policy component, KPC-R. They share information to make performance optimal. KPC-P is similar to SPP, except it maintains additional two-bit vectors for cache lines in a page to monitor which cache blocks have been prefetched and referenced to avoid redundancy. KPC-R uses a table that randomly selects 64 cache blocks from LLC and monitors hits, misses, and L2 prefetches at the LLC level. The hits and misses are used to update the hysteresis counters (2 per core), which are used to determine the LRU priority of future fetched blocks.

Bouquet of Instruction Pointers (BIP) [27] uses a bouquet of multiple IP-based prefetchers: stride prefetcher, complex stride prefetcher, which is based on SPP with a fixed prefetch degree, Global Stream Prefetcher, and the Next line prefetcher. A single combined table is used for all these classes that are used to issue prefetches based on L1 cache accesses. The prefetch metadata is then stored into another table accessed with every L2 cache access so that these prefetches can be reused. Compared to others, the performance improvement of BIP is substantial because it uses a diverse group of prefetchers with different advantages.

Multi-Lookahead Offset Prefetcher (MLOP) [71] is an offset prefetcher similar to BOP. It can prefetch many blocks, which are expected to be seen far into the future. It consists of an Access Map Table (AMT) that is indexed by block addresses, and each entry contains a bit vector for offsets of blocks in that address range. Each entry also includes a list of last accessed block addresses and stores the offset's prediction scores. At each lookahead level, the score gets updated on each cache access. When prefetching, lower lookahead values are given priority, and the highest scored offset is the block prefetched at each lookahead level.

BINGO [72] is a spatial prefetcher that records the block accesses to a page using a bit vector called the footprint. When a block in the page is evicted, the page's footprint is stored in a history table that contains multiple entries (like set-associative cache). Future references to a new page refer to the history table to retrieve the block access pattern and prefetch blocks from the footprint. Events are used to make prefetch decisions and index them into the table. BINGO uses two events: PC+Address (Long) and PC+Offset (short and contained in the former). Long event predictions are accurate with a low probability of occurrence, while short event predictions are not accurate but have high coverage.

2.2.2 Restricted Data Prefetcher

Researchers have also controlled the negative aspects of prefetching [73, 74, 75, 76, 77, 78, 79, 80, 81, 24, 82, 83, 84, 85, 86, 87] like cache replacement, cache pollution, lack of prefetch accuracy and timeliness. Some of these restricts the generation of prefetch requests while some applies different cache management and NoC management techniques to place prefetch blocks efficiently. We name the former category of prefetchers as *Prefetch Request Throttling techniques* while the later category as *Prefetch Block Management techniques*.

A. Prefetch Block Management (PBM)

Some of the prefetching technique uses different cache management policies to handle an incoming prefetch block to avoid cache pollution. Seshadri et al. proposed ICP [83] which experimentally find that the reuse of a prefetch block is very low. Hence, such blocks are evicted quickly by changing the underlying replacement algorithm's insertion, promotion, and demotion policy. ICP also uses an evicted address filter, EAF, which reduces prefetcher caused cache pollution by monitoring the prefetch accuracy. However, ICP does not consider the throttling of inaccurate prefetches. PACMan [87] also targets to reduce prefetch induced cache pollution by changing the insertion, demotion and promotion policy for various replacement techniques. The replacement policy for inaccurate prefetches is dynamically controlled at run-time to avoid the thrashing of useful cache blocks.

There exists a few packet management techniques to control the negative aspects of prefetching in NoC. Ebrahimi et al. [88] addresses the problem of shared resources contention which occurs on prefetching. Since the underlying NoC is designed for demand packets only, prefetching might degrade the performance and fairness as well. Thus, three different shared resource management schemes are used: network fair queuing, parallelism aware batch scheduling and fairness via source throttling, for managing prefetch packets in NoC.

Albericio et al. [89] proposed an adaptive controller, ABS to prefetch blocks in LLC. The controller controls the number of misses generated from each L2 bank when the prefetch aggressiveness varies. Since increase in miss count reduces system performance, ABS dynamically assigns different aggressiveness value to each L2 bank. It uses a hill climbing approach to control the aggressiveness of each prefetchers. Andre et al. [90] proposed a balanced prefetch controller to prefetch blocks in L1 cache. The controller reduces miss penalty by controlling prefetch aggressiveness in TCMPs. The author uses a cache sniffer and a directory sniffer at each tile. The directory sniffer samples out misses to predict the

next cache request. The cache sniffer estimates the delay involved in fetching a block from lower level of memory to L1 cache. Both together controls the aggressiveness of prefetchers by reducing network delays. Hence, miss penalties in a core reduces. The paper also emphasizes in reducing miss penalty by placing prefetch blocks closer to the source core.

B. Prefetch Request Throttling (PRT)

In PRT techniques, prefetch requests are regulated in the prefetch engine, and unwanted prefetches are dropped by not inserting them into the prefetch queue. Srinath et al. [74] proposed FDP that throttles a prefetcher by changing its prefetch distance and prefetch aggressiveness by periodically monitoring prefetch accuracy (conventional), lateness, and cache pollution. Nachiappan et al. [91] introduces application aware prefetch prioritization techniques to remove the negative effects of prefetching in the network. In the paper, the author ranks prefetchers from different applications based on their prefetch utility for the applications. Since prefetchers causes pollution in the network as well as in the cache [92], a useless prefetcher rather than improving performance rather degrades it severely. The paper introduces a prioritization mechanism that differentiates between prefetches from different applications and prioritizes prefetches from those applications which are useful.

SPP [25] uses confidence parameters to throttle prefetches that are too far from the memory access path. KPC [26] uses two parts: KPC-P and KPC-R to control the prefetch blocks. KPC-P decides the cache level where a prefetch block is to be placed and uses the conventional prefetch accuracy to throttle inaccurate prefetches. KPC-R selects an appropriate replacement policy for the prefetch block placed in the cache. Both PBM and PRT techniques can independently work together in a system as in KPC. PPF uses SPP as the underlying prefetcher, but it replaces the existing confidence-based prefetch throttling with a perceptron based filtering mechanism.

Table 2.1 categorizes various prefetching techniques according to the architecture used, i.e, TCMP or non-TCMP architecture. In the table, the prefetchers are also grouped into their types such as simple prefetcher (SimP), Irregular prefetch (IRP), Server prefetching (ServP), Software prefetcher (SofP), GPU prefetching (GP), Correlated prefetching (CP), Thread-aware prefetching (ThP), Temporal prefetcher (TP), Spatial prefetcher (SP), Delta prefetcher (DP), and controlling negative aspects of prefetching (NP).

Table 2.1: State-of-the-art prefetching techniques.

Prefetcher	Type	Sub Category	Architecture
Jouppi et al. 1990 - [13]	SimP	–	non-TCMP
Palacharia et al. 1994 - [45]	SimP	–	non-TCMP
Zhang et al. 2000 - [23]	SimP	–	non-TCMP
Sherwood et al. 2000 - [93]	SimP	–	non-TCMP
Sorin et al. 2004 - [69]	SimP	–	non-TCMP
Huang et al. 2012 - [94]	NP	–	non-TCMP
Cavus et al. 2020 - [95]	IRP	–	non-TCMP
Naithani et al. 2020 - [96]	IRP	–	non-TCMP
Jain et al. 2013 - [97]	IRP	CP	non-TCMP
Shevgoor et al. 2015 - [28]	IRP	TP, DP –	non-TCMP
Bakhshalipour et al. 2017 - [60]	IRP	TP	non-TCMP
Bakhshalipour et al. 2018 - [98]	IRP	TP	non-TCMP
Bakhshalipour et al. 2019 - [99]	ServP	–	non-TCMP
Bakhshalipour et al. 2019 - [72]	IRP	SP	non-TCMP
Lee et al. 2014 - [94]	NP	–	non-TCMP
Nachiappan et al. 2012 - [91]	NP	–	non-TCMP
Ebrahimi et al. 2011 - [88]	NP	–	non-TCMP
Ozturk et al. 2008 -[67]	SofP	–	non-TCMP
Neves et al. 2021 - [100]	SimP	SofP	non-TCMP
Chen et al. 2010 - [59]	ServP	–	non-TCMP
Chen et al. 1995 - [58]	SimP	–	non-TCMP
Heirman et al. 2018 - [80]	NP	–	non-TCMP
Crago et al. 2018 - [101]	GP	–	non-TCMP
Liao et al. 2020 - [63]	IRP	SP	non-TCMP
Xiangyao et al. 2015 - [64]	IRP	SP	non-TCMP
Golshan et al. 2020 - [61]	IRP	TP	non-TCMP
Zhang et al 2020 - [62]	IRP	TP	non-TCMP
Wenisch et al. 2009 - [102]	IRP	TP	non-TCMP
Bhatia et al. 2019 - [24]	NP	–	non-TCMP
Pakalapati et al. 2020 - [27]	IRP	SP	non-TCMP
Michaud et al. 2016 - [44]	IRP	SP	non-TCMP
Yasuo et al. 2009 - [70]	IRP	SP	non-TCMP
Jorge et al. 2012 - [89]	NP	–	TCMP
Lu et al. 2020 -[103]	NP	–	non-TCMP
AlBarakhat et al. 2020 - [104]	ThP	–	non-TCMP
AlBarakhat et al. 2018 - [105]	ThP	–	non-TCMP
Somogyi et al. 2006 - [106]	IRP	SP	non-TCMP

Prefetcher	Type	Sub Category	Architecture
Kim et al. 2016 - [25]	IRP	DP, TP	non-TCMP
Kim et al. 2017 - [26]	IRP	DP, TP	non-TCMP
Lai et al. 2001 - [107]	CP	–	non-TCMP
Varkey et al. 2017 - [108]	IRP	TP	non-TCMP
Ainsworth et al. 2019 - [68]	SofP	–	non-TCMP
Kindguli et al. 2018 - [109]	SimP + IRP	–	non-TCMP
Lee et al. 2010 - [54]	GP	–	non-TCMP
Xiang et al. 2015 - [110]	GP	–	non-TCMP
Zhuang et al. 2007 - [111]	NP	–	non-TCMP
Selfa et al. 2018 - [112]	NP	–	non-TCMP
Pugsley et al. 2014 - [113]	SimP+IRP	–	non-TCMP
Dahlegren et al. 1995 - [114]	SimP + IRP	–	non-TCMP
Liao et al. 2009 - [82]	NP	–	non-TCMP
Torrents et al. 2016 [115]	SimP+IRP	–	TCMP
Huang et al. 2012 - [94]	NP	–	non-TCMP
Wu et al. 2016 - [116]	GP	–	non-TCMP
Chen et al. 2015 - [117]	GP	–	non-TCMP
Jerger et al. 2006 - [118]	NP	–	non-TCMP
Mehran et al. 2019 - [71]	SimP+IRP	–	non-TCMP
Srinath et al. 2007 - [74]	NP	–	non-TCMP
Ebrahimi et al. 2009 - [76]	NP	–	non-TCMP
Muralidhara et al. 2011 - [77]	NP	–	non-TCMP
Ebrahimi et al. 2011 - [78]	NP	–	non-TCMP
Yu et al. 2014 - [79]	NP + ThP	–	non-TCMP
Yedlapalli et al. 2013 - [81]	NP	–	non-TCMP
Sheshadri et al. 2015 - [83]	NP	–	non-TCMP
Sheshadri et al. 2012 - [84]	NP	–	non-TCMP
Victor et al. 2012 - [85]	NP	–	non-TCMP
Jimenez et al. 2015 - [86]	NP	–	non-TCMP
Navarro et al. 2020 - [119]	NP	–	non-TCMP
Wu et al. 2011 - [87]	NP	–	non-TCMP
Cireno et al. 2016 - [120]	IRP	TP	TCMP

TP: Temporal Prefetcher; **SP:** Spatial Prefetcher; **GP:** Prefetcher for GPU, **NP:** Controlling negative aspects of prefetching; **DP:** Delta Prefetcher; **SimP:** Simple prefetcher, **SofP:** Software Prefetcher, **IRP:** Irregular Prefetcher. **CP:** Correlated Prefetching. **ThP:** Thread aware Prefetching. **ServP:** Prefetcher for server

2.2.3 Unsuitability of Existing Prefetchers for TCMPs

As shown in Table 2.1, most of the existing prefetchers use non-TCMP as the underlying architecture. Hence, these prefetchers does not consider the impact of NoC while prefetching cache blocks. Recent state-of-the-art prefetchers like PPF, SPP, KPC, VLDP, etc. can predict both simple and complex patterns in a non-TCMP architecture having unified LLC. These techniques also adopt an inbuilt prefetch filter to throttle useless prefetch requests. PPF uses a perceptron model to filter the useless prefetches and provides 16.9% and 11.3% speedup over 4-core and 8-core, respectively, compared to a system with prefetching disabled. But the behavior of PPF drastically changes when modeled on an 8x8 TCMP.

Because perceptron learning requires continuous monitoring and training, PPF trains the prefetcher with a prefetch table and a reject table. The prefetch table keeps track of requested prefetch addresses, while the reject table keeps track of useless prefetches. Whenever LLC evicts a prefetch block from LLC, it is removed from the prefetch table, and the reject table records the addresses. Similarly, a demand hit in the reject table means the block was misclassified as useless and the address is recorded in the prefetch table. As a result, every LLC bank access updates the weights associated with each perceptron (hits, misses, and block eviction). NoC packets carry such inputs across the tiles as feedback packets. These packets increase NoC traffic, slowing down the packet transmission rate. Similarly, L2 stats are also used to train SPP and KPC's prefetch filters. An LLC hit for an address that the prefetcher has previously retrieved increments the useful counter, but an LLC miss for an address that the prefetcher has previously fetched indicates that the prefetch block is late. As a result, the distributed LLC in TCMP makes perceptron learning cumbersome.

Experimentally we observed that in 4x4 TCMP, 33.23% of LLC blocks are evicted, and 63% LLC hits are experienced in each bank. On the other hand, for a 8x8 TCMP, 46.8% of LLC blocks are evicted, and 78.8% of LLC hits are observed in L2 cache. Figure 2.6 shows the network packet latency observed in PARSEC benchmarks [41] where the y-axis corresponds to the average number of cycles a packet requires to travel from source tile to a destination tile. From the figure, we can observe that there is a significant increase in packet latency for PPF and SPP. The average packet latency for PPF and SPP is 65 cycles and 63 cycles, respectively whereas for stream prefetcher is less than 20 cycles. We have also calculated the contribution of different packets in PPF, as shown in Figure 2.7. We can observe that out of the total NoC packets, the majority are feedback packets. A

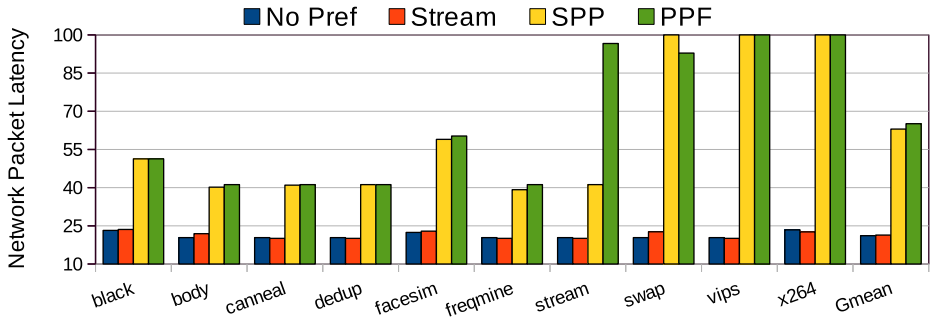

Figure 2.6: Comparison of network packet latency in different prefetchers.

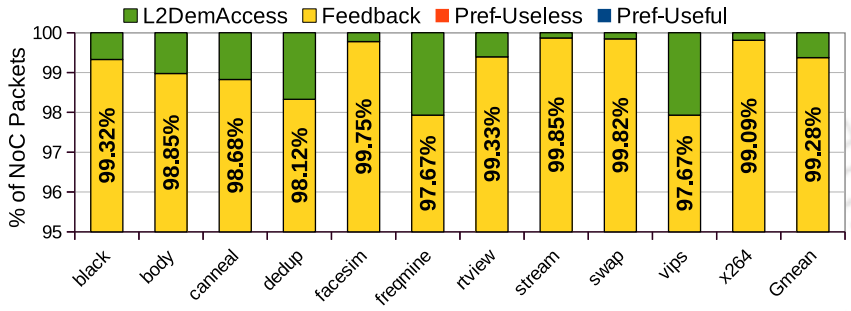

Figure 2.7: Distribution of NoC packets (in %) in PPF.

similar scenario is also obtained for SPP and KPC. We have also plotted packet latency for stream prefetcher and a system with no prefetching (No Prefetcher). The difference between PPF, SPP, and stream prefetcher is that in SPP and PPF, the average packet latency is contributed by useless prefetches (Pref-Useless), useful prefetch (Pref-Useful), demand packets (L2DemAccess), and the extra feedback (Feedback) packets. While in stream prefetcher, NoC consists of Pref-Useful, Pref-Useless and L2DemAccess packets only. In a system with no prefetching, the NoC packet consists of L2DemAccess only. The absence of feedback packets in other prefetchers makes the average packet latency of such prefetchers minimal compared to SPP and PPF. This shows the unsuitability of recent prefetchers for TCMPs and demands revisiting prefetching for TCMP architectures.

2.3 Packet Compression

Even though prefetching and compression at cache memory level are explored from early 90's, optimizations for TCMP designs emerged very recently. There are a very few works explored in on-chip packet compression in NoC. In this section, we discuss briefly on most of the relevant state-of-the-art packet compression techniques in NoC. On-chip packet compression

techniques can be lossless [17, 21, 22, 34, 33, 33, 32] or lossy (approximation techniques) [121, 122, 123]. Lossy technique aims to attain a higher compression ratio without the need for exactness. Thus, packets before compression and packets after decompression may not be the same. Such techniques are generally faster than lossless techniques and are used in error-tolerant applications like image processing, voice recognition, etc. On the other hand, the lossless technique aims for exactness. Such techniques are used for applications where data loss may change the application execution path. Thus, packets received after decompression are the same as that of the original packet. Error-tolerant applications can also apply lossless compression.

Unused Significant Bit Removal (USBR) [33] was the first to recognize the need for packet compression in NoC to reduce NoC congestion and packet loss. It primarily aims at datasets whose variance is low such that the significant bit changes less frequently and hence becomes a use case to reduce network packets. Kim et al. [124, 125, 126] uses the concept of spatial locality speculation to compress network packet to reduce NoC energy consumption. It uses the concept that words that the processor uses are only required to be updated in different cache levels. Thus, the technique transmits only those flits containing words that are marked used by a statistically generated Used Vector, thereby reducing the amount of data sent across the NoC.

Zhan et al. [17] proposed NoΔ that uses a delta compression technique to compress network packets. It uses the fact that the relative deviation between the data values varies a little in a packet. Hence, a part of the packet (X bytes) can be represented as a common base of B bytes and rest of the packet is represented as an array of differences: $\{\Delta_1, \Delta_2, \Delta_3, \ldots, \Delta_n\}$ where $n = X/B$, from the base. It uses multiple de/compression modules that execute all combinations of (Base, Δ) to achieve a better packet compression ratio. If multiple combinations of base and delta are found to be suitable for packet compression, NoΔ uses the one that can achieve the highest compression ratio.

Das et al. [32] proposed ZeroCompr that reduces packet size by encoding consecutive zeroes in the head flit. Frequent Pattern Compression (FPC) [34] uses an encoding table to compress an incoming packet against the patterns stored in the table. The table contains eight frequently occurring patterns, and it is shared among all the NIs. Wang et al. proposed DISCO router [21, 22], where an incoming packet is selectively compressed at the router using delta compression [18, 17]. The selection criteria are based on the idle time of a packet stored in the internal buffers of a router. The selected packets then undergo compression

using delta compression (BDI or NoΔ) or FPC.

Boyapati et al. introduced a lossy compression, Approx-NoC [121] that approximates frequently occurring data values to reduce the network load for error-tolerant applications. Chen et al. [122] also proposed an approximate communication framework that reduces packet size by approximating data values within network packets. Raparti et al. proposed DAPPER [127] that uses approximate-computing for GPUGPU architectures. It trades-off computation accuracy for energy savings and uses an overlay circuit prepared dynamically between Memory Controllers (MCs) for a time window. Chen et al. proposed DEC-NoC [123] that considers that the NoC is prone to error. Hence, packets are retransmitted to mitigate the errors. To reduce power consumption, the author reduces the amount of error correction and checking during packet transmission. However, the de/compressor units used by the existing techniques consumes tremendous area and power. Also, these techniques cannot efficiently reduce packet redundancies which can be exploited to compress packets further. Moreover the existing compression techniques operates at a larger granularity of data at the base level and delta level that limits in achieving better compression ratio. This demands revisiting on-chip packet compression for NoC based TCMPs.

2.4 Chapter Summary

In this chapter, we described the baseline TCMP architecture, followed by a detailed discussion on state-of-the-art prefetching techniques. We have observed that most of the prefetchers are designed for non-TCMP architectures. Thus, the existing techniques avoids the underlying network stall time involved while fetching a cache block from a different tile. The chapter also provides some profound insights into the recent works in on-chip packet compression techniques. We observed that the existing on-chip packet compression techniques cannot compress network packets efficiently. This demands revisiting both prefetching and on-chip packet compression technique for TCMP architectures.

CHAPTER

3

Experimental Framework

Modeling a proposed architecture in real hardware is highly expensive. Also, prototyping all the techniques in FPGAs with various configurations is time-consuming and not budget-friendly. Since architectural simulators can mimic a real system's behavior, computer architects model and design different techniques in these simulators to explore the performance of the proposed architectures. Also, the tools used to design real hardware are costly, and it is not feasible to model and experiment the proposed architectures with various configurations such as different cache sizes, branch predictors, number of cores, etc. Therefore, computer architects use various architectural simulators for obtaining different performance, timing, power ,and area related metrics such as CPI, cache hit time, area overhead, etc.

The chapter discusses some important experimental tools that have been used for the various experimental purpose in the thesis. The chapter is organized as follows. Section 3.1 discusses various types of simulators followed by the benchmarks used in Section 3.2. The experimental details are discussed in Section 3.3. Section 3.4 contains various performance metrics used in this thesis and we finally conclude the chapter in Section 3.5.

3.1 Simulators

Simulators are software-defined models written in different programming languages in order to build a prototype of any proposed architecture in software. Since the simulators are programs, they allow incorporating changes in the code as per the requirement. gem5 [40], Orion 2.0 [128], Cacti [129], BookSim [130] are architectural simulators to name a few. These simulators run on real machines (host machines) and create a prototype of the proposed

architecture, known as the target machines. BookSim is trace-based NoC simulator which is used for simulating NoC prototypes. Cacti is a cache modeller, which is used to calculate the access latency and area/power consumption of if different cache configurations. The values of Cacti are normally used in other simulators as a parameters. Orion 2.0 is a simulator used for calculating the power consumption of the system during the execution. gem5 is a full-system simulator which is used to simulate the complete computer system. Since most of the experiments of this thesis use gem5, a detailed discussion about this simulation tool is given next in this chapter.

A full-system simulator can simulate a complete system, including devices, CPU, memory, interconnection network, operating system, etc. There exist two types of full system simulators: timing and functional. Timing simulators model an actual system and take the real-time behavior of the system into account. On the other hand, functional simulators only focus on implementing the actual system without considering the timing aspects. Thus, a timing simulator can mimic the behavior of an actual system where tasks such as cache miss, instruction execution, etc., are triggered in a timely manner. Since the thesis requires measuring $AMAT_{TCMP}$, we use a full-system timing simulator, gem5 [40] to model each contribution.

3.1.1 gem5

gem5 is a full-system and event-driven timing simulator. It is known as full-system simulator as it can simulate a complete system including processing cores, controllers, memory systems, I/O devices, interconnection network, etc. gem5 is also called an event-driven simulator because all the actions in the system are represented as events that are timestamped. The events are stored in an event queue and a time is attached with each event. As per the specified time, an event gets scheduled. gem5 can also operate on SysCall Emulation (SE) mode. In SE mode, it emulates most of the system level services (kernel codes) and executes the user code only. It cannot model any devices or OS. gem5 also records various parameters such as number of caches misses at various cache levels, execution time, number of prefetch requests issues, prefetch hits obtained, average packet latency, number of flits generated, number of instructions executed, cycles per instruction (CPI), etc. that is used to evaluate the performance of a technique. To test the performance of each contributions on the target machine, real benchmarks [41, 131] are used.

gem5 is an amalgamation of m5 [132] and GEMS [133]. m5 is built by the University of

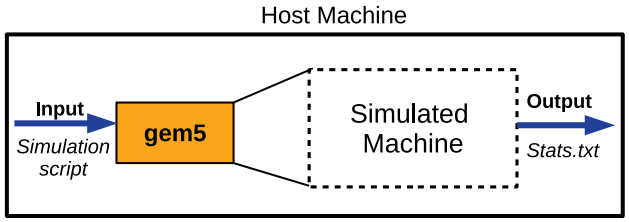

Figure 3.1: Overview of gem5.

Michigan, while GEMS is built by the University of Wisconsin. m5 supports multiple CPU models and Instruction Set Architectures (ISAs) such as ALPHA, ARM, x86, MIPS, SPARC and PowerPC. In contrast, GEMS provides a detailed and flexible memory system (Ruby) and supports various coherence protocols [50] with a flexible interconnection network connecting all on-chip modules (Garnet) [134]. Both m5 and GEMS are combined to form a flexible system that helps computer architects to simulate a target machine according to their needs. Figure 3.1 shows how gem5 builds a simulated machine with the machine configuration as input (Simulation script). The simulation script is written in python programming language and it contains the machine configuration such as cache size, associativity, main memory size, number of cores, etc. The two most well-known simulation scripts are *se.py* and *fs.py*. gem5 takes the simulation script as input and models a target (simulated) machine. The behaviour of the simulated machine is written in C++ and the performance of the simulated machine can be observed in an output file (*stats.txt*). The simulation script in gem5 has two phases: Configuration and Simulation. The configuration phase contains the machine specification while the simulation phase models the defined machine in the host system. Basically gem5 provides an environment which can simulate different types of systems, designed by using the simulation scripts.

Figure 3.2 shows the interaction of CPU and memory in gem5. As shown in the figure, the CPU models are designed with plug-in-capability with different ISAs and memory systems. gem5 supports three In-order CPU models: *AtomicSimpleCPU, TimingSimpleCPU (timing)*, and *InOrderCPU (MinorCPU)* and one Out of Order model, known as *O3CPU* or *DerivO3CPU* or *detailed*. It also provides two memory modules: *Classic* and *Ruby*. Classic memory model is provided by m5, whereas GEMS provides the Ruby memory model. gem5 also has two interconnection networks: simple and garnet. As shown in the figure, CPU and memory in gem5 interacts with each other as master and slave using m5 ports where the CPU acts as a master and memory is the slave. However, m5 has only point-to-point bus based interconnection and uses snooping coherence protocol that is not

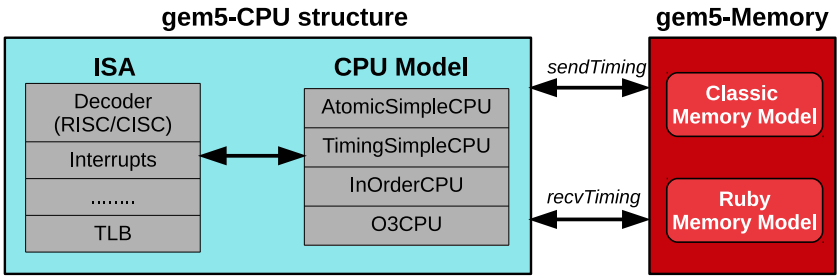

Figure 3.2: CPU and memory interaction in gem5.

scalable as the number of cores on-chip increases. Hence, garnet [134] provides a flexible interconnect that can connect multiple cores on-chip together to form a modern CMP.

Since we use a Tiled CMP with SNUCA as the underlying cache architecture, Ruby in gem5 supports such an architecture. It can be modeled using various parameters such as cache size, main memory size, associativity, number of cache banks, cache access latency, CPU type, coherence protocol, etc. Along with multiple cache levels, ruby can also simulate main memory and directories. Since LLC is shared and distributed among the tiles, various coherence protocols are implemented using a domain-specific language called SLICC, to design the coherence protocols. However, gem5 can simulate a maximum of 64 core CMP. But garnet standalone can simulate up to 256 nodes in NoC.

3.1.2 BookSim

For the first contribution, i.e., ECAP, gem5 is integrated with BookSim2.0 [130] to model NoC. BookSim is a cycle-accurate trace-based NoC simulator used for simulating NoC prototypes. It is written in C++. It allows modeling a wide range of NoC topologies, such as 2D mesh, Torus, butterfly, etc., routing algorithms, flow control mechanism, allocators, etc., in them. BookSim also supports various routers such as input queued router, event router, and chaos router. The simulator takes synthetic traffic patterns or traces from real benchmark suites as inputs and models the interconnection network. It enables evaluating an extensive range of performance parameters to analyze the impact of NoC. It provides various NoC related performance metrics such as average packet latency, execution time, average number of flits per cycle, etc.

3.1.3 Orion

The thesis also uses Orion 2.0 [128] which is an accurate NoC power and area modeling tool for *800nm, 400nm, 350nm, 250nm, 180nm, 110nm, 90nm, 65nm, 45nm,* and *32nm* transistor technologies. It also offers various design models for routers and links, including input/output buffers, arbiters, and crossbar switches. Orion is added to existing full system simulators to calculate the power and area aspects of the complete network.

3.1.4 Cacti

Cacti [129] model cache memory in order to calculate the access latency and area/power consumption for various cache configurations. The values of cacti are normally used in other simulators as input parameters. It takes various parameters as inputs such as SRAM technology, cache size and associativity, number of cache levels, cache access technique, i.e., UCA/NUCA, etc., to simulate a cache. Cacti uses various power and performance modes for simulating a cache, such as High Performance cells (HP), Low STandby Power cell (LSTP), and Low Operating Power cells (LOP). Among these types, we use the HP cells to model cache. It also supports various cache access techniques such as fast, sequential, and normal. In this thesis, we use fast access mode as in this mode, both the data and tag arrays are accessed simultaneously while accessing cache blocks.

3.1.5 Hardware Analysis Tool

Each of the thesis contributions is implemented in Verilog HDL and synthesized the designs at various transistor technologies using Synopsys DC. The design has been successfully tested with custom test bench for behavioral, post synthesis functional, and timing simulations to verify its integrity as in hardware. For obtaining the hardware overhead, all the contributions are evaluated on Xilinx Vivado 2016.2 that is implemented on ZedBoard Zynq-7000 ARM/FPGA SoC Development Board.

3.2 Application Workloads

Workloads are a set of benchmarks that run together in multiple cores to evaluate the performance of simulated architectures. These are classified into two categories: Multithreaded and MultiProgrammed. Multithreaded workloads spawn multiple threads of the same benchmark across all the cores, and the threads share data among themselves. On the

Table 3.1: PARSEC benchmark suite.

Benchmark name	Workload Domain	Prefetch Friendly	Working Set Size	Data Sharing
blackscholes (black)	Financial analysis	Low	Small	Low
bodytrack (body)	Computer vision	Low	Medium	High
canneal	Engineering	High	Large	High
dedup	Enterprise storage	High	Large	High
facesim	Animation	Medium	Medium	Low
ferret	Similarity Search	Medium	High	High
fluidanimate (fluid)	Animation	Medium	High	High
freqmine	Data mining	High	Large	High
raytrace	Rendering	Low	Medium	Medium
streamcluster (stream)	Data mining	Medium	Medium	Low
swaptions (swap)	Financial analysis	Low	Medium	Low
vips	Media processing	High	Medium	Low
x264	Media processing	High	Medium	High

other hand, in multiprogram workloads, a single benchmark runs in a single core. Thus, multithreaded benchmarks obtain a higher degree of parallelism as compared to multithreaded ones. Princeton Application Repository for Shared-Memory Computers (PARSEC) [41] contains multithreaded benchmarks and Standard Performance Evaluation Corporation (SPEC) CPU 2006 [131] contains multiprogram benchmarks. Since the experimental setup differs for each contribution, each chapter has a description of the benchmark used.

3.2.1 PARSEC

It is a widely accepted benchmark suite used by academia and industry to evaluate the performance of multicore architecture. PARSEC consist of a set of multithreaded benchmarks where the actual multithreading occurs at a particular position called "Region of Interest (ROI)". In the thesis, we use the standard *sim-medium* files as input to these benchmarks, and all the statistics are collected from the ROI region. In this thesis, we use benchmarks from both PARSEC 2.0 and PARSEC 3.0 suites. PARSEC 2.0 has thirteen workloads from various areas such as simulated annealing, multimedia, animation, computer games, etc. The description of workloads from the PARSEC suite is summarized in Table 3.1. The benchmarks are categorized as high/medium/low in terms of prefetch friendly by comparing the execution time of a system enabled with stream prefetcher with a system that has no prefetching. A benchmark is said to be highly prefetch friendly (high) if the reduction in execution time for the benchmark is more than 10%, medium if the reduction is in between $(2\% - 10\%)$, and low if the reduction is less than 2%. Data sharing and working set size are the standard characteristics of these benchmarks [41].

Table 3.2: SPEC CPU 2006 benchmark suite.

	Workload Domain	Programming Language
	—CINT Benchmarks—	
perlbench	Programming Language	C
bzip2	Compression	C
gcc	C Compiler	C
hmmer	Search Gene Sequence	C
sjeng	Artificial Intelligence: Chess	C
libquantum (lib)	Physics / Quantum Computing	C
h264ref	Video Compression	C
omnetpp	Discrete Event Simulation	C++
xalancbmk	XML Processing	C++
	—CFP Benchmarks—	
bwaves	Fluid Dynamics	Fortran
gamess	Quantum Chemistry	Fortran
milc	Physics/Quantum Chromodynamics	C
gromacs	Biochemistry / Molecular Dynamics	C, Fortran
cactus	Physics / General Relativity	C, Fortran
leslie	Fluid Dynamics	Fortran
namd	Biology / Molecular Dynamics	C++
calculix	Structural Mechanics	C, Fortran
GemsFDTD	Computational Electromagnetics	Fortran
lbm	fluid dynamics	C
wrf	weather	C, Fortran
sphinx	speech recognition	C

3.2.2 SPEC CPU 2006

In order to evaluate the performance of the proposed techniques for multiprogram workloads, benchmarks from SPEC CPU 2006 suite [131] is used. It contains a set of CPU-intensive benchmark suites that are categorized into two groups: CINT and CFP. Both CINT and CFP consist of compute-intensive integer and compute-intensive floating point benchmarks. Similar to the PARSEC benchmark suite, the benchmarks from SPEC suites belong to real-work applications. Table 3.2 summarizes the workloads that are used in the thesis from SPEC CPU 2006 benchmark suite. In this thesis, we use 64 core system modeled as 8x8 TCMP. The benchmarks mentioned in the table are used to form workload mixes that are mapped to 64 cores in 24 different ways as shown in Figure 3.3. For example, in the first mapping, the first benchmark is mapped to core $[0-15]$, the second benchmark is mapped to core $[16-31]$, the third benchmark is mapped from core $[32-47]$, and the fourth benchmark is mapped from core $[48-63]$. Hence, a total of 384 different workloads for 64-core and 128 for 16-core are created to evaluate the effectiveness of different techniques.

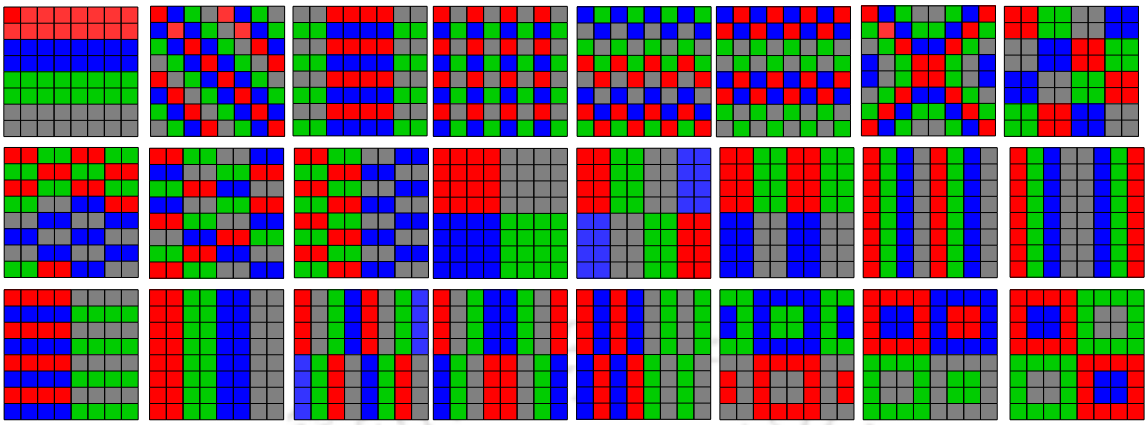

Figure 3.3: Mapping of SPEC benchmarks to a core in 8x8 TCMP.

Since different benchmark has different properties, mapping them to a multicore system in various ways helps in improving the experimental coverage. It also accounts for a wide range of application behavior.

3.3 Experimental Setup

Table 3.3 describes the simulation parameters used to model each contribution of this thesis, in gem5. The thesis models an SNUCA based TCMP with two levels of cache hierarchy, and uses directory-based MESI CMP coherence protocol. The core can dynamically schedule 128 instructions enabled with tournament branch predictor. The L2 cache is shared and distributed as set-interleaved banks among all the tiles, and the L1 caches (L1I-cache and L1D-cache) are private to each core. The NoC is clocked at 1GHz clock frequency and using garnet 2.0, we have built 8×8 and 4×4 2D mesh topology. It consists of a 3-stage pipelined router that has five input-output ports. Packet processing at each router requires two cycles, and the link requires one cycle to transfer a packet from one router to the adjacent router. Since the NoC link bandwidth is 128 bits, a $64B$ block is divided into four reply packets and one head flit.

3.4 Various Performance Related Parameters

The most commonly used performance metrics for experimental analysis are as follows.

- **Weighted Speedup (WS) [135]**: Since we use a multicore system, weighted speedup gives a better view of the throughput achieved by the system. It is calculated using

Table 3.3: Simulation parameters used for various experimental analysis in the thesis. *May vary according to the best experimental setup.

Name	Specifications
Processor:	16/64, x86 cores OoO superscalar
Processor frequency:	3 GHz
NoC frequency:	1 GHz
Branch predictor:	Tournament*
L1 cache per tile:	32KB*, 2-way associative, 64B block
L2 cache per tile:	256KB*, 8-way associative, 64B block
L1 and L2 MSHR size:	128 each
L1 cache access time:	2 cycles
L2 cache access time:	12* cycles
Cache Type:	PIPT
Replacement Policy (all caches):	Least Recently Used (LRU)
DRAM configuration:	DDR3, 4GB, 64-bits channel, 2 ranks per channel, and 8 banks per rank
Prefetcher:	stream prefetcher*, aggressiveness = 2
Coherence:	MESI CMP directory protocol
NoC topology:	8×8 2D mesh with XY routing, 1 cycle link delay,and 2 cycles router delay
NoC link bandwidth:	128 bits
Packet size:	1-flit request and 4-flit reply
VC buffer size:	4
Buffer Organization	Virtual Cut Through

Equation 3.1 where IPC_i^{shared} is the Instructions Per Cycle (IPC) of core i shared with other (P-1) applications running on a MPSoC of P cores and IPC_i^{alone} is the IPC of core i running alone in a MPSoC of P cores (Higher is better).

$$Weighted\ Speedup = \sum_{i=0}^{T-1} \frac{IPC_i^{shared}}{IPC_i^{alone}} \tag{3.1}$$

- **Flit Compression Ratio (FCR)**: FCR is measured using Equation 3.2 where $FC_{compressed}$ is the number of flits obtained after applying the compression technique and $FC_{uncompressed}$ is the number of flits generated when no compression is applied. Therefore, the value of FCR always lies between 0 and 1 (Closer to 1 is better).

$$Flit\ Compression\ Ratio = \frac{FC_{uncompressed} - FC_{compressed}}{FC_{uncompressed}} \tag{3.2}$$

- **Bandwidth Utilization (BU)**: BU is the average number of flits passing through a link per cycle (Lower is better).
- **Prefetch Accuracy (PA)**: PA is measured as the ratio of useful prefetches to the ratio of total prefetches generated as shown in Equation 3.3 (Higher is better).

$$Prefetch\ Accuracy = \frac{Useful\ Prefetch}{Total\ Prefetch} \tag{3.3}$$

- **Prefetch Coverage (PCov)**: PCov is defined as the number of misses that are covered by the prefetcher. It is calculated using Equation 3.4 where $PrefetchHits(i)$ is the number of cache hits obtained by the prefetcher in tile i and $DemandMiss(i)$ is the number of misses observed in the L1 cache of tile i (Higher is better).

$$Prefetch\ Coverage = \frac{PrefetchHits(i)}{PrefetchHits(i) + DemandMiss(i)} \tag{3.4}$$

- **Prefetch Lateness (PL)**: If a demand miss occurs for an outstanding prefetch request, it indicates a late prefetch block. Hence, PL is measured as the ratio of late prefetch blocks to the useful prefetch blocks in each tile (Lower is better).
- **Cache Pollution**: It is measured using bloom filter [136] where it stores the block information that got replaced by a prefetch block. When the processor requests for a block and its a hit in bloom filter, is an indication of cache pollution. Thus, the number of such instances are counted to calculate cache pollution (Lower is better).
- **Packet Queuing Latency (PQL)**: It is the average amount of time when the flits of a packet are queued in the internal buffers of a router, i.e., VC during its transmission from a source tile to a destination tile (Lower is better).

Among these parameters, WS is used by all the contributions to evaluate the throughput achieved by them. PA, PCov, PL and cache pollution is used by ECAP, ZPP and COPE, to measure the effectiveness of the underlying prefetcher. On the other hand, FlitZip uses FCR, BU, and PQL to measure how well it compresses NoC packets.

3.5 Chapter Summary

This chapter described architectural simulators, followed by a description on the real benchmarks used to evaluate the performance of a system. The chapter broadly discussed the experimental setup used by the contributions made towards the thesis and explains the commonly used performance metrics to evaluate their performance.

CHAPTER

4

A Dynamic Caching Strategy for Prefetch Blocks in TCMPs

The chapter proposes a novel prefetch block placement technique for applications, whose working set size exceeds the cache size. For such applications, an inefficient prefetch block placement strategy may cause more harm than benefit to the application by causing cache pollution, thereby increasing AMAT for the application. When compared with a conventional TCMP architecture, the proposed technique significantly reduces cache pollution that improves system performance.

4.1 Introduction

In TCMPs, each tile runs an application with varying memory footprints. Some applications with larger memory footprints use most of their cache space, while some applications with smaller memory footprints may waste their cache space. The former is called as heavy applications, while the latter is known as light applications. For heavy applications, either the cache size is insufficient to hold the working set of the application, or the memory access pattern of the application may not exhibit any locality of reference. In both these cases, the number of cache misses for heavy applications is comparatively higher than that of the light applications. Since in TCMP, cache misses are served by sending request and reply packets across the tiles, heavy applications tend to generate more NoC packets than the light ones. As increasing the cache size is not a viable solution for reducing cache misses, prefetching may help in such a case. However, for heavy applications, prefetching may cause thrashing by evicting useful blocks from the cache, thereby demanding frequent cache block replacements. Such evicted blocks are requested again, generating more cache misses,

contributing to the network traffic. Thus, the speculative nature of a prefetcher and an inefficient caching strategy for prefetch blocks can pollute the cache, which can trigger frequent cache block replacements [15].

In TCMP, frequent cache block replacement generated due to prefetcher-caused cache pollution increases network packets, which in turn contributes to increased NoC power consumption. It may also dampen the packet movement rate, increasing the miss penalty at the cores that further contributed to $AMAT_{TCMP}$. Therefore, if prefetch blocks avoid replacing the demand blocks from a cache, a substantial amount of performance gains can be achieved. This chapter proposes Energy Efficient Caching of Prefetch Blocks (ECAP), where prefetch blocks of heavy applications are placed at the underutilized L1 cache sets of adjacent tiles running light applications. This virtually increases the cache size of heavy applications at run time. Also, prefetch blocks avoid replacing the useful blocks from a cache, effectively reducing cache pollution in TCMPs. On experiencing a hit in the adjacent tile, the prefetch blocks are brought into the local cache, thereby avoiding a longer memory access latency to bring the block from a distant tile. ECAP also uses a novel prefetch-aware replacement policy, CARP that helps in identifying deadblocks in a cache. By removing deadblocks, ECAP helps in caching such prefetch blocks for a longer duration, improving the system performance. Table 4.1 defines some important terms used throughout the chapter.

Table 4.1: Definition of some important terms used in the chapter.

Terms	**Definition**
Target-set	conventional set in the cache where a block is mapped.
Home cache	When a block b is requested by a core in a tile, the private L1 cache of the tile is called as the home cache for the block, b.
Remote cache	For a block b, the neighboring L1 caches of its home cache (one-hop neighbors) are considered as remote cache.

The key contributions of the chapter are as follows:

1. We identify the limitations of the conventional block placement technique in TCMPs that causes severe prefetch-caused cache pollution for some applications.
2. We propose an efficient prefetch block placement strategy where the less used cache block locations of neighboring tiles are used to avoid cache pollution.
3. We also propose a novel prefetch-aware replacement policy for efficient handling of prefetch blocks.
4. We validate our claims by experimenting on real workloads from SPEC CPU 2006 benchmark mixes.

The rest of the chapter is organized as follows. The next section describes the motivation behind ECAP. Section 4.3 describes the proposed technique followed by experimental analysis in Section 4.4 and Section 4.5 finally summarizes the chapter.

4.2 Motivation

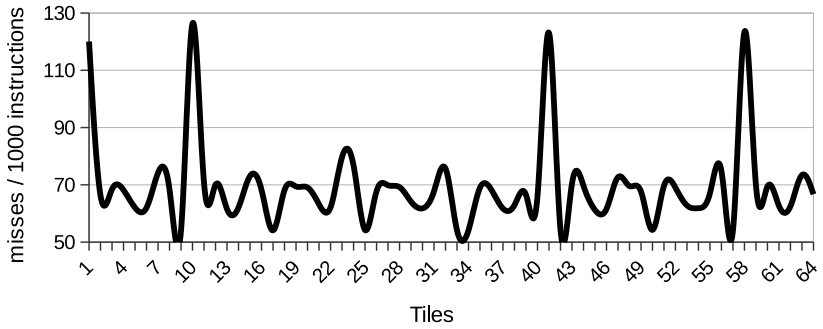

Figure 4.1: Distribution of L1 cache miss.

Figure 4.1 shows the tile-wise distribution of L1 cache miss in a 64-core TCMP enabled with next line prefetcher [14]. In the figure, we can observe that the L1 cache miss varies significantly across the cores. The experiment is performed on a 64 core TCMP with the simulation parameters as mentioned in Section 4.4. Since cache usage is dependent on the application running in the core, low misses for tile 2, 6, 14, 16, 23 indicates that the application is either a light application or the L1 cache size is sufficient to handle the working set of the application. A higher number of misses for tiles 1, 11, 42, 59 indicates a heavy application running in the core. For such applications, the working set size may be larger than the cache size. Hence, the requested cache blocks cannot be fully accommodated in their L1 cache. This results in frequent cache block replacement, which increases NoC traffic, thereby affecting the overall system performance. On the other hand, the available L1 cache space may remain under-utilized for light applications, resulting in cache space wastage.

We have also analysed the set access behavior of L1 cache in a light application as shown in Figure 4.2. We collected the statistics for a time window of 10K cycles after performing sufficient fast-forwarding to eliminate compulsory misses. We can observe that the cache access pattern is non-uniformly distributed across the cache sets. Some sets are highly used, while some of them are less used. Thus, we can conclude that some applications under-utilize their cache space. The neighboring cores running heavy applications can use few cache sets of such applications for which either the cache is of insufficient size or has

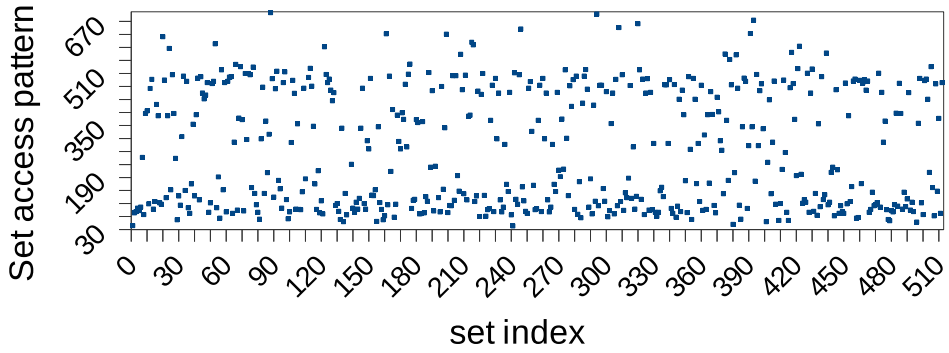

Figure 4.2: Set access behavior in L1 cache. X-axis shows the set number and Y-axis shows the number of accesses observed within the set.

higher miss rates. Hence, a heavy application can use the less used cache sets of other cores to accommodate its prefetch blocks temporarily. This motivated us to propose ECAP that increases the cache size of heavy applications as per its need.

4.3 Proposed Technique

ECAP consists of a prefetch block placement strategy known as Near Vicinity Placement (NVP), and a prefetch-aware block replacement policy, Confidence-Aware Replacement Policy (CARP). Section 4.3.1 and Section 4.3.2 explain NVP and CARP, respectively.

4.3.1 Near Vicinity Placement

Figure 4.3(a) shows a 4x4 TCMP where tile 6 is running a heavy application and the adjacent tiles, 2, 5, 7, and 10 runs light applications. The detailed architectural description about TCMP is given in Chapter 2. ECAP initially analyzes the opportunity of storing a prefetch block at its home cache in tile 6. If the home cache's target-set is heavily used, ECAP searches for a suitable, less used set in the remote caches that runs a light application. Subsection 4.3.1 explains the searching procedure in detail. Upon finding a less used target-set, the prefetch block is stored in the suitable remote cache, and some metadata is stored in the home cache containing the block's information. If no such target-set is found in the remote caches, ECAP uses a special buffer called Prefetch Buffer Pool (PBP) in each local tile to accommodate a few such blocks. Hence, a prefetch block at tile 6 can be placed in one of the neighboring tiles, increasing the cache size of tile 6, virtually.

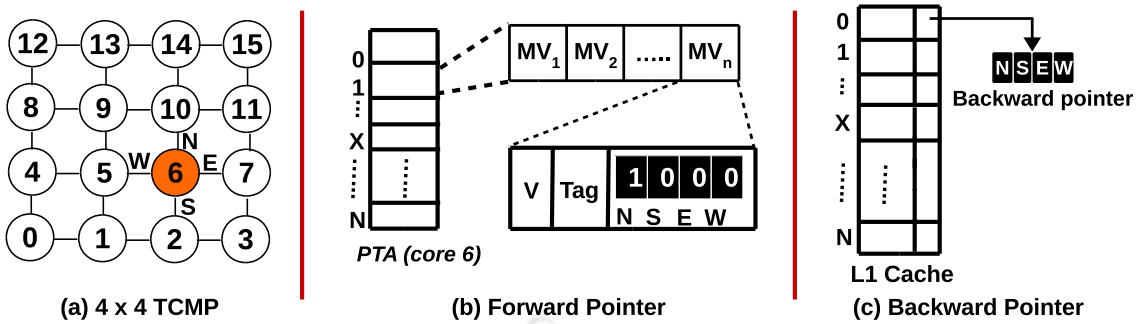

Figure 4.3: Additional hardware used for block placement and identification in ECAP.

In this chapter, Adjacent and Neighbor are used interchangeably to indicate one-hop distant tiles.

The chapter uses next line prefetching as baseline. As already discussed in the previous chapter, the next line prefetcher on a cache miss for block B initiates a prefetch request for the next sequential block $B+1$, if it is not already present in the cache. We call this technique as Source Core Prefetching (SCP) throughout the chapter. Figure 4.4 shows an ECAP enabled cache where the L1 cache may contain four different types of blocks. As the name suggests, Invalid blocks (I) are invalid in the cache. Demand blocks (D) are fetched on core's demand, while the prefetch blocks that are placed in the local tile are known as local prefetch blocks (P). On the other hand, satellite blocks (S) belong to an adjacent tile that runs a heavy application and are placed in the cache (remote cache). To reduce the prefetch block access time, ECAP limits remote caches to one hop distance from the home cache. On a hit in the remote cache, the prefetch block is placed in its home cache, and the corresponding metadata is updated accordingly. Kindly note that SCP and ECAP are not prefetchers. Both of them uses different caching strategy for prefetch block placement (remote or home). Hence, ECAP can be used as an add-on with existing prefetchers [14].

A. Searching for Target-set in Neighboring Tiles

Whenever a prefetch request is issued and the block's target-set is highly used, ECAP adopts a simple neighbor searching scheme where the source tile probes its one-hop neighbors for prefetch block placement at the remote caches. For this purpose, ECAP uses NVP and CARP. NVP deals with the selection of a target-set in the remote cache. It uses the frequency of set access patterns in the remote caches and classifies them as either highly

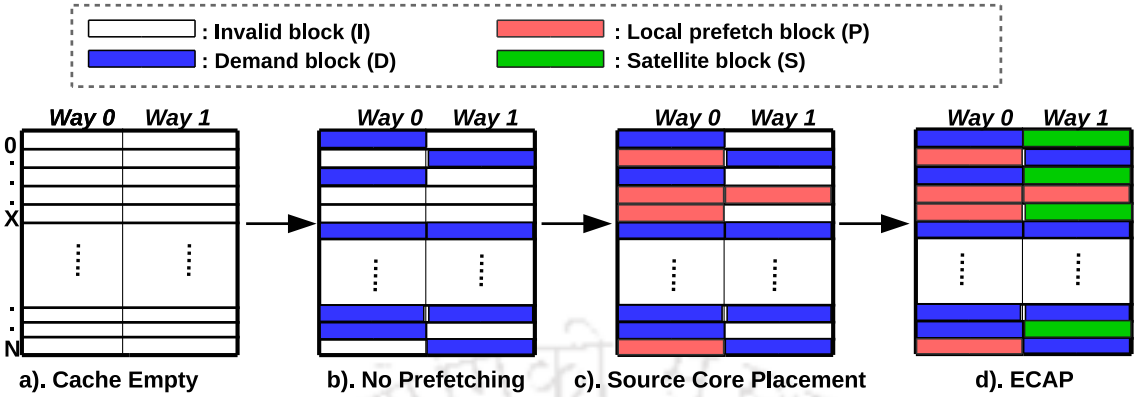

Figure 4.4: NVP enabled cache organization.

used or less used by counting the frequency of accesses within an interval of 1024 cycles. A 4-bit saturating counter per set is used with a threshold value of 12 for categorizing these sets. If one such target-set is identified, the corresponding remote cache acknowledges a positive response to the home cache. The remote cache also books one of its ways in the target-set to avoid race conditions. Hence, the prefetch block upon reaching the home cache is forwarded to the corresponding target-set in the remote cache where it is placed.

ECAP uses CARP for block replacement within a cache set. CARP assigns a confidence value to each cache blocks. The confidence value is a measure of block usage during its residency in the cache. Therefore, the block with the least confidence value is the victim block that is replaced by a prefetch block in the set. Section 4.3.2 explains CARP in details.

B. Metadata Management for Satellite Block Mapping

For maintaining the records of satellite blocks and their subsequent management, ECAP uses two pointers: forward and backward in the L1 cache of each tile. The forward pointer contains the detail of all prefetch blocks placed at the remote caches, and a backward pointer is used to identify a cache block owner (local block or satellite block).

Forward pointer: Forward pointer consists of Prefetch Tag Array (PTA) as shown in Figure 4.3(b). Each PTA index contains a set of Mapping Vectors (MV) where each MV has three fields: valid bit (V), tag, and direction flags. Tag field stores the address of a valid satellite block. Since in a mesh topology, a tile is surrounded by four adjacent tiles in North (N), South (S), East (E), and West (W), the direction flag corresponds to one of the four neighbors. Since MV is used to forward a cache miss request to a remote cache, we call the mapping vector a forward pointer. Hence, in a valid MV, precisely one of the

direction flags will be set. Also, the target-set of prefetch blocks stored in the remote cache is the same as that of the home cache. This is done to avoid storing the set number in MV, thereby reducing the hardware overhead. Experimentally, we found that each PTA index can store at most four prefetch blocks (n = 4) for optimized system performance.

Backward pointer: As shown in Figure 4.4(d), an ECAP enabled cache may contain I, D, P, or S type of blocks in its home cache. Four flags are used to indicate whether the block stored in the L1 cache belongs to itself or a neighbor (satellite blocks). Each flag corresponds to one of the four neighbors in a mesh topology. For satellite blocks, precisely one of the flags is set. On the other hand, none of the flags is set for demand blocks and local prefetch blocks. Since the flag indicates the block's owner, these are also called backward pointers, as shown in Figure 4.3(c).

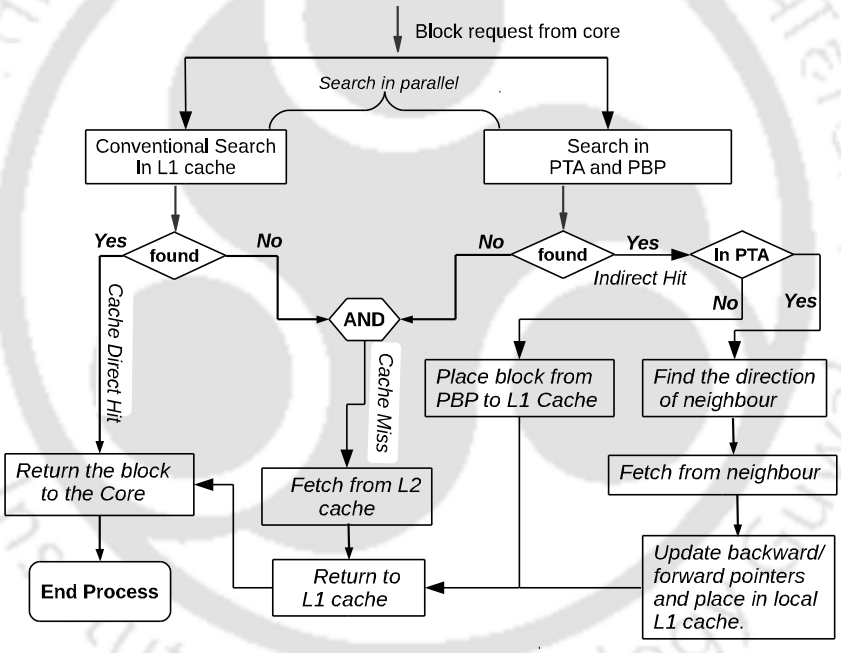

Figure 4.5: Flowchart for block searching.

C. Block Searching and Identification

Figure 4.5 shows the block searching technique in ECAP. Whenever the application requests for a block, it is initially searched in the home cache, PTA, and PBP of local tile, in parallel. If the block is a hit in the home cache, it is called a direct hit; otherwise, an indirect hit. On a direct hit, the requested word is returned to the core immediately. If the indirect-hit is from PTA, the block is in a remote cache. Using the forward pointer, the corresponding

mapping vector is retrieved that idenitfies the remote cache where the block is placed. Request packets are then sent to the remote tile to bring the cache block to its home cache. Upon placing the block in the home cache, the backward/forward pointers are updated in the remote cache and the local PTA. Similarly, on a PBP hit, the block is removed from PBP and placed in its home cache's target-set. Thereafter it is used as a demand block.

D. Cache Coherence Management

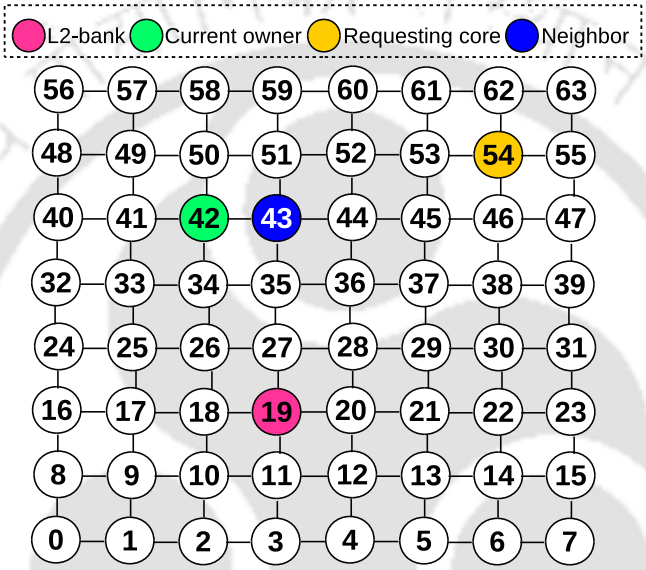

Figure 4.6: Example of coherence protocol for 8×8 2D mesh in ECAP.

As mentioned in Chapter 2, we use MESI CMP directory-based coherence protocol [50] to maintain data coherent across all the caches. In ECAP, the actual placement of prefetch blocks is known to the L1 cache only by recording its information in the PTA. The L2 cache is unaware of any such placement and is hidden from the underlying coherence protocol. However, implementing ECAP requires minor modification in the cache coherence protocol, as described in the following example.

Figure 4.6 shows a scenario where a prefetch request is sent from tile 42 to its home-bank i.e., tile 19. Considering that the block's target-set in tile 42 is a highly used set, the prefetch block upon reaching tile 42 is placed as a satellite block in tile 43 in the same target-set. The block's address is stored in the PTA of tile 42, and in tile 43, a backward pointer is set to tile 42, indicating the block's owner. The coherence protocol in the LLC updates the block status in its home-bank as E (Exclusive) and the owner as 42 (not tile 43). Upon receiving any RD/WR request for the same block from a different tile (say 54), the L2 bank sends a

request to the block's current owner, i.e., tile 42 as per the record. Tile 42 searches for the block using the block searching procedure, which results in a PTA hit. The remote cache is determined from the mapping vector, and tile 42 sends a request to tile 43 to forward its block to the requesting core in tile 54. Tile 54 acknowledges back to the block's home-bank (tile 19), and the status of the block is changed from E to S (Shared) with another owner as 54. Since the remote cache has no RD/WR permission on its satellite blocks, ECAP avoids data inconsistency that may arise when the block resides in a remote cache.

4.3.2 Confidence-Aware Replacement Policy

In this section, we describe the need for a prefetch-aware replacement policy followed by a detailed description on the insertion, promotion and demotion policy of CARP.

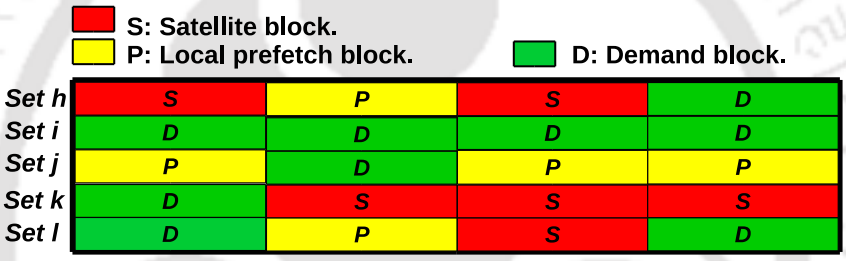

Figure 4.7: Snapshot of ECAP enabled cache where multiple colors are used to show the different blocks present in each cache set.

Need for a new replacement policy in ECAP

Traditional CMPs uses LRU replacement policy because of its simplicity [83, 84, 137]. LRU initially places an incoming block in the MRU position, and for every reuse of the block, it is promoted to MRU. Otherwise, the block eventually reaches the LRU position and gets evicted from the cache [83, 84]. Figure 4.7 shows an instance of a 4-way set-associative L1 cache filled with three types of blocks in ECAP. The sets are categorized as follows.

- **Category I:** Less used sets (*set i*) containing demand blocks only.
- **Category II:** Less used sets (*set j*) containing a combination of demand and local prefetch blocks.
- **Category III:** Less used sets (*set h, k, l*) containing a combination of demand blocks, local prefetch blocks and satellite blocks.
- **Category IV:** Highly used sets that are not used for prefetch block placement.

ECAP selects a less used set of category I, II, or III, as the target-set. Sets are classified as highly used or less used with the application's cache access behavior. When the access to a set is less, demand blocks residing in such a set experience minimal access. If LRU replacement policy is used, the blocks residing in the set may require a longer time to demote to the LRU position before its eviction. Hence, the chances of a demand block of becoming a deadblock [138, 139] is more. A block is said to be dead if it is never referenced after its last reference before eviction. Eviction of such blocks from the set gets delayed, resulting in poor usage of the cache space. Since prefetch blocks are expected to experience a hit in the future compared to a demand block, LRU replacement policy may demote such blocks faster. For categories II and III, the deadblocks may evict the prefetch blocks early. Therefore, LRU replacement policy may decrease the effectiveness of ECAP. In category IV, deadblocks may create a similar issue. However, ECAP does not use such sets for placing satellite blocks.

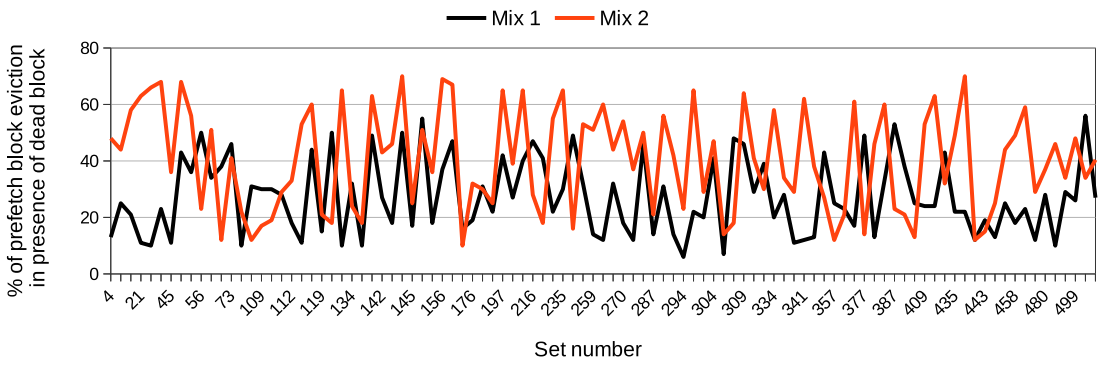

Figure 4.8: Percentage of prefetch blocks evicted in the presence of deadblock in a set.

Figure 4.8 shows the percentages of prefetch blocks evicted from a cache set in the presence of deadblocks. It is calculated as the ratio of prefetch blocks to deadblocks. The experiment is performed in an in-house stand-alone simulator written in C++. To detect deadblocks, the program takes the traces as input and simulates the cache evictions. Without a trace, detecting a deadblock with 100% accuracy is not possible as the future cache access pattern is unknown. From the figure, we can observe that in most cache sets, the prefetch blocks are unnecessarily evicted in the presence of deadblocks within the same set. We understand this limitation of conventional LRU policy and propose CARP that is more suited for ECAP enabled caches.

A. Insertion, Promotion and Demotion Policy of CARP

CARP attaches a confidence value per block that indicates the block's reuse frequency while its residency in the cache. It is represented using a 3 bit saturating counter per block. Thus, confidence values vary from 0 to 7, with 0 being the lowest and 7 being the highest. An incoming block is assigned a confidence value based on the set access. An equal confidence value is assigned to each block in a highly used set, i.e., 4 irrespective of its type. Hence, for such sets, CARP performs similar to LRU, where each block is initially placed in the MRU position. For less used sets, the confidence value for local prefetch block, satellite block and demand block are 4, 4, and 3, respectively. Since placing satellite blocks to cache incurs more cost in terms of network packets, more priority is given to such blocks by assigning a confidence value of 4 to keep them for a longer period. This helps in effectively using the less used sets in L1 caches of light applications. The confidence values are decided experimentally.

B. Replacement Policy of CARP

During block replacement, CARP chooses the cache block with the lowest confidence as the victim. If multiple such block exist, CARP selects a random block for eviction. Before evicting a victim block, the L1 cache controller checks the direction flags in its backward pointer. If any of the direction flags are set, it shows that the victim is a satellite block. Hence, the cache controller informs the owner of the satellite block by sending an invalidation message. In the meantime, the evicted block is stored in the victim buffer until it hears from the block's owner. Meanwhile, the tile can store the incoming block in its target-set. Upon receiving the invalidate request, the owner of the prefetch block invalidates the forward pointer in its PTA and acknowledges back to the remote cache. Upon receiving the acknowledgment, the remote cache evicts the block from the victim buffer to ensure cache consistency.

4.4 Experimental Analysis

ECAP is modelled in gem5 [40] with Booksim 2.0 [130] integrated to model the NoC. NVP is designed on Xilinx Vivado 2016.2 and implemented on ZedBoard Zynq-7000 ARM/FPGA SoC Development Board for hardware analysis and Orion 2.0 [128] is used to calculate the network power. The simulators are discussed in detail in Chapter 3.

Table 4.2: Simulation parameters for ECAP.

Processor	64, x86 OoO superscalar
L1 cache per tile	64KB, 4-way associative, 32B block
L2 cache per tile	1MB, 16-way associative, 64B block
L1 and L2 cache access time	2, 12 cycles
Victim buffer size	32
Prefetch block placement	SCP (baseline), ECAP (proposed)

4.4.1 Experimental Setup and Workload Description

We conduct the experiments using real benchmarks from SPEC CPU 2006 suite [131]. The experimental setup is already discussed in Table 3.3. However, Table 4.2 provides the simulation parameters that differs from those used in Chapter 3. We use SCP as the baseline prefetch block placement technique with the conventional next line prefetcher for data prefetching. PBP and PTA access latency are equal to that of L1 cache for experimental purposes. We have implemented NVP with LRU replacement policy (NVP-LRU), NVP with a tree-based Pseudo LRU replacement policy (NVP-PLRU), and NVP with CARP replacement policy, i.e., ECAP. ECAP's performance is analyzed with parameters such as cache miss in MPKI, flit count, packet latency, AMAT, percentage of direct and indirect hits, and packet distribution per hop. Section 4.4.7 analyzes the sensitivity of various design parameters to evaluate ECAP's performance.

Table 4.3: Workloads generated from SPEC CPU 2006.

High MPKI (H):	hmmer, leslie3d, lbm, mcf
Medium MPKI (M):	bwaves, gcc, bzip2, gamess
Low MPKI (L):	calculix, gromacs, h264ref, gobmk
B1 (Mix1)	$P(16), Q(16), R(16), S(16)$
B2 (Mix2)	$(P(4), Q(4), R(4), S(4))^4$
B3 (Mix3)	$(P(2), Q(2), R(2), S(2))^8$
B4 (Mix4)	$(P(4), Q(4))^4, (R(4), S(4))^4$
B5 (Mix5)	$(P, Q, R, S, S, R, Q, P)^8$

Workload Description: ECAP is evaluated using 12 benchmarks from SPEC 2006 suite. Table 4.3 shows the classification of benchmarks into three categories based on their MPKI as Low (MPKI < 5), Medium ($5 \leq$ MPKI < 25), and High (MPKI ≥ 25). The MPKI's are calculated after fast-forwarding for 10L cycles to avoid cold misses. As shown in Figure 4.9, 64 cores workload mixes are generated from the benchmarks where in each core one of the benchmark is mapped. The mixes are categorized into 5 patterns (B1 to B5), each consisting

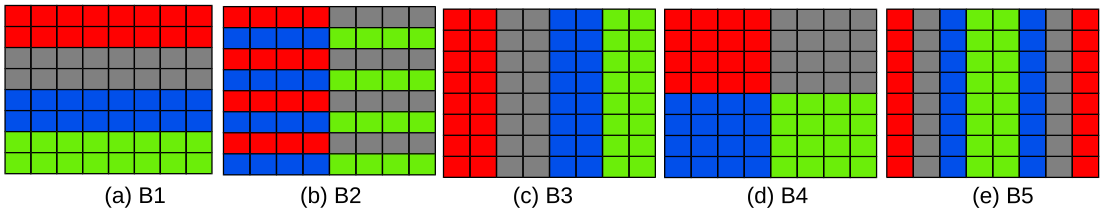

Figure 4.9: Workload mixes generated from SPEC CPU 2006 benchmarks.

of four benchmarks (P, Q, R, and S) chosen from High, Low, or Medium MPKI categories. P, Q, R and S are a combination of benchmarks selected from High/Low/Medium categories. As shown in the table, B1 is generated by mapping the first benchmark,P (red) to first 16 cores ($[0-15]$), next benchmark, Q to core numbered [16-31], third benchmark, R to cores numbered $[32-47]$, and fourth benchmark, S to the remaining set of cores, $[48-63]$. Similarly, B2 is generated by dividing 64 cores into four clusters, with each cluster having 16 cores. Within each cluster, the first benchmark, P runs on core numbered $[0-3]$, second benchmark, Q runs on core $[4-7]$, the third benchmark, R runs on core $[8-11]$, and the fourth benchmark, S runs on core $[12-15]$. The pattern is repeated across all four clusters. Similarly, B3, B4, and B5 workloads are also created.

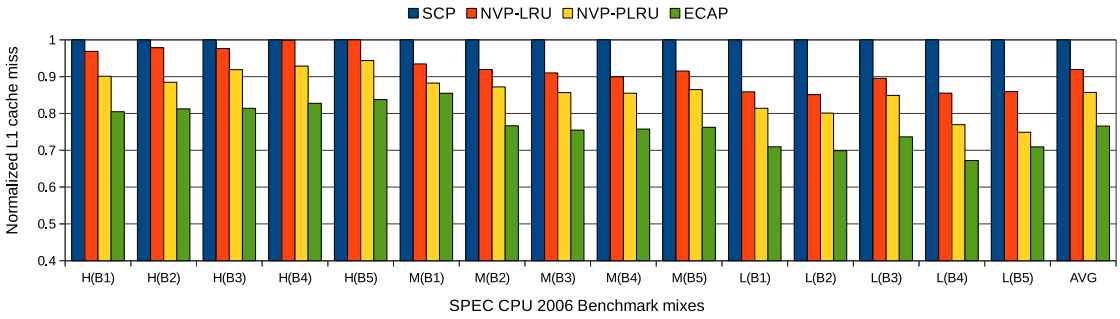

Figure 4.10: Comparison of L1 cache miss count normalized to SCP.

4.4.2 Effect on L1 Cache Miss

Figure 4.10 shows that, on average, the number of L1 cache misses for NVP-LRU, NVP-PLRU, and ECAP reduces by 8.05%, 14.28%, and 23.42%, respectively compared to SCP. ECAP reduces L1 cache miss by efficiently utilizing the less used cache sets and removing deadblocks from the caches, resulting in efficient cache space utilization. Since ECAP does not place prefetch blocks in the highly used set, it reduces cache pollution for such applications. ECAP only targets the less used set for prefetch block placement in its

home cache and remote caches, thereby decreasing the L1 cache miss count for all types of applications. However, for high MPKI workloads, the performance improvement of NVP-LRU is negligible compared to that of SCP. Since the L1 cache load is more in high MPKI applications, the home caches and the remote caches are almost utilized by the respective cores' applications. This makes it hard for NVP to place prefetch blocks in the remote caches.

LRU replacement policy cannot recognize deadblocks within a cache set, making NVP-LRU unable to utilize the cache space efficiently. Thus, the performance of NVP-PLRU and ECAP is better than NVP-LRU. The relative reduction of L1 cache misses in both the techniques is because NVP-PLRU provides a sub-optimal choice. In contrast, ECAP produces an optimal choice in identifying victim blocks during cache replacement. PLRU policy distinguishes only the most recently accessed cache block within a set. Apart from that, PLRU selects a victim randomly among the other blocks within the cache set. This results in providing a sub-optimal solution to find a victim block during prefetch block placement in the home cache and remote caches. Hence, NVP-PLRU may also evict a demand block that the core within few cycles has accessed.

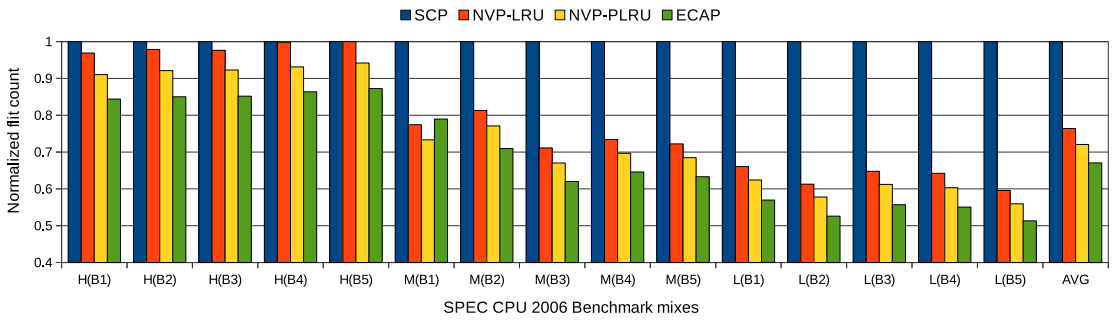

Figure 4.11: Comparison of count of flits generated in the NoC normalized to SCP.

4.4.3 Effect on NoC Related Parameters

When ECAP is incorporated in a tile, it effects the amount of flits generated (flit count) and average packet latency in the network. Figure 4.11 shows the flit count reduces by 23.58%, 27.93%, and 32.93% in NVP-LRU, NVP-PLRU, and ECAP, respectively compared to SCP. Reduction in cache miss results in generating fewer network packets. Since reduction in the number of network packet reduces the flit count, the average time required by a packet to travel from a source tile to the destination tile, i.e., packet latency also reduces. From

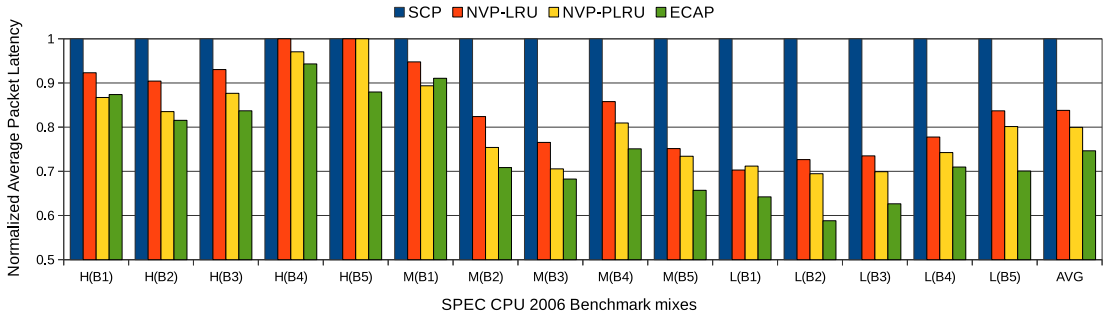

Figure 4.12: Comparison of average packet latency normalized to SCP.

figure 4.12, we can observe that NVP-LRU, NVP-PLRU, and ECAP reduce packet latency by 16.2%, 20.05%, and 25.34%, respectively.

In NVP, the reduction of network packets for high MPKI workload is less than medium and low MPKI workloads. This is evident from the fact that in high MPKI workloads, less used sets are very small in number. Therefore, most of the prefetch blocks are placed in the PBP of local tiles. Hence, high MPKI applications are not NVP friendly. However, ECAP reduces network packets for many applications by identifying deadblocks from the cache irrespective of the set type. We can observe that ECAP performs better than NVP-LRU and NVP-PLRU. In less used sets, the satellite blocks may experience delayed access due to their speculative nature. Therefore, the chances of evicting such a block are more than an inactive demand block. Since CARP provides more chances for satellite blocks to stay in a cache (assigning higher confidence values), such blocks are retained for a long duration.

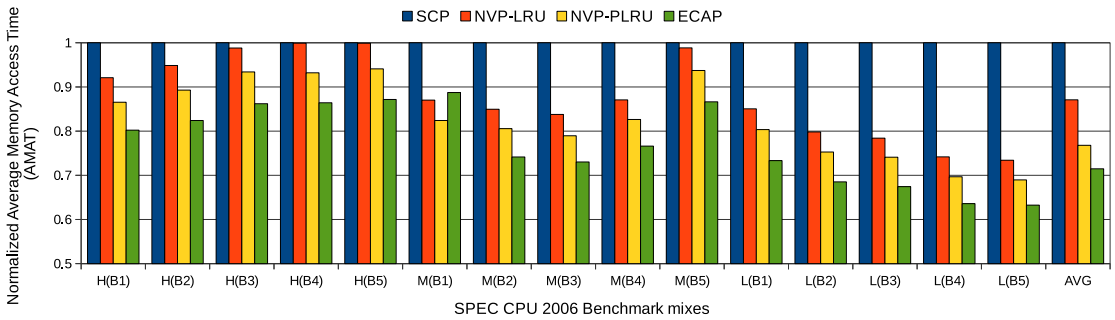

Figure 4.13: Comparison of AMAT normalized to SCP.

4.4.4 Effect on AMAT

As discussed in Chapter 1, AMAT in TCMP depends upon the hit time of L1 cache, miss rate of L1 cache, miss penalty of L1 cache, and the underlying network condition. Since cache misses contribute to network traffic, network congestion plays a major role in determining the AMAT. As shown in Figure 4.13, NVP-LRU, NVP-PLRU, and ECAP reduce 12.9%, 17.86%, and 23.56% of AMAT as compared to SCP. An average reduction of 32.93% network packets in ECAP reduces the packet latency by 25.34%. This has a direct impact on reducing AMAT during cache misses occurring in the core. For high applications, the reduction of packets in the network is less than compared to SCP. Therefore, the reduction of AMAT is also less for such applications. For medium and low MPKI applications, NVP-LRU, NVP-PLRU, and ECAP perform better. Therefore, the AMAT for medium and low applications shows a significant reduction compared to high MPKI benchmarks. However, ECAP achieves the best performance than NVP-LRU and NVP-PLRU, as shown in the figure.

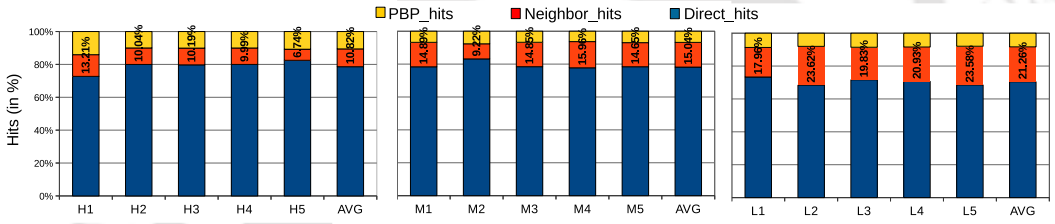

Figure 4.14: Percentage distribution of direct and indirect hits in ECAP.

4.4.5 Analysis of Hit Classification

Figure 4.14 shows the percentages of direct and indirect hits in ECAP for various benchmark mixes. Indirect hits are achieved either from PTA (PTA_hits) or PBP (PBP_hits). PTA_hits means the prefetch block is placed in remote caches. Therefore, applications with more PTA_hits are better ECAP friendly than those with fewer PTA_hits. From the figure, we can observe that the number of PBP_hits is more than that of PTA_hits for most heavy applications. From 18.8% of indirect hits, 7.9% hits are obtained from PTA, and the rest 10.9% hits are from PBP. As mentioned earlier, for high MPKI applications, the number of less used sets is substantially less. Therefore, most prefetch blocks are accommodated in the local tile PBP as shown in the figure. For medium MPKI applications, 21.74% of hits are indirect hits. Out of the total indirect hits, most hits are from PTA, i.e., 15.04%. In

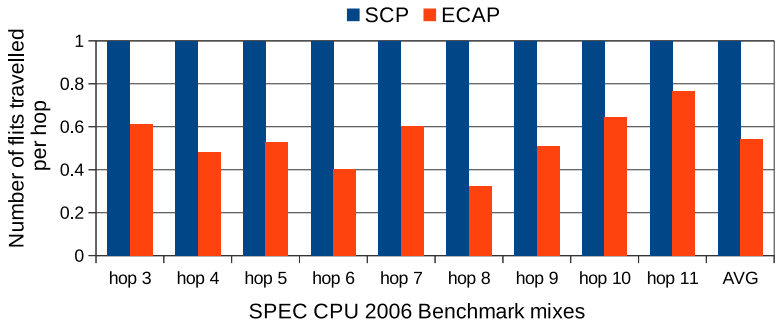

Figure 4.15: Comparison of long distance communication normalized to SCP.

low MPKI applications, 30% hits are indirect hits, and maximum hits are PTA_hits that constitute around 21.26% of total indirect hits. Therefore, ECAP is more suitable for light and medium MPKI applications.

4.4.6 Impact on Long Distance Network Packets

Figure 4.15 shows the normalized reduction of long distance packets in ECAP compared to SCP. In this figure, long distance packets are those that requires three or more hops to reach their destination tile. From the figure, we can observe that ECAP reduces around 49.2% of long distance packets in the network during a cache miss. In ECAP, the prefetch blocks are placed at remote caches that are at neighboring tiles. Such prefetch blocks are fetched from shorter distances (one hop), avoiding long distance communication during cache misses. Thus, ECAP hides long distance communication required to travel a request packet in the network by placing the prefetch blocks in a remote cache. This also justifies that the cache's existing demand blocks are not evicted due to the placement of prefetch blocks in remote caches. Thus, in a prefetch-enabled cache, ECAP reduces cache pollution.

4.4.7 Sensitivity Analysis

CARP uses window size (W) and different confidence values for satellite and local blocks (prefetch and demand block) in L1 cache to analyze the sensitivity to these parameters. For ECAP, the PBP size is also one of the parameters that determine ECAP's performance. Hence, for each PBP size, we change these parameters and each variation is named as $C\_W\_S\_D$ where W is the window size, S is the confidence value for satellite and local prefetch block, and D is the confidence value for local demand block. Figure 4.16, 4.17, 4.18 shows variation in different parameters and its impact on system performance.

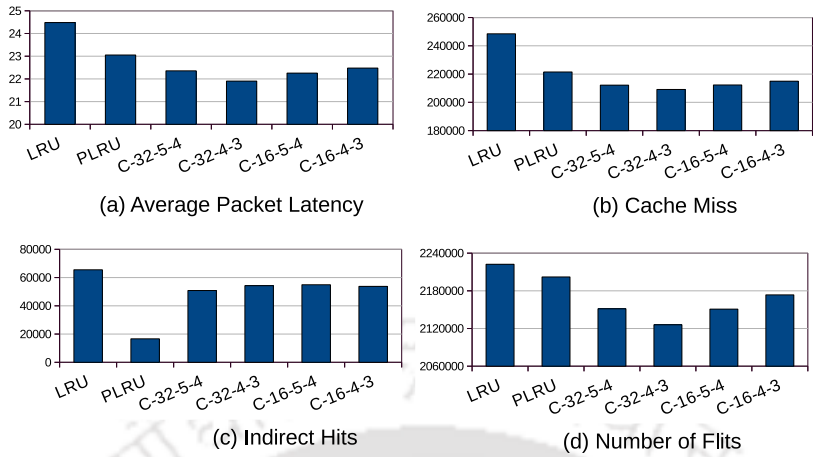

Figure 4.16: Performance of CARP with PBP size as 8.

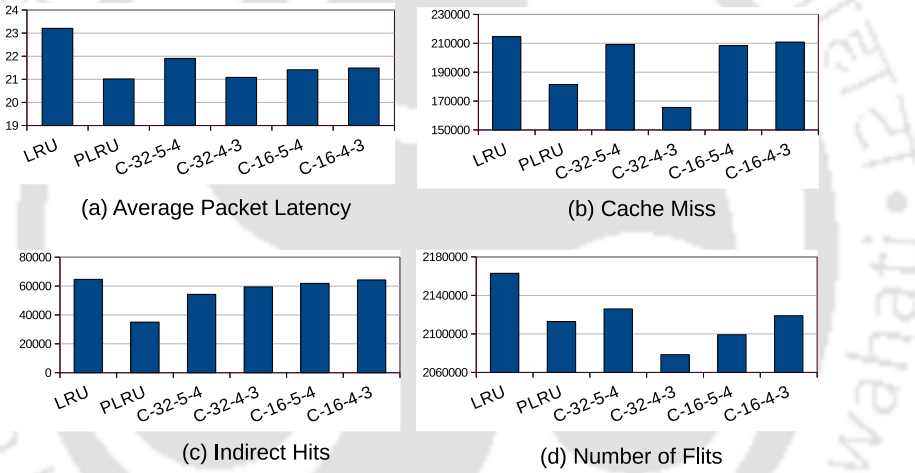

Figure 4.17: Performance of CARP with PBP size as 16.

From the figures, we observe that for different PBP size, the least value of average packet latency in figure 4.16(a), 4.17(a), and 4.18(a) is around 21.8, 21, and less than 21, respectively. This behavior is found in C_32_4_3 (PBP size = 32). It is self-explanatory because more prefetch blocks can be placed in the local tile due to the larger PBP size. Hence, the packet latency in the network reduces. Among all the variations, C_32_4_3 performs better than C_32_5_4, C_16_5_4, and C_16_4_3. In Figure 4.16(b), Figure 4.17(b) and Figure 4.18(b), we can observe that for W=32, the number of PTA_hits is almost similar for (C_32_5_4 and C_32_4_3). For W=16, the PTA-hits are less as compared to that of W=32. Figure 4.16(c), Figure 4.17(c), and Figure 4.18(c) shows the number of L1 cache miss occurred in different combination of W and confidence values. In

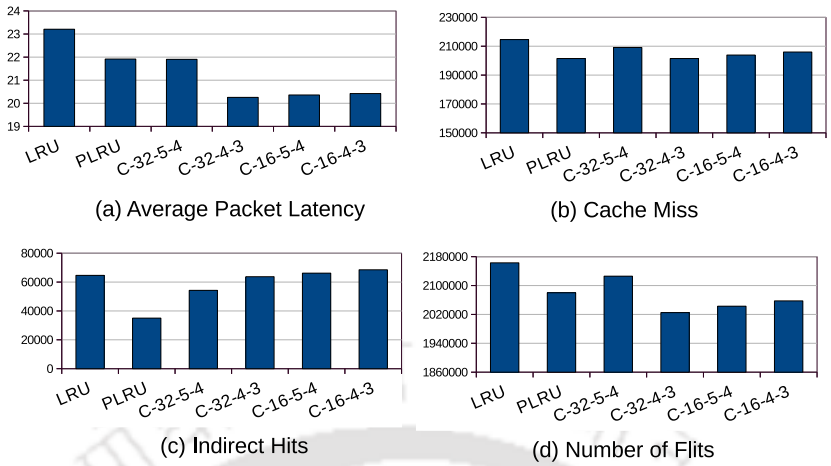

Figure 4.18: Performance of CARP with PBP size as 32.

Figure 4.17(c), C_32_4_3 experiences less miss as compared to C_32_5_4, C_16_5_4, and C_16_4_3. This scenario is also similar for Figure 4.16(c), and Figure 4.18(c).

When the window size is small, CARP reduces the confidence value of blocks faster. As a result of this, the blocks are demoted faster. This may evict a cache block from the lightly used set much before experiencing a hit. Experimentally it is found that C_32_4_3 is the optimal combination of PBP size, window size, and confidence value compared to other combinations. This is also evident from figure 4.16(d), Figure 4.17(d), and Figure 4.18(d), which shows that the reduction of flits is maximum for $W = 32$ where the confidence value for satellite block and local prefetch block is four and that of demand block is three. Therefore, we use PBP size as 16 and a window of 32 cycles throughout our experiments.

4.4.8 Hardware Analysis

The hardware analysis of ECAP is shown in Table 4.4 and Table 4.5. Table 4.4 shows that the extra hardware used in ECAP is a prefetch tag array, a prefetch buffer pool, backward pointer, access counter per set, and confidence value per block for CARP. Since, the main memory size is 4GB, a 32-bit address is used. Based on the conventional address split-up, 17-bits are reserved as tag per block, and 9-bits are for set index. Experimentally, ECAP performs well when PTA has four MVs per entry, where each MV comprises a valid bit, 17 tag bits, and 4 bits for direction flag. Thus, the total bits required for each MV is 22 bits. The backward pointer requires 4 bits as direction flag per cache block. As mentioned in subsection 4.4.7, the optimal value of PBP size is 16 which requires around 5632 bytes.

Table 4.4: Hardware overhead of ECAP per tile.

Cache size per tile:	**1024 KB (L2) + 64 KB (L1) = 1088 KB**		
Component	Bits/entry	number of entries	Bytes per tile
Backward Pointer	4	512 (sets) * 4 (ways)	1024
PTA	22	512 (sets) * 4 (MV)	5632
PBP	274	16	548
Access counter/set	4	512 (sets)	256
Confidence value/block	3	512 (sets) * 4 (ways)	768
Additional storage in ECAP per tile:			**≈9 KB**
Hardware cost per tile (in %):			**0.83%**

Table 4.5: Area and power consumption analysis in ECAP.

Component	L1 cache	L2 cache	PTA	PBP
Area(mm^2)	0.1374	1.92	0.0074	0.00027
Power (nJ)	0.0304	0.2872	0.0017	0.00006
Total Area (in SCP): (L1 + L2)				**2.0574** mm^2
Total Area (in ECAP):				**2.066** mm^2
Total Power (in SCP): (L1 + L2)				**0.3176** nJ
Total Power (in ECAP):				**0.3194** nJ
Increase in area per tile (in %):				**0.42%**
Increase in power per tile (in %):				**0.57%**

ECAP also stores a 4-bit value per set to check the access count within 1024 cycles. Also, 3-bits per block is used for assigning the confidence value of each block. If LRU replacement policy is used, a 4 way set-associative cache would require 2 bits per block. Hence, the additional hardware required in ECAP is 512 x 4 x 3 bits ≈ 768B.

Table 4.4 shows the hardware cost per tile for ECAP increases by 0.83% as compared to SCP. If a 64-bit address is used, ECAP has an additional hardware overhead of 1.47% per tile because of the increase in tag bits per mapping vector. From table 4.5, we can observe that the area and power consumption of ECAP increase minimally by 0.42% and 0.57% per tile as compared to SCP. Power is consumed when a flit resides in the VCs and flit traversals from one router to another through NoC links. ECAP, on average, reduces the router power and link power by 14.42% and 27%, respectively, compared to SCP. Since the reduction in cache pollution and cache miss reduces network traffic, the underlying NoC has to carry less traffic across the tiles. This resulted in reducing the dynamic power consumption at the routers and links.

4.5 Chapter Summary

In this chapter we discussed the limitations of placing prefetch blocks conventionally in a TCMP system and proposed ECAP that mitigates these problems by placing the prefetch blocks of heavy applications in the less used sets of light applications. This reduces prefetcher-caused cache pollution and results in fewer cache block replacement(s), having an incremental effect in reducing network packets. Reducing network packets directly impacts reducing network stall time, which reduces AMAT in TCMP. Thus, the throughput of core increases. However, ECAP cannot remove prefetcher-caused cache pollution altogether from the caches. In the next chapter, we look more into the prefetcher-caused cache pollution and propose ZPP, that removes cache pollution completely.

CHAPTER

5

ZPP: A Dynamic Technique to eliminate Cache Pollution

This chapter proposes Zero Prefetch Pollution (ZPP) to eliminate pre-fetcher-caused cache pollution completely from the caches. The difference between ECAP and ZPP is that ECAP uses application's behaviour (heavy or light) to reduce cache pollution while ZPP removes cache pollution for all applications irrespective of their behavior. Experiments conducted with real workloads showed enhanced system performance compared to existing techniques.

5.1 Introduction

As already discussed in Chapter 1 and Chapter 2, there exist a variety of prefetchers that provides good coverage [25, 140, 70, 44, 26, 66, 24], but not a single prefetcher can predict the complete memory access pattern of an application. Thus, useless prefetches that are fetched along with the useful ones can adversely harm an application by causing cache pollution where it replaces frequently used demand blocks from the caches. Cache pollution causes extra strain on the network, thereby increasing the $AMAT_{TCMP}$. Since the NoC is responsible for inter-tile communication and carries useful demand blocks required by the applications for their execution, its role in TCMP is central to the system performance. Thus, to make prefetching efficient, cache pollution must be handled properly. ECAP technique proposed in chapter 4 reduces prefetcher-caused cache pollution for those applications whose working set size does not fit into the available cache space, and prefetching on top of that causes severe performance degradation. However, it cannot completely eliminate the prefetcher-caused cache pollution. We cannot ignore the fact that both useless and useful prefetches may result in cache pollution. Since prefetch blocks are fetched for future use,

it is impossible to identify whether a prefetch block would experience a cache hit or not, irrespective of its usefulness.

This chapter proposes ZPP that exploits the fact that most of the exclusive blocks present in shared LLC banks are stale or useless copies. The updated copy of such block is present in some private L1 cache, and any request that the LLC receives for such blocks is served from the exclusively owned L1 cache to maintain data coherent across the system. If the stale block's metadata is kept intact in LLC, prefetch blocks can use such locations. Thus, ZPP reduces cache pollution by caching all prefetch blocks in the data portion of the stale block location of local L2 bank. Throughout the chapter, we use the term local bank to indicate the L2 bank of the same tile. Whenever a prefetch block experiences a hit, it is migrated from the local bank to L1. Thus, L1 cache always contains demand blocks. In addition to these, prefetch blocks in the local bank do not occupy any new locations and reuse those cache locations that hold stale blocks. This results in eliminating cache pollution at both L1 and L2 cache altogether as the name suggests, Zero Pollution Prefetcher. Another advantage of ZPP is increased prefetch requests. Since the size of local bank is larger than L1 cache, ZPP can prefetch more blocks by placing them in the local bank, increasing prefetch aggressiveness and coverage.

The key contributions of this chapter are as follows:

1. We study the limitations of existing prefetching techniques and to the best of our knowledge we find that none of them could completely reduce cache pollution.
2. We propose, ZPP, that eliminates cache pollution completely by placing prefetch blocks in the local bank's stale locations.
3. We perform extensive experiments using real workloads from PARSEC and SPEC CPU 2006 benchmark suites to validate our claims.

The chapter is organized as follows. Section 5.2 describes the motivation behind this work followed by the proposed technique in Section 5.3. Section 5.4 explains the experimental analysis and the chapter is finally summarized in Section 5.5.

5.2 Motivation

The experiments conducted in this section uses the similar configurations as mentioned in Table 5.1. The prefetcher used for the experimental analysis is BOP [44].

Motivation 1: What is the percentage of stale blocks in LLC? Figure 5.1 and Figure 5.2 shows the stale blocks present in L2 bank (Y-axis, green colour) for the benchmarks

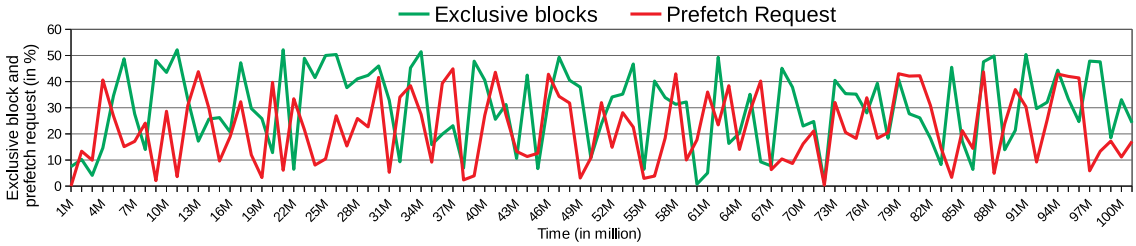

Figure 5.1: Percentages of stale blocks and prefetch requests in *vips*. The percentage is shown w.r.t. L1.

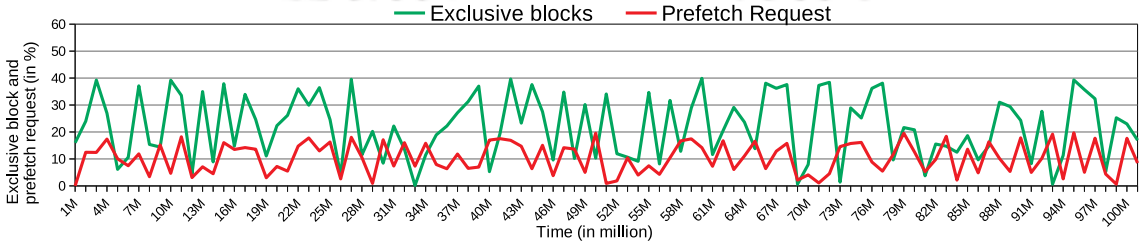

Figure 5.2: Percentages of stale blocks and prefetch requests in *milc*. The percentage is shown w.r.t. L1.

vips and *milc* from PARSEC and SPEC benchmark suite, respectively. The percentage in Y-axis is shown w.r.t. L1 cache. We can observe that each L2 bank contains some stale blocks at every time interval. As the application phase changes, the stale block count also changes. In Figure 5.1, *vips* has maximum number of stale blocks at $11M, 21M, 34M, 91M$ (more than 50%). On the other hand, at time $9M$, $38M$, $49M$, and $72M$, it has less than 4% of stale blocks. Similarly, in Figure 5.2 it can be observed that at time $3M$, the L2 bank has 40% of stale blocks while the same benchmark has 10% of stale blocks at time $7M$. Experimentally we found that in each L2 bank, among all the blocks currently present in L1, on average 32.4% and 52.6% blocks are stale blocks in PARSEC [41] and SPEC CPU 2006 [131] benchmarks, respectively. Since SPEC is multiprogrammed, each application runs only on a single core. Hence, we observe a higher number of stale blocks for these benchmarks. In PARSEC, multiple threads of the same application run on different cores, resulting in more shared blocks than SPEC. From this we can conclude that at every time instance, LLC banks contain some stale blocks whose data has no significance during its residency in the cache.

Motivation 2: How frequent is prefetch request generated? Figure 5.1 and Figure 5.2 shows that in *vips* and *milc*, majority of the time the number of prefetch requests is less

than the stale blocks available in the local bank.

Rationale: This shows that the stale blocks in local bank is sufficient to store prefetch blocks except for a few cases. In Figure 5.1 we can observe that prefetch requests are more than stale blocks at time $5M$, $21M, 33M, 38M, 59M, 67M, 83M$, and $97M$. For *milc*, few such cases are observed because the percentage of stale blocks is higher in SPEC than in PARSEC. Handling exceptional cases requires additional care, as described in Section 5.3.

Motivation 3: How long does it take for an stale block in LLC to change its state to non-stale? What is the average usage time of a prefetch block? The time duration of an L2 block to remain stale is called *stale duration (T_E)* of the block and the time taken by a prefetch block to get accessed is called *prefetch duration (T_{PU})*. Note that the stale duration and prefetch duration may vary for the same block in different occurrences. For getting benefits from ZPP, it must satisfy the condition $T_E > T_{PU}$.

Rationale: There are some cases when T_E of a local block (B_{local}) is less than T_{PU} of a prefetch block (B_P). In such cases, if the data of B_P is placed in B_{local}, it has to be invalidated before its usage. Section 5.3 addresses the issues. We have proposed three design policies that decide what to be done to the prefetch blocks that are currently placed in those locations. We experimentally found that the condition ($T_E > T_{PU}$) is satisfied for most of the cases. Hence, the data locations of stale blocks are sufficient for prefetch block placement. Section 5.4 supports this claim.

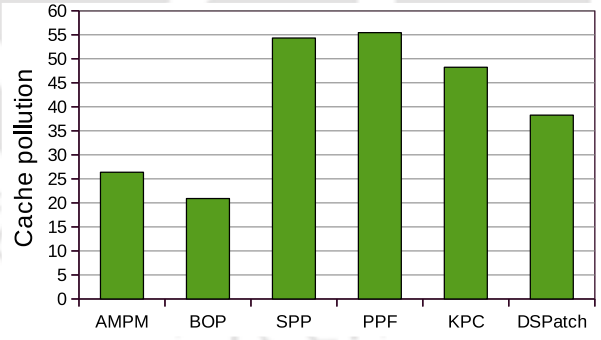

Figure 5.3: Percentage of cache pollution in state-of-the-art prefetchers

Motivation 4: Do existing prefetching techniques cause cache pollution? Cache pollution is a major concern of a prefetcher. Figure 5.3 shows the amount of cache pollution caused by prefetch blocks in L1 caches by AMPM, BOP, SPP, PPF, KPC and DSPatch is 26.38%, 20.9%, 54.3%, 55.45%, 48.2% and 38.3%, respectively.

Rationale: Prefetcher-caused cache pollution is inevitable. Though generating fewer

prefetch requests can reduce cache pollution, it may result in under-utilizing the prefetcher. So there arises a demand for new policies that can reduce or completely eliminate prefetcher-caused cache pollution by intelligent caching of prefetch blocks.

5.3 Proposed Technique

Whenever L1 prefetcher issues a request, it is sent to the block's home-bank through NoC. As the prefetch block reaches the requesting tile, instead of placing it in the conventional set in L1 cache, ZPP searches for a suitable location in the local bank that holds an stale block. ZPP takes benefit from such blocks of local bank to store the data part of the prefetch block. For this purpose, it uses an additional hardware, *Tracker*, that helps in finding one such location to be explained in Subsection 5.3.1. Upon finding a suitable location, the prefetch block is placed in the local bank, and the corresponding metadata is stored in *Tracker*. Subsection 5.3.2 explains prefetch block placement in ZPP. In case no location is found in the local bank, it uses a temporary buffer, *Prefetch Buffer* to store a few such blocks. Subsection 5.3.3 explains the searching of prefetch block in ZPP, and Subsection 5.3.4 explains the policy used to handle prefetch blocks when an stale block changes its state to non-stale states in the local bank (PF-Invalidation).

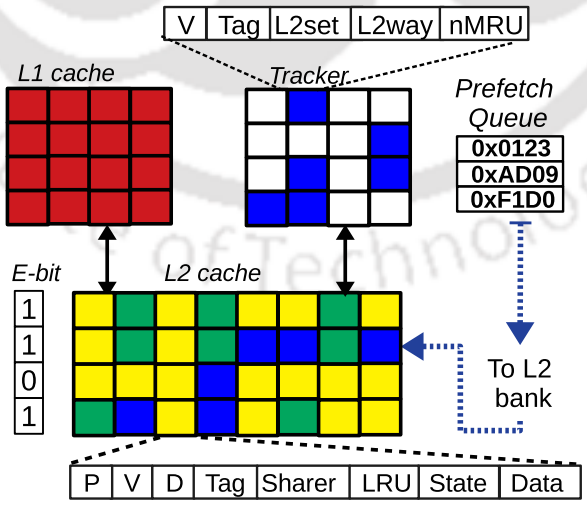

Figure 5.4: Interaction of Tracker module of ZPP with L2 cache bank inside a tile.

5.3.1 Tracker

As shown in Figure 5.4, each tile contains an additional hardware, *Tracker* that (a) keeps track of prefetch blocks placed in the local bank and (b) searches for stale blocks in the local bank for prefetch block placement. Each entry in Tracker consists of four fields: $<V$, Tag, $L2set$, $L2way$, $nMRU>$. *V-bit* indicates if the address of a prefetch block stored in the *Tag* field is valid. $L2set$ and $L2way$ is the set and way in the local bank (stale block location) where the prefetch block's data portion is stored. $nMRU$ bit is used to replace an entry from the Tracker using the non-MRU replacement policy. In this replacement policy, a block that is not MRU is evicted. The organization of Tracker is similar to that of L1 cache (Section 5.4.9). To represent the size of Tracker module, we use the notation xE-yW, where xE means that x number of prefetch tag (analogous to the number of total cache blocks in L1 cache) and yW indicates y way set-associative. For example, a Tracker of size 512E-4W is equivalent to a 32KB, 4 way set-associative L1 cache with 64B block size (can store 512 blocks).

On issuing a prefetch request, ZPP searches both L1 cache and Tracker in parallel. In this example, the number of entries, sets, and ways is the same in both Tracker and the L1. Therefore, Tracker is indexed using L1 set number determined from the block address. Upon finding an entry in the Tracker, the prefetch block can be directly accessed in the local bank using locations (L2set, L2way). Thus with search time equivalent to L1 cache access time, ZPP can locate prefetch blocks in the local bank.

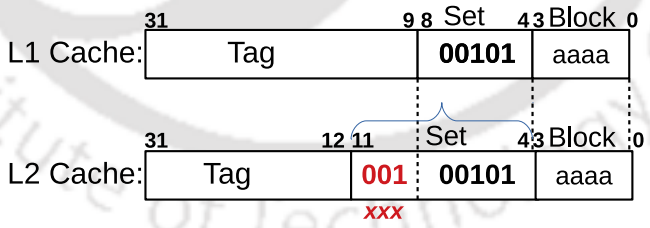

Figure 5.5: Address mapping between L1 and L2 cache.

ZPP uses the interesting property of conventional block mapping between L1 and L2 cache. Since L1 cache size is smaller than L2, a block that maps to a set in L1 can map to a total of $2^{|L2setbits-L1setbits|}$ locations in L2. Let us assume a 32-bit address whose L1 and L2 address split up is shown in figure 5.5. We can observe that if bits ranging from $(4-8)$ corresponds to the set-index portion in L1 cache, the same block maps to an L2

bank whose set is determined using bits $(4-11)$. Thus, only $(4-8)$ bits from the set index portion's LSB remain the same for both L1 and L2 cache. Therefore, even if the first few MSB bits from L2 varies, i.e., bit $(9-11)$, the block maps to the same location in L1. As shown in the figure, any block that maps to set $(00101)_2$, i.e., set 5 in L1 can map to sets $5, 37, 69, 101, 133, 165, 197$, and 229 in the local bank by varying the MSB bits 'xxx,' i.e., bit $(9-11)$. We call them $L2MappedSets$ for the corresponding L1 set.

ZPP uses the above mentioned static one-to-many block mapping between Tracker and the local bank to search for an stale block. The advantage of using static mapping is that it removes the restriction of having a one-to-one mapping of blocks between Tracker and the local bank. It also provides more options for prefetch block placement in the local bank. The maximum locations in the local bank that can be used for a prefetch block placement are equal to $\#L2MappedSets \times \#L2ways$. However, as shown in Figure 5.4, to reduce the search space, each L2 set uses an *E-bit* that indicates if there is at least one way in the set that holds an stale block. If the value of E-bit is 0, it indicates that no stale block exists, or prefetch blocks have used all the stale blocks in the set. It is also possible to design Tracker having numbers of sets and ways different than L1. However, we experimentally found that ZPP performs well when the entries that Tracker can hold is same as that of L1 cache.

5.3.2 Prefetch Block Placement in ZPP

A prefetch block (B_p) when maps to a particular set (S_t) in the Tracker, the $L2MappedSets$ are searched for an stale block. For each of these sets in the local bank, the E-bit is checked. If a set is found with *E-bit* as 1 ($L2set$), a way of the set with an stale block ($L2way$) is used by ZPP to store only the data portion of B_p. Another bit, P is used per block in the local bank to indicate if the data currently stored in the location ($[L2set][L2way]$) belongs to a prefetch block ($P\text{-}bit = 1$) or its own stale block ($P\text{-}bit = 0$). $L2set$ and $L2way$ are stored back in the Tracker along with the tag bits of B_p. If none of the $L2MappedSets$ has *E-bit* as 1, B_p is stored in a small buffer called *Prefetch Buffer*. Please note that a prefetch block in Tracker only maps to a particular set as per the mapping policy of any set-associative cache. For example, block A maps to set S_{t1} and block B maps to S_{t2}. The Tracker is considered full for B, if S_{t2} is full irrespective of S_{t1}. Placing a new prefetch block in ZPP requires handling four cases, as mentioned below.

Case I: When Tracker has space (not full) and there are unused stale blocks in the local bank. In this case ZPP works normally, as mentioned above.

Case II: Tracker is not full, but there are currently no unused stale blocks in the local bank. In this case, the new incoming prefetch block replaces an entry from the Tracker's corresponding set using non-MRU replacement policy. The corresponding stale block location in the local bank is used for storing the data portion of B_p.

Case III: Tracker is empty and there are no stale blocks in the local bank. In this case, a prefetch block is stored in the Prefetch Buffer. However, we experimentally found that such scenarios are very rare, as explained in Section 5.4.

Case IV: Tracker is full. In this case, irrespective of unused stale blocks in local bank, an entry from Tracker has to be removed to place the new prefetch block. Removing an entry from Tracker always removes the corresponding data from the local bank.

There may also arise a scenario such that the requested prefetch block's home-bank is the local bank. In such a case, the conventional way of block mapping is used, and the corresponding set and way where the block is stored in home-bank (same as local bank) are updated in the Tracker.

5.3.3 Searching for a Prefetch Block

Whenever a core requests a block, it is searched in L1 and Tracker in parallel using the conventional block mapping policy. If the block is found in L1 cache (demand hit), it is served immediately. On the other hand, if its a hit in Tracker (prefetch hit), the $L2set$ and $L2way$ are used to directly index into the local bank where the prefetch block is placed. The prefetch block is migrated from the local bank to L1 cache as a demand block. Subsequently, the Tracker's prefetch entry is invalidated, and the *P-bit* and *E-bit* are updated in the local bank, accordingly. Thus the prefetch block is served from the local bank, thereby avoiding the NoC stall time required to fetch the block from its home-bank. Upon a miss in both L1 and Tracker, the request is forwarded to the home-bank through NoC.

5.3.4 Stale to Non-stale State (PF-Invalidation)

In MESI coherence protocol, a prefetch block is replaced from the local bank when **(a)** The stale location where the block is placed changes its state from stale to S (Shared). This happens when the stale block receives a read request from another L1 cache, or **(b)** The stale location where a prefetch block is placed changes its state from E/M to I (Invalidation). This happens when the cache is full, and there is a new block request. Therefore, the stale block must be replaced from LLC to make room for the new incoming block. In both the

cases, when an stale block changes its state into a non-stale state (S or I), extra care is needed. This is because changing the state directly may result in inconsistencies as such location may contain a prefetch block whose metadata is in the Tracker. To overcome such an issue, the *P-bit* is checked. If the bit is set, it indicates that the data currently stored in the corresponding location is a prefetch block. Since there exist one-to-many mappings from Tracker and the local bank, the same technique is used for backward mapping from the local bank to Tracker, thereby determining Tracker's unique set storing the block's metadata. We call this PF-invalidation. Hence, we propose three different design policies for PF-invalidation.

A. ZPP with NoReordering (ZPP-NoR)

In this design policy, when an stale block, B, has a transition into a non-stale state, the corresponding prefetch data (if exists) must be invalidated. The presence of prefetch data can be detected from the *P-bit* of B. Let the location of B in the local bank is $[set, way]$. To invalidate the data, the Tracker invalidates the block having $L2set = set$ and $L2way = way$.

B. ZPP with Reordering (ZPP-RO)

In ZPP-RO, during PF-invalidation, the prefetch data is first tried to relocate into another unused stale location. To do this, at least one of the *L2mappedSets* must have the *E-bit* set. After reordering, the new set and way are updated in the Tracker. Thus, ZPP-RO avoids replacing prefetch blocks from the local bank, thereby storing such blocks for a longer time. If no such set in *L2mappedSets* exist (*E-bit*=0), ZPP-NoR is used.

C. ZPP with Hop restriction and Reordering (ZPP-HRO)

ZPP-HRO is an improvement over ZPP-RO where prefetch blocks fetched from far tiles are stored for a longer duration in LLC. It uses a selection critera for remapping the victim prefetch blocks to other L2mapped sets. The selection criterion is hop count. As shown in Figure 2.1, to access the one-hop neighboring tiles, minimum number of cycles (one-way) required in a zero-network load scenario is 5 (2 cycles in router + 1 cycle in link). Hence, long distance communication is very costly in terms of network hops [141]. Additionally, congestion along the path further adds to the NoC stall time experienced by the packets. For 4x4 TCMP, experimentally we found that an average of 35.6% blocks is prefetched from one-hop neighbors in a zero-network load. Therefore, during PF-invalidation, ZPP-HRO

remaps only those prefetch blocks to other L2mapped sets that are fetched from more than one-hop neighbours. For the other case, ZPP-NoR is used. Since these actions occur locally within a tile, no modification is required in the coherence protocol.

5.3.5 ZPP Vs Conventional L1 and L2 Prefetchers

ZPP is different from L2 prefetcher in the following aspects.

- It can prefetch a block in the stale locations of local bank for a longer time.
- During a cache miss it avoids network stall time in fetching block from its home-bank.
- It avoids tag comparison in L2 as the prefetch block location is directly obtained from the Tracker.
- Conventional L2 prefetchers causes cache pollution, which is eliminated in ZPP.
- The accuracy of L2 prefetcher is less than that of L1 prefetcher as the memory access pattern is jumbled up in L2 level. But ZPP uses the accuracy of L1 prefetcher and places block in L2 bank to increases prefetch aggressiveness.

ZPP is different from L1 prefetcher in the following aspects.

- An L1 prefetcher causes cache pollution, but ZPP does not cause any cache pollution. If any demand request to a block is a hit in the Tracker, the prefetch block is placed from the local bank to L1 cache. Therefore, L1 cache contains only demand blocks, thereby avoiding cache pollution.
- The prefetch hit time in ZPP is equal to that of L1 cache. ZPP only requires few additional cycle in fetching the block from local bank to L1 cache
- In conventional L1 prefetcher, maximum prefetch blocks that it can request at a time is equal to the number of L1 cache blocks. But replacing all demand blocks may degrade the applications' performance. ZPP, on the other hand, increases prefetch aggressiveness and coverage by placing the prefetch blocks in local banks whose size is larger as compared to L1 cache.

5.3.6 Algorithm of ZPP

Algorithm 1 has two main modules: tracker and local bank. The three main operations of ZPP are shown in the algorithm as placement, search, and PF-invalidation. The algorithm only shows the functionality of ZPP-NoR. Whenever a prefetch request is generated, the function ***issuePFRequest*** (Line 1) is invoked. The function places prefetch blocks based on the availability of stale blocks in the local bank. In case of prefetch hit, the function

managePFHit (Line 24) is invoked which migrates the block to L1 cache. The PF-invalidation is performed by invoking the function ***invalidPF*** (Line 18). For ZPP-RO and ZPP-HRO, a slight modification of function *invalidPF* would suffice.

ALGORITHM 1: Algorithm for ZPP.

```
1  Tracker: void issuePFRequest(Address addr)
       /* Prefetcher issues request with Address addr.                    */
2      if aaddr is already in tracker then
3          return /* stop prefetching                                     */
4      if tracker has space for allocating addr then
5          if free stale location available in local bank then
6              allocate addr in tracker;
7              place the block in local bank, set P-bit;
8          else if Tracker has space but local bank has no free stale space. then
9              remove an existing entry from Tracker.
10             the same will be removed from local bank
11             allocate new entry in tracker and local bank
12         else /* When Tracker is empty but local bank has no stale location for
                addr.                                                      */
13             store in Prefetch Buffer.
14     else /* When Tracker has no free space.                            */
15         remove an existing entry from Tracker.
16         the same will be removed from local bank
17         allocate new entry in tracker and local bank
18 Local: void invalidPF(Address local-addr)
       /* Invalidate the prefetch data stored in local bank, where the block with
       address local-addr is placed.                                      */
19     a) Find the location (set_id, way_id) of local bank where local-addr is placed.
20     b) Get the location in the L2MappedSets.
21     c) Invalidate the entry from Tracker.
22     d) Remove the prefetch entry from local bank.
23     e) Change E-bit for local bank set (if required).
24 Tracker: void managePFHit(Address addr)
       /* In case of prefetch hit.                                        */
25     a) find the entry, addr in Tracker and local bank.
26     b) migrate the entry to L1D.
27     c) remove the entry from Tracker and local bank.
```

5.4 Experimental Analysis

ZPP is modeled in gem5 [40] and CACTI 6.5 [129] is used to measure Tracker's area and power. Details of these simulators are already discussed in Chapter 3.

Table 5.1: Simulation parameters for ZPP

Processor	64, x86 cores OoO superscalar
L1 cache per tile	32KB, 4-way associative, 64B block
L2 cache per tile	256KB, 8-way associative, 64B block
Tracker Size for ZPP	512E-4W (Section 5.3.1)
L1 and L2 cache access time	2, 10 cycles
Tracker Access Latency (TrAL)	2 cycles
Prefetcher	Best Offset Prefetcher [44]
Prefetch Aggressiveness	4

Table 5.2: SPEC CPU 2006 benchmarks categorized into high/medium/low.

High (H)	lbm, lib, sphinx, sjeng, leslies3d, canneal, dedup, vips
Medium (M)	hmmer, gromacs, facesim, stream, mcf
Low (L)	soplex, bzip2, gobmk, milc, swap, black, body

5.4.1 Experimental Setup and Workload Description

The simulation parameters used to model ZPP is mentioned in Table 5.1. All the variants of ZPP; ZPP-NoR, ZPP-RO and ZPP-HRO are implemented as the new prefetch block placement technique over the baseline prefetcher, BOP [44]. Tracker and Prefetch Buffer size used for experimental analysis is mentioned in Section 5.4.10 and 5.4.8. For evaluating ZPP's performance, we model state-of-the-art prefetchers AMPM [70], BOP [44], SPP [25], PPF [24], and DSPatch [142] (with SPP as the underlying prefetcher) with similar simulation parameters as mentioned in Table 5.1. The table contains only those simulation parameters that differs from Table 3.3 in Chapter 3. Different performance matrices are used to evaluate ZPP such as ZPP-induced prefetch hits, weighted speedup, network stall time, prefetch coverage, and prefetch accuracy normalized to a system with prefetching disabled as Baseline (NoPref). We have also performed some detailed analysis of ZPP in Subsection 5.4.8. All the timing constraints are taken care of while implementing ZPP.

We perform extensive analysis on 64 cores TCMP using 8 multithreaded benchmarks from PARSEC [41], and a total of 320 workloads consisting of benchmark mixes from SPEC CPU 2006 benchmark suite [131] (described in Chapter 3). Among the PARSEC benchmarks *black, body, canneal, dedup, facesim, stream, swap*, and *vips* are used with *sim-large* as input to these benchmarks. From SPEC benchmarks *lbm, libquantum (lib), sphinx, soplex, sjeng, bzip2, leslie*3*d, mcf, hmmer, gobmk, gromacs, and milc* are used. As shown in Table 5.2, the benchmarks are categorized into High (H), Medium (M) and Low (L)

Table 5.3: Workload mixes generated using SPEC benchmark for 64 cores; xP - yQ means x% of P benchmark and y% of Q benchmark is used to create the mix.

W1, W2, W3	64H, 64M, 64L
W4 , W5 W6, W7 W8, W9	(0.75H, 0.25M), (0.75H, 0.25L) (0.75M, 0.25L) , (0.75L, 0.25M) (0.75L, 0.25H), (0.75M, 0.25H)
W10, W11 W12	(0.5H, 0.5M), (0.5H, 0.5L) (0.5M, 0.5L)
W13, W14, W15	(0.25H, 0.5M, 0.25L) (0.25H, 0.5L, 0.25M) (0.25M, 0.5H, 0.25L)
W16	Mixed combination

based on the IPC values when prefetching is enabled. High are those benchmarks where IPC increases by more than 10%, medium are those where IPC ranges between $2\% - 10\%$ and low are those benchmarks whose IPC is less than or equal to 2%. Table 5.3 shows the combination of benchmarks used from each of the H/M/L categories to form a workload. In the table, Wi represents a workload where xH/M/L indicates that x number of benchmarks from H/M/L category is used for the mix and mapping of each benchmarks to a core is already discussed in Chapter 3 (Section 3.2). During simulation, each workload is fast forwarded for 200 million instructions, warmed up for next 200 million instructions and then executed for next 350 million instructions for collecting the statistics. The results plotted for each of the workload mix (Wi) is a geometric mean of 384 different ways of benchmark mapping for 8×8 TCMP.

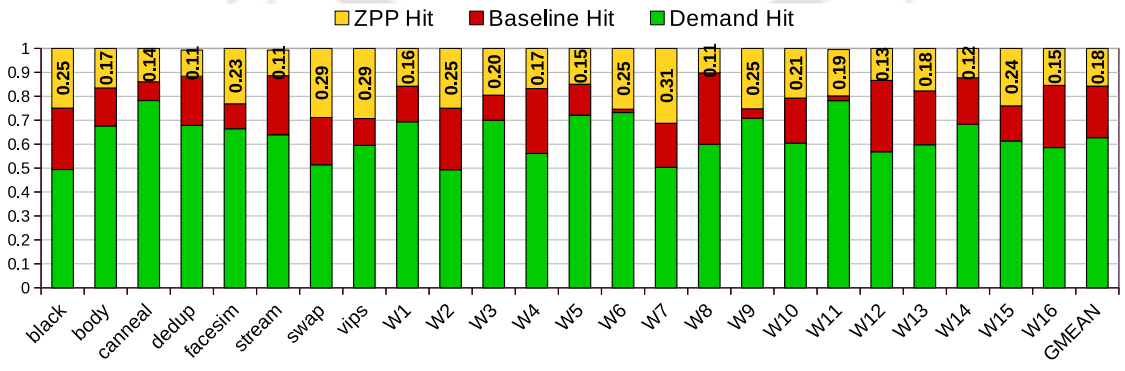

Figure 5.6: Classification of L1 cache hits obtained in ZPP.

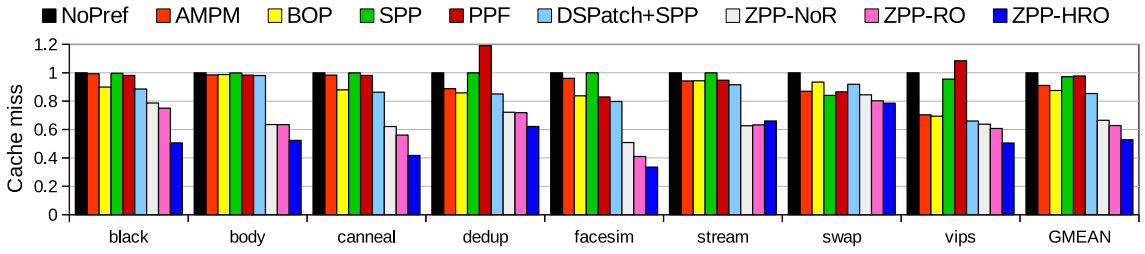

Figure 5.7: Comparison of L1 cache miss count normalized to NoPref for PARSEC benchmarks.

5.4.2 Analysis of ZPP-induced Prefetch Hits

Figure 5.6 shows the distribution of cache hits obtained in ZPP. ZPP-induced hits are the number of prefetch blocks that experience cache hit when it exceeds the baseline's prefetch duration, T_{PU}. BOP is used as the baseline prefetcher whose average T_{PU} is 132 cycles (between $20 - 348$ cycles) across all benchmarks. Baseline hit is the number of prefetch blocks that experiences hits within the prefetch duration ($\leq T_{PU}$), and demand hit is the hits obtained when blocks are fetched on demand. We categorize the prefetch hits as Baseline hit and ZPP-induced hits to observe the additional cache hits obtained due to ZPP. As shown in the figure, 18% of the extra cache hits are obtained due to ZPP along with 21% of Baseline hits. Had ZPP not been incorporated in the system, 18% of the cache hits would have turned into a miss that would require additional cycles to fetch the block from another tile. Thus, ZPP reduces tile-to-tile communication by reducing cache miss.

5.4.3 Effect on L1 Cache Miss

It is impossible to predict an application's entire access pattern. However, as illustrated in Figures 5.7, Figure 5.8, and Figure 5.9, different forms of ZPP substantially reduce L1 cache miss when compared to existing strategies. In these figures, all techniques are normalised to a system that does not use prefetching (NoPref). As shown in the figures, an increase in prefetch hits by ZPP resulted in a reduction in the number of cache misses for various benchmarks. We can see that the cache misses in the PARSEC benchmarks *canneal, facesim*, and *tvips*Âăare the least in all three variations of ZPP. *canneal* and *vips* are highly prefetch friendly. The working set in *canneal* is large and does not completely fit into the cache.

Similarly, *vips* is a multimedia application where the data have more spatial than temporal locality. Thus, these benchmarks benefit from prefetching. However, threads

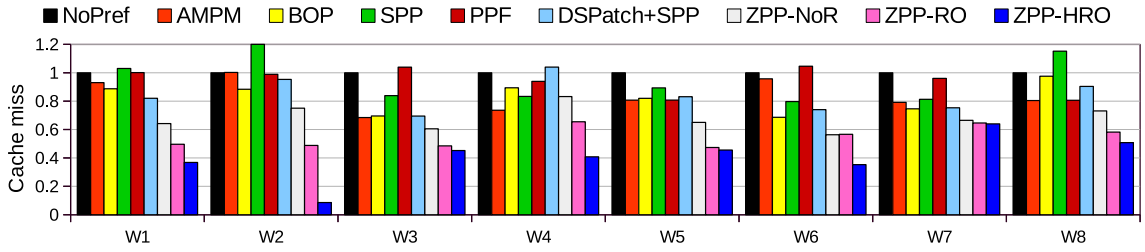

Figure 5.8: Comparison of L1 cache miss count normalized to NoPref for SPEC Workloads (W1-W8).

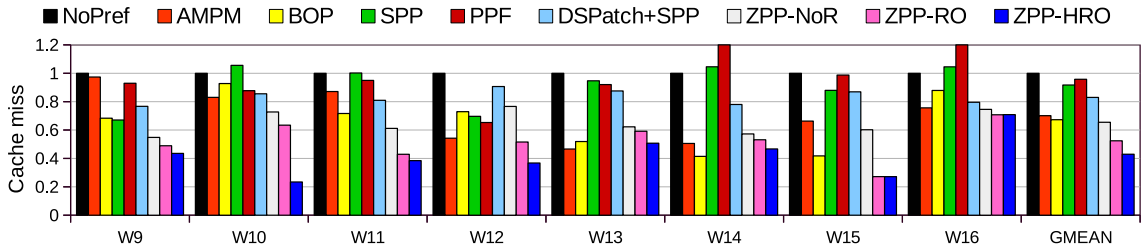

Figure 5.9: Comparison of L1 cache miss count normalized to NoPref for SPEC Workloads (W9-W16).

running parallel in multiple cores for *canneal* heavily share data among themselves. Therefore, prefetch blocks may evict the data shared among the threads, thereby resulting in prefetcher-caused cache pollution. Since ZPP completely removes cache pollution, the overall cache miss reduces for the benchmark. On the other hand, for *vips*, the size of the local bank is larger than that of the L1 cache, ZPP tries to place more prefetch blocks in the local tile. This reduces the number of cache miss in ZPP-HRO by 27.7% as compared to conventional BOP. *Facesim*, on the other hand, is a medium prefetch friendly application and the data sharing among the threads is also less. This, in turn, reduces the number of shared blocks in L2 banks. Thus, ZPP can prefetch more blocks that are placed in the local bank, which reduces the number of cache misses for the application.

The reduction of cache miss in ZPP-NoR, ZPP-RO and ZPP-HRO for the application *dedup* is 27%, 28% and 37.8%, respectively. *dedup* uses a data pipeline, and each stage of the pipeline is computed on a single thread. The concurrently running threads communicate among themselves, resulting in higher data sharing in the system. Thus, the number of prefetch blocks that ZPP fetches is more than that of the existing techniques but it is slightly less as compared to *canneal* and *vips*. *Black* and *body* are prefetch non-friendly

benchmarks, and excessive prefetching for these benchmarks may harm the application. Therefore, existing lookahead prefetchers like SPP and PPF increase the number of cache misses for these applications. *Black* shares less data among the concurrent running threads. Thus the number of stale blocks in the local bank is more.

Also, the application's working set is small, which indicates that the blocks fetched are important and evicting such blocks may harm the application. Therefore, it falls under the low prefetch friendly category. But prefetching for these benchmarks may evict the important blocks from the cache, which ZPP and all its variants avoid entirely. Thus, *black* performs better than the existing prefetch techniques. *Swap* also shares less data among the threads, making ZPP efficient for the benchmark. ZPP benefits *stream* in reducing cache misses because the applications' memory access pattern follows a regular pattern, and it has less data sharing among the threads. Thus, ZPP takes the benefit of both stale blocks and prefetching correct blocks for the application. Among all the benchmarks, ZPP performs least in *body*. *Body* is a prefetch unfriendly application with a medium working set that has high data sharing among the threads. Thus, blocks in local banks are mostly in the shared state. This results in frequent evicting of prefetch blocks from the local bank in ZPP-NoR. ZPP-RO reorders the prefetch blocks in other stale locations. However, with the number of stale blocks being less, ZPP-NoR and ZPP-RO perform the same. ZPP-HRO performs better than the other two variants because it selectively reorders prefetch blocks when the stale location changes into a non-stale state.

Among the workload mixes, the number of cache misses for workloads such as $W1$, $W3$, $W6$, $W9$, $W11$, $W13$, $W14$, and $W15$ is less in ZPP and its variants as compared to the other techniques. Most of these benchmarks contain a combination of *lib-lbm-leslie3d-sjeng-gromacs* which are either high or medium prefetch friendly benchmarks. Though the fetch time from the local bank is more in ZPP, the number of prefetch hits obtained in ZPP is higher than a conventional L1 prefetcher. This increases prefetch accuracy. Thus, ZPP reduces the number of cache misses for these applications, thereby providing better prefetch coverage as shown in Figure 5.10. The overall reduction in the number of cache misses for AMPM, BOP, DSPatch, ZPP-NoR, ZPP-RO and ZPP-HRO is 24.8%, 28%, 16.7%, 34%, 47.6% and 62%, respectively while in SPP, PPF the overall cache miss reduces by mere 7.76% and 3.88%.

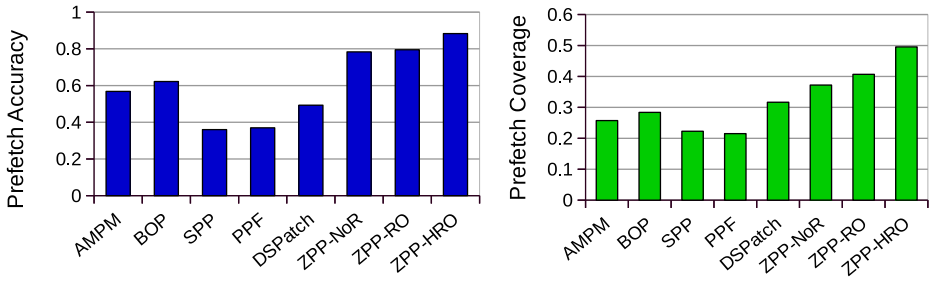

Figure 5.10: Comparison of prefetch accuracy and coverage in different techniques.

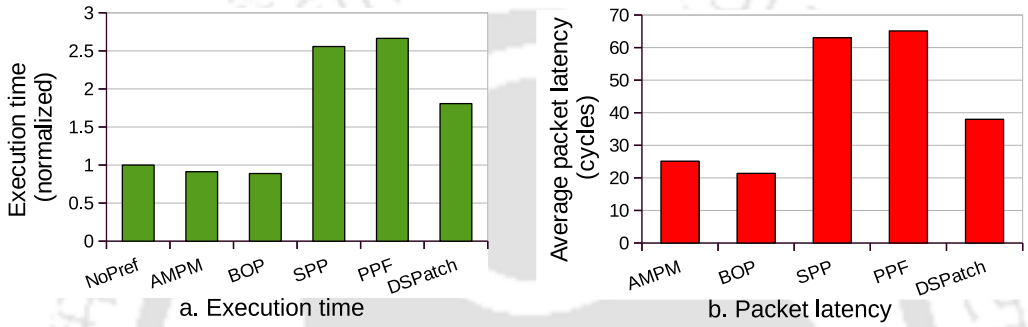

Figure 5.11: (a) Normalized execution time (wrt NoPref) and (b) Average packet latency in different techniques.

5.4.4 Effect on Prefetch Related Parameters

Figure 5.10 shows the average prefetch accuracy and coverage taken from PARSEC and SPEC 2006 benchmarks. Prefetch accuracy and coverage (accuracy, coverage) obtained in AMPM, BOP, SPP, PPF, DSPatch, ZPP-NoR, ZPP-RO and ZPP-HRO are (0.56, 0.26), (0.62, 0.28), (0.36, 0.22), (0.36, 0.21), (0.49, 0.31), (0.78, 0.38), (0.79, 0.41) and (0.88, 0.50), respectively. Upon comparing the variants of ZPP with other techniques, we observe that the accuracy and coverage of ZPP variants are better than the existing techniques with ZPP-HRO performing the best.

We also observe that in Figure 5.10, BOP performs better than AMPM, SPP, PPF and DSPatch. As mentioned in Chapter 2 SPP and PPF generate unnecessary NoC packets to train the prefetcher in throttling unwanted prefetches. In this process, all L2 demand accesses and block evictions are communicated to all the tiles as training packets. These packets are sent using NoC, which additionally contributes 68% to the existing NoC traffic consisting of demand and prefetch packets. This is clearly observed in the results plotted in Figure 5.11. We can observe that the training packets significantly contribute to NoC

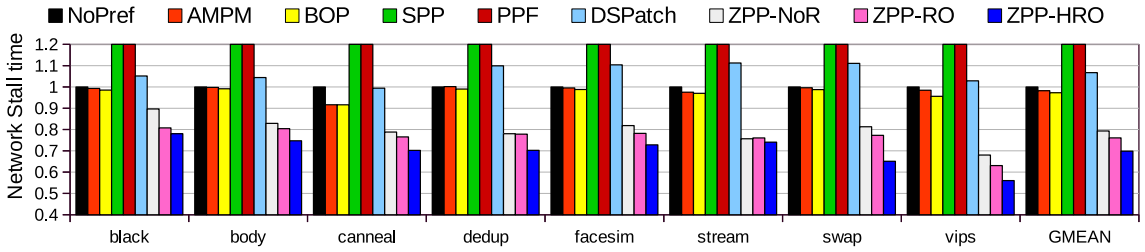

Figure 5.12: Comparison of NST normalized to NoPref for PARSEC benchmarks.

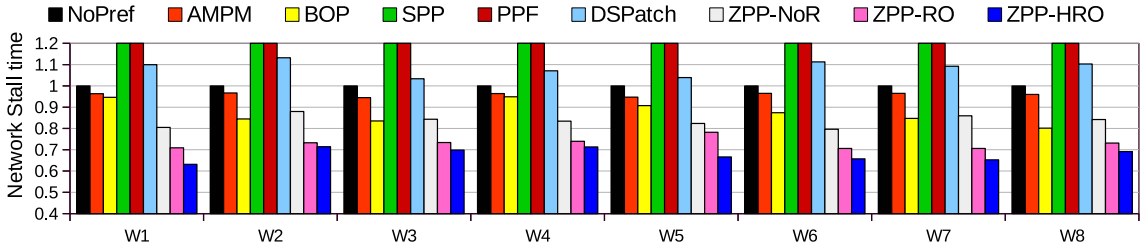

Figure 5.13: Comparison of Network Stall Time normalized to NoPref for SPEC workloads (W1-W8).

contention, which delays the demand and prefetch packets. As a result of this, the execution time of these applications increases as shown in the figure. ZPP, on the other hand, holds prefetch blocks for a longer duration in the local tile. Therefore, the future cache misses that could have occurred are covered by it, resulting in faster instruction execution.

5.4.5 Effect on Network Stall Time

Network stall time is calculated as the sum of packet latency and queuing latency of NoC packets. The lower the stall time, the better is the technique. Figure 5.12, Figure 5.13 and Figure 5.14 shows the network stall time in different techniques normalized to a system with prefetching disabled (NoPref). Reduction in cache miss reduces the network stall time. Hence, among all the benchmarks, ZPP and its variants perform better than the existing techniques. SPP and PPF are the least in terms of network stall time as the training packets generate tremendous NoC contention, which increases packet latency and queuing latency in NoC. The overall reduction in network stall time for ZPP-NoR, ZPP-RO and ZPP-HRO is 18%, 24%, and 30.1%, respectively. Since network stall time reduces in ZPP, $AMAT_{TCMP}$ reduces by 27.82% across all the benchmarks.

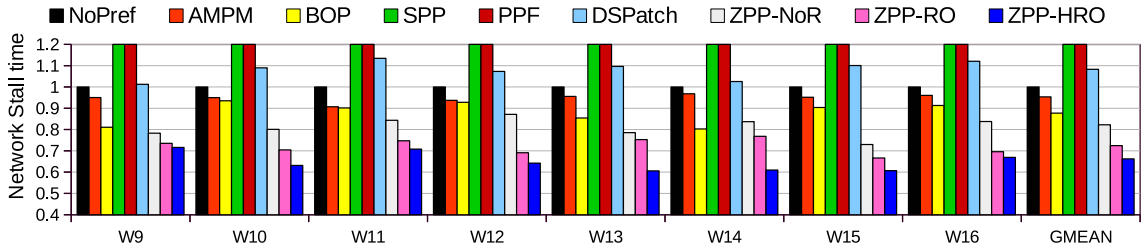

Figure 5.14: Comparison of Network Stall Time normalized to NoPref for SPEC workloads (W9-W16).

Table 5.4: Effect on Weighted Speedup normalized to NoPref.

	AMPM	BOP	SPP	PPF	DSPatch	ZPP-NoR	ZPP-RO	ZPP-HRO
WS	1.026	1.055	0.989	0.944	1.078	1.13	1.646	2.19

5.4.6 Effect on Weighted Speedup

Table 5.4 shows the weighted speedup obtained in different techniques. From the table, we can observe that across all the techniques, ZPP-HRO performs the best with an increased speedup of 2.19 as compared to a system with NoPref. As mentioned in Section 5.3.2, ZPP-NoR replaces the prefetch blocks to make room for new blocks. This makes ZPP-NoR lose on the performance aspect as the replaced prefetch blocks when re-requested, are re-fetched from the home-bank of the block rather than the local tile. If the home-bank lies in a different tile, fetching such blocks requires additional network latency. ZPP-RO avoids this scenario by searching for stale blocks in any of the *L2mappedSets*. If there is one such location, ZPP-RO copies the prefetch data in the new location, and the *L2set* and *L2way* are updated in the Tracker to locate the prefetch block correctly. Thus, it can place prefetch blocks for a longer period as compared to ZPP-NoR. However, we have observed that in ZPP-RO, there are few cases when there exist no stale locations in the local bank (*E-bit*=0). The number of prefetch blocks that fail to re-order is addressed in the next section. In such a scenario, ZPP-RO changes back to ZPP-NoR.

ZPP-HRO, in turn, avoids replacing blocks that are prefetched from farther tiles (i.e., network hops > 2.). Since fetching blocks from nearby tiles requires a minimum of 10 cycles in a zero-load network than prefetching from a far tile, ZPP-HRO prioritizes prefetch blocks on hop distance. This avoids reordering prefetch blocks of neighboring tiles (hops=1) and uses those reordering for blocks prefetched from far tiles. As a result, ZPP-HRO caches far prefetch blocks for a longer time duration than the other techniques. Thus, ZPP-HRO

performs better as compared to others.

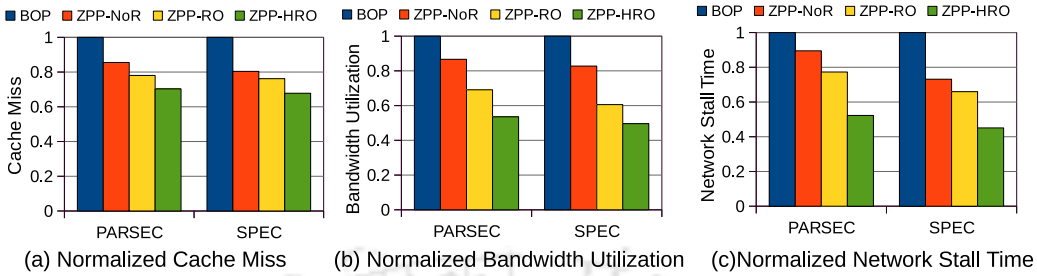

Figure 5.15: Comparison of ZPP with respect to BOP.

5.4.7 Comparison of ZPP with BOP

In ZPP, BOP [44] is used as the underlying prefetcher. Kindly note that ZPP can be used as complementary to any existing prefetcher. ZPP and its variants improve the underlying prefetcher's performance by (a) increasing prefetch aggressiveness, thereby reducing cache miss (b) reducing cache pollution by avoiding evicting any demand blocks from either L1 or L2 cache. This, in turn, reduces cache miss. As shown in Figure 5.15(a), ZPP-NoR, ZPP-RO, and ZPP-HRO reduce cache miss by 14.5%, 21.95%, and 29.7% in PARSEC benchmarks and 19.6%, 23.8%, and 32.2% in SPEC benchmarks, respectively as compared to BOP. Reduction in cache miss further reduces the number of network packets (cache miss and reply). Thus, ZPP reduces the underlying bandwidth utilization, as shown in Figure 5.15(b). Since NoC plays an important role in a TCMP system, reduction in cache misses and bandwidth utilization reduces network stall time as shown in Figure 5.15(c). Therefore, ZPP overall helps in increasing system performance in a manycore system.

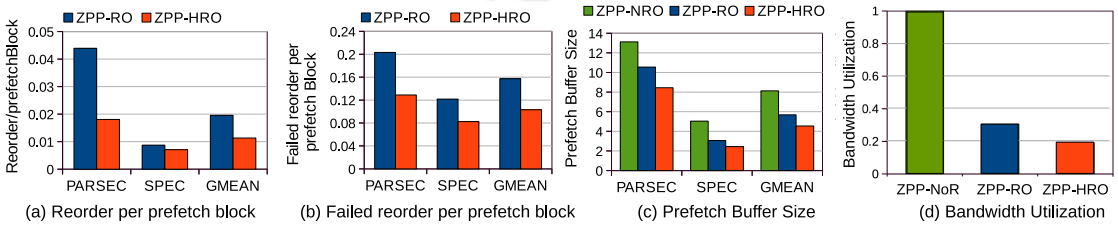

Figure 5.16: Performance of ZPP-NoR, ZPP-RO and ZPP-HRO.

5.4.8 Detailed Analysis

This section shows detailed analysis such as reorder per prefetch blocks, failed reorder per prefetch block, prefetch buffer size, bandwidth utilization and packet distribution in ZPP-HRO that switches between ZPP-NoR or ZPP-RO.

Figure 5.16(a) shows the number of internal reorder of prefetch blocks required in ZPP-RO and ZPP-HRO when an stale location in local bank changes into a non-stale state. In such a scenario, ZPP-RO tries to internally reorder all prefetch blocks in *L2mappedSets* if an stale location is available. On the other hand, ZPP-HRO reorders only those prefetch blocks that are prefetched from far tiles. ZPP-NoR, however, does not consider re-ordering of any prefetch blocks. As shown in the figure, the number of reordering done per prefetch block in ZPP-RO, ZPP-HRO is 0.019 and 0.011, respectively across all the benchmarks. Among the benchmarks, reorder per prefetch block is the least in SPEC as compared to PARSEC. This is because the number of stale blocks in PARSEC is less than that of SPEC. However, reorder per prefetch block being less indicates that the stale locations can sufficiently place most of the prefetch blocks.

Figure 5.16(b) shows the number of prefetch blocks that could not reorder because of the unavailability of stale blocks in local bank in ZPP-RO and ZPP-HRO. The parameter *Failed-Reorder per prefetch block* quantifies this. From the figure, we can observe that the number of failed-reorder per prefetch block across all the benchmarks is 0.16, 0.1 for ZPP-RO and ZPP-HRO, respectively. This shows that a sufficient number of stale block exists in the local bank that can hold prefetch blocks. Only a few fractions of prefetch blocks could not reorder internally which uses ZPP-NoR.

Initially, we have performed our experiments: ZPP-NoR, ZPP-RO and ZPP-HRO using an infinite buffer size to observe the maximum buffer size required by each of the techniques. Figure 5.16(c) shows that the maximum buffer size used by ZPP-NoR, ZPP-RO, and ZPP-HRO on average is 13, 10.5 and 8.45 in PARSEC and that for SPEC is 5, 3 and 2.5 for SPEC workloads. Since the number of stale blocks in SPEC benchmarks is more than PARSEC, the prefetch buffer is less used by the workloads. As a geometric mean across all the benchmarks, the prefetch buffer size required is 8.2, 5.7 and 4.5 for ZPP-NoR, ZPP-RO, ZPP-HRO, respectively. Hence, we use buffer size as 15 in all variants of ZPP.

Figure 5.16(d) shows bandwidth utilization in ZPP-NoR, ZPP-RO and ZPP-HRO normalized to ZPP-NoR. Since ZPP-HRO reduces cache miss by efficiently handling the prefetch blocks fetched from far tiles, the bandwidth utilization for this variant of ZPP is the

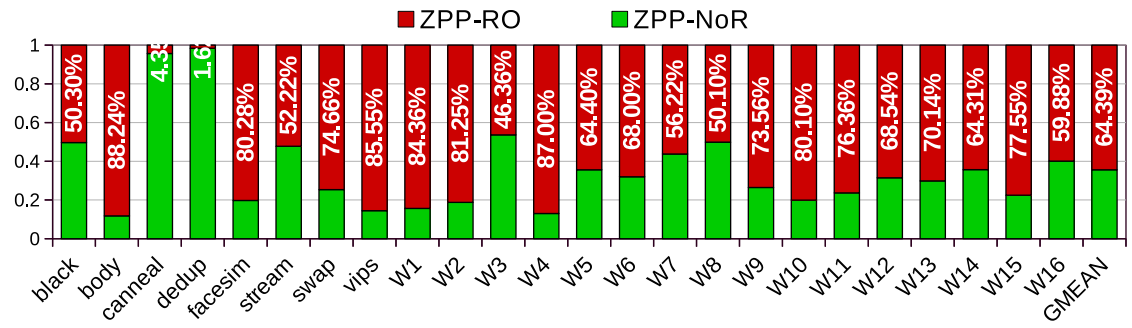

Figure 5.17: Distribution of prefetch blocks that used ZPP-NoR or ZPP-RO in ZPP-HRO.

least. Figure 5.17 shows the distribution of prefetch blocks that uses ZPP-NoR and ZPP-RO in ZPP-HRO. Since ZPP-HRO switches to and fro between these two techniques based on the hop count in 4x4 TCMP, we can observe that around across all the benchmarks, 64.4% of the prefetch blocks are reordered by ZPP-HRO while 35.6% prefetch blocks that are fetched from neighboring tiles uses ZPP-NoR. ZPP-RO tries to reorder all the prefetch blocks without prioritizing a prefetch block based on the hop count. Therefore the performance of ZPP-HRO is better than ZPP-NoR and ZPP-RO as shown in Table 5.4.

5.4.9 Sensitivity Analysis

In this section, we use various parameters such as prefetch aggressiveness, cache access latencies, and tracker size to analyze the sensitivity of ZPP to these parameters.

Table 5.5: Performance of ZPP (ZPP-HRO) on varying prefetch aggressiveness ($PrefAggr$) in multiples of 2.

$PrefAggr$	WS	$Exec\ Time$
1	1.673	2.78
2	1.871	2.561
4	2.19	2.44
8	1.81	2.512
16	0.998	2.897
32	0.92	3.23

A. Prefetch Aggressiveness (PrefAggr)

We have conducted experiments on varying PrefAggr from 1 to 16 as shown in Table 5.5. As PrefAggr increases, prefetch coverage also increases. But the performance of ZPP is

optimal when Pref Aggr is 4. Increasing PrefAggr further congests the network, thereby delaying demand and prefetch packets. Hence, network stall time increases, which further increases the execution time. When PrefAggr is less than 4, the prefetcher cannot cover the maximum cache miss. Thus, the execution time is sub-optimal for PrefAggr 1 and 2.

Tracker Variant	Hardware Overhead	Weighted Speedup	Exec Time
1024E-4W	5.67KB	2.198	2.42 sec
512E-4W	3.6KB	2.19	2.44 sec
256E-4W	2.57KB	1.768	2.586 sec
128E-4W	2.06KB	1.571	2.937 sec

Table 5.6: Performance of ZPP-HRO on different Tracker variants.

B. Tracker Variants

Table 5.6 shows the sensitivity of ZPP on different Tracker variants. We can observe that the optimal performance of ZPP is obtained when the Tracker variant is either 512E-4W or 1024E-4W. This is because the number of prefetch requests generated at a particular time instance can be easily handled using 512E-4W. For other cases, when the Tracker variant is less than the number of prefetch requests, it may compromise the application's performance as the number of prefetch blocks placed in the local bank reduces. On the other hand, if the Tracker variant is greater than the number of prefetch requests, its hardware overhead increases significantly. The hardware overhead of 1024E-4W is nearly 1.6x as that of 512E-4W. Also, as the tracker size increases, its access latency also increases. Therefore, to gain optimal performance from ZPP, we have used the variant 512E-4W that has an access latency of 2 cycles.

5.4.10 Hardware Analysis

Table 5.7 shows the extra hardware required for ZPP. The baseline system consists of L1 cache, L2 bank and a BOP prefetcher per tile. The caches and prefetcher require 320KB and 0.41KB per tile, respectively. As shown in Table 5.1, the size of the Tracker used for ZPP is 512E-4W which is obtained empirically as mentioned in Section 5.4.9. Each entry in Tracker requires 33 bits (V-1 bit, Tag-19 bits, L2set-9 bits, L2way-3 bits, nMRU-1 bit). Prefetch Buffer requires 532 bits per block (Block size + Tag bits + Valid bit) and in each L2 bank, *E-bit* per set (1 bit) and *P-bit* per block (1 bit) are used. Therefore, the additional hardware required per tile is 3.6KB, which is around 1.12% extra hardware overhead compared to

Table 5.7: Storage overhead per tile in ZPP for 8x8 TCMP.

Cache size per tile: **256KB (local bank) + 32*2KB (L1) = 320KB**	
Prefetcher per tile: **0.41KB**	
Component in ZPP	Total Bytes
Tracker	33 bits x 512 = 16896 bits
Prefetch Buffer	532 bits x 15 = 7980 bits
E-bit, P-bit	512 bits, 4096 bits = 4608 bits
Total Bytes for ZPP:	3.6 KB
Hardware overhead per tile as compared to a tile without ZPP: **1.12%**	

baseline. The area and power overhead of Tracker are 0.45% and 1.03%, per tile, respectively. We have also conducted experiments to observe whether increasing L1 cache size by $3.6KB$ (32KB+3.6KB $\approx$ 36KB) can provide system performance comparable to ZPP. We found that there is hardly any change in the execution time or weighted speedup when the L1 cache size is $36KB$ as compared to the baseline system with a $32KB$ L1 cache.

5.5 Chapter Summary

In this chapter, we proved that ZPP reduces cache pollution completely for TCMP architectures as compared to state-of-the-art prefetchers. For this purpose, it places prefetch blocks at local LLC bank that currently holds stale blocks, thereby avoiding replacing important demand blocks from the caches. The LLC bank being in the same tile avoids NoC communication, which reduces network stall time involved in fetching the block from a different tile. Experiments performed with real workloads showed significant improvement compared to the existing techniques.

CHAPTER

6

COPE: Identifying and Throttling Unwanted Prefetches in TCMPs

This chapter describes different scenarios where a prefetcher generates useless requests in TCMP architecture and proposes a novel technique to throttle them. When compared with the existing techniques, the proposed technique throttles useless prefetch requests, thereby improving system performance.

6.1 Introduction

Implementing prefetchers has certain challenges. To gain the maximum benefit from a prefetcher, it should accurately predict applications' future memory access patterns. However, it is a debatable topic as an ideal prefetcher is nearly impossible to build for any architecture. This results in prefetching some useless blocks along with the useful ones that waste network bandwidth and may cause cache pollution. Also, prefetch packets are additional packets that NoC routes among the tiles. Hence, if the prefetcher is not accurate, it may congest the network by generating some useless prefetches that can increase the packet transfer rate in the network. Thus, it may hamper system performance by congesting NoC and reducing prefetcher's effectiveness which in turn affects NST. Increasing NST directly contributes to increasing $AMAT_{TCMP}$. The techniques adopted in the previous two chapters (ECAP and ZPP) focused on improving $AMAT_{TCMP}$ by reducing the prefetcher-caused cache pollution. The chapters use different caching strategies for prefetch block placement such that an incoming prefetch block does not evict an important block from the cache. However, both ECAP and ZPP do not propose any new prefetch throttling techniques. Instead, it uses the underlying throttling mechanism to filter the useless prefetch requests.

$$Prefetch\ Accuracy = \frac{Useful\ PrefetchBlocks}{Total\ PrefetchBlocks} \tag{6.1}$$

Most of the existing technique uses the conventional parameter, prefetch accuracy, to identify whether a prefetcher is accurate or not. Prefetch accuracy refers to the fraction of prefetches that are accessed by the application out of the total prefetches brought to the cache. It is computed using Equation 6.1. An inaccurate prefetcher may fetch a large number of useless prefetch blocks to the caches. These useless prefetches might evict useful cache blocks and cause cache pollution. Hence, the chapter focuses on identifying whether a prefetcher is accurate or not. If it is inaccurate, any further requests generated by it are throttled, reducing the useless prefetch packets in the network. However, the conventional prefetch accuracy parameter has major limitations in TCMPs [38]. In TCMP, an inaccurate prefetcher may falsely generate prefetches that may harm the system performance because of the distributed nature of LLC. Similarly, there may arise a scenario where an accurate prefetcher is falsely throttled, considering it as inaccurate. Therefore, in this chapter, we show that prefetch accuracy in TCMP can create two major false positive cases of prefetching: (a) Under-estimation, Over-estimation problem, and (b) false feedback loop that can mislead a prefetcher in generating more useless prefetches. This chapter proposes COordinated Prefetching for Efficient resource sharing (COPE) [38] which shows that prefetch accuracy alone is not an appropriate parameter to throttle useless prefetches in TCMPs. COPE identifies such false positive cases of prefetching and throttles useless prefetch requests, thereby reducing $AMAT_{TCMP}$.

The key contributions of this chapter are as follows:

1. We experimentally prove that the conventional method of calculating prefetch accuracy is not suitable for TCMPs and generates false positive cases where a prefetcher generates a large amount of useless requests in the system.
2. We find that prefetch accuracy is necessary but not a sufficient parameter to throttle useless prefetches.
3. We propose COPE to effectively calculate prefetch accuracy and hence to reduce useless prefetch requests.
4. We compare COPE with state-of-the-art techniques [73, 74, 70, 44, 23, 106, 24, 25] for an extensive experimental analysis.

The rest of the chapter is organized as follows. Section 6.2 provides motivation behind proposing COPE. Section 6.3 describes COPE followed by the experimental analysis in

Section 6.4 and we finally summarize the chapter in Section 6.5.

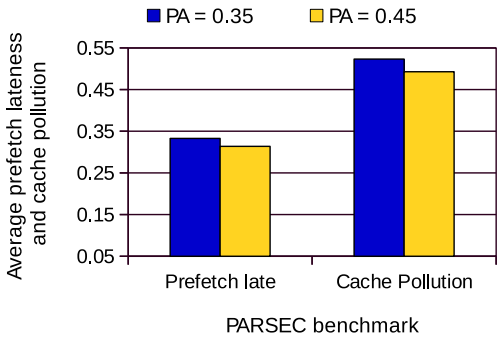

Figure 6.1: Prefetch lateness and cache pollution on varying prefetch accuracy (PA).

6.2 Motivation

As discussed in Chapter 3, we use gem5 [40] to model a stream prefetcher [57] on 64 core TCMP to explore the efficiency of prefetch accuracy parameters. The simulation parameters and benchmarks used are discussed in Section 3.3. Figure 6.1 shows prefetch lateness and cache pollution averaged across all the benchmarks in PARSEC suite. We observe the system behavior on two different prefetch accuracy; 0.35 and 0.45. Prefetch accuracy of 0.45 means that the application utilizes higher than 45% of the prefetch blocks. From the figure, we can observe that on an average 33%, and 31% of the prefetch blocks are late when the prefetch accuracy threshold is 0.35 and 0.45, respectively. Similarly, the cache pollution in both the cases is 52% and 49%, respectively. We can also observe that cache pollution is almost similar for both the prefetch accuracy threshold. This shows that even after increasing the prefetch accuracy from 0.35 to 0.45, the cache pollution is not effectively reduced. Hence, we can conclude that prefetch accuracy alone cannot control cache pollution, and some other factor also causes increased cache pollution. This motivated us to perform an in-depth study on prefetch accuracy parameter for TCMPs.

6.2.1 Anomaly in Prefetch Accuracy

Figure 6.2 and Figure 6.4 shows how the conventional prefetch accuracy parameter can mislead a prefetcher into generating useless prefetches in TCMPs. Figure 6.2 shows an $m \times n$ TCMP where the prefetcher at tile-$^0T_1$ fetches Y blocks from L2 bank at tile-$^3T_0$. Out of Y prefetch blocks only X ($X < Y$) prefetch blocks are useful (shown as X/Y). Similarly, the

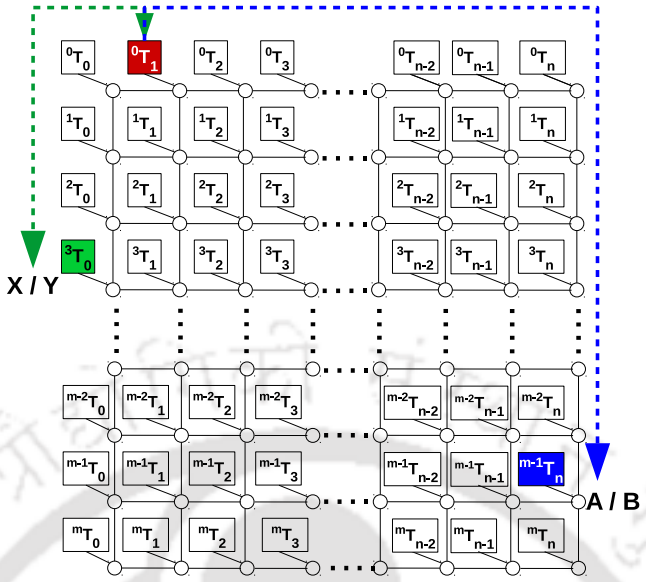

Figure 6.2: Under-estimation and over-estimation problem in mxn TCMP.

prefetch engine at tile-${}^0T_1$ prefetches B cache blocks from L2 bank at tile-${}^{m-1}T_n$. Out of B blocks, only A prefetch blocks ($A < B$) are useful (shown as A/B).

A. Under-estimation problem: Let us assume that $X = 0$, $Y = 5$, $A = 5$, and $B = 10$. The prefetch accuracy at tile-${}^0T_1$ is ≈ 0.33 ($(0 + 5)/(5 + 10)$). Since the accuracy is very low, the tile's prefetch requests can be throttled, considering that an inaccurate prefetcher causes cache pollution and NoC contention. But on careful observation, we can observe that prefetching from the L2 bank of tile ${}^{m-1}T_n$ alone is 50% (5/10) accurate while prefetching from the L2 bank of tile ${}^3T_0$ is completely inaccurate (0%). Hence, we can conclude that the cumulative effect of prefetch accuracy from tile ${}^3T_0$ is damping the efficiency of prefetching from tile ${}^{m-1}T_n$, which results in underestimating prefetching from the tile ${}^{m-1}T_n$.

B. Over-estimation problem: Let us assume another case where $X = 0$, $Y = 5$, $A = 8$, and $B = 10$. Using Equation 6.1, the cumulative prefetch accuracy at tile ${}^0T_1$ is 0.533 while the individual accuracy of prefetching from the L2 bank of tile ${}^{m-1}T_n$ is 80% and that of tile ${}^3T_0$ is 0%. Therefore, the cumulative effect of prefetch accuracy overestimates prefetching from tile ${}^3T_0$ due to tile ${}^{m-1}T_n$.

Figure 6.3 contains a plot of L2 bank at each tile vs. prefetch accuracy using PARSEC benchmark in a 4×4 TCMP. In the figure, Tot_PA is the total accuracy of the prefetcher in

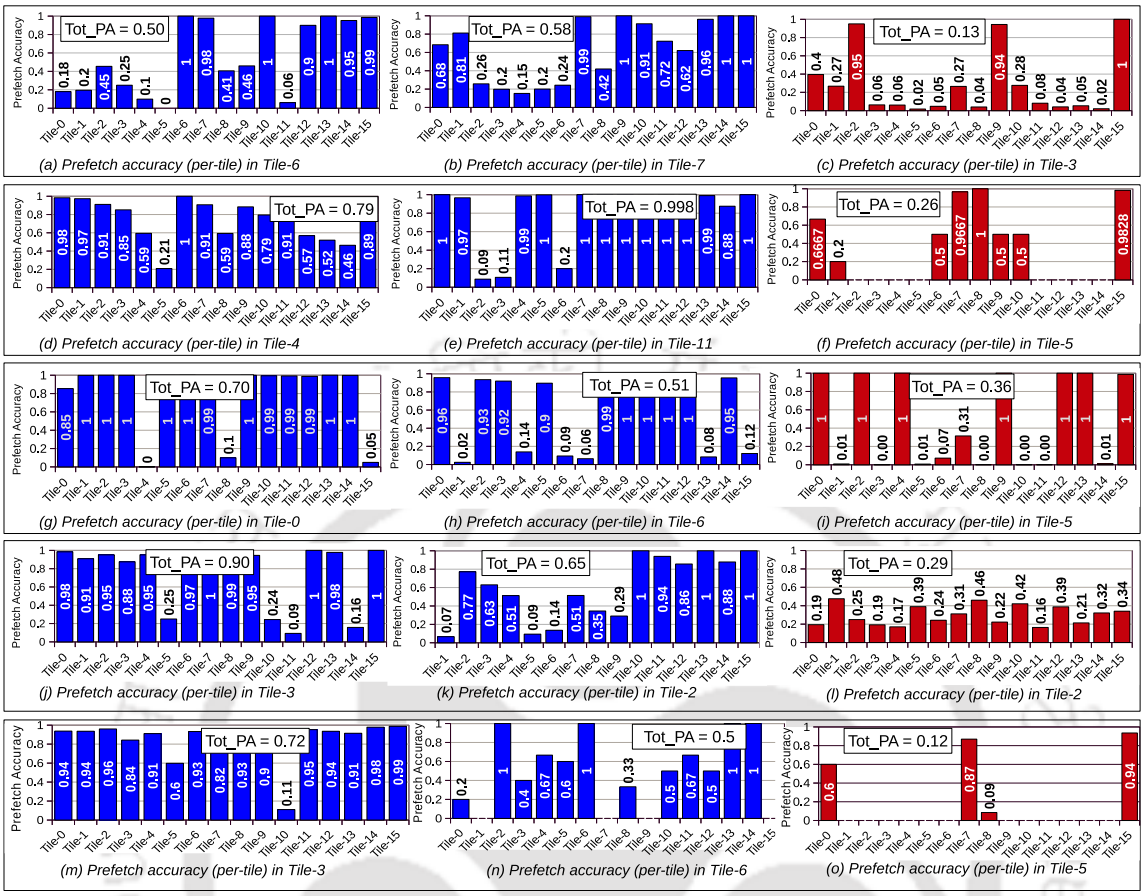

Figure 6.3: Experimental analysis of under-estimation and over-estimation in 4x4 TCMP using various benchmarks. [{(a), (b), (c)}: *vips*, {(d), (e), (f)}: *canneal*, {(g), (h), (i)}: *x264*, (j): *freqmine*, (k): *dedup*, (l): *body*, (m): *black*, and {(n),(o)}: *stream*]

tile-x ($0 \leq x < 16$). Each plot shows the accuracy of prefetching from the L2 bank of other tiles in the system. Over-estimation and Under-estimation caused by the prefetchers in a tile are shown by the blue and red plots, respectively. In Figure 6.3(a), we can observe that the Tot_PA at Tile-6 is 0.50 (50% accurate). But the individual accuracy of prefetching from the L2 bank of various tiles differs. Prefetching from tile $6, 7, 10, 12, 13, 14$ and 15 to tile-6 is more than 50% while prefetching from the other tiles is as low as 0%. Since the cumulative prefetch accuracy is high, the prefetcher in tile-6 continues prefetching even from those tiles whose per-tile prefetch accuracy is low (tile-$0, 1, 3, 4, 5$, and 11). We also observe that in Figure 6.3(c) the Tot_PA is very low (13%). But prefetching from the L2 bank of tiles $2, 9, 15$ is highly accurate. The cumulative prefetch accuracy being low, the prefetcher at tile-3 is throttled to reduce inaccurate prefetches. Hence, after throttling, the prefetch hits the application running in tile-3 got from tiles $2, 9$ and 15 results in a demand miss.

Thus, the prefetcher is not used properly because of under-estimation. A similar scenario is observed in other plots as well.

From the figure, we can also conclude that *canneal*, *vips* and *x*264 suffers severely from both under-estimation and over-estimation problem. *canneal* and *vips* have 25% tiles showing over-estimation problem and 8% tiles showing under estimation problem. Similarly for *x264*, 18% and 6% tiles has over-estimation and under-estimation problem, respectively. On the other hand, stream has 3.13%, 4.69% of the tiles suffering from over-estimation and under-estimation problem. In benchmarks like *facesim* and *swap* none of the two problems are observed. For *freqmine* and *dedup*, we have observed only over-estimation problem in 31% and 37% tiles, respectively and in *black* only one tile suffers from over-estimation problem as shown in Figure 6.3(m).

Over-estimation and under-estimation problems prevent prefetchers from achieving their actual performance. Over-estimation generates unnecessary network packets (tile ${}^3T_0$) that congest the network and LLC with useless prefetch requests. On the other hand, under-estimation hampers the system performance by restricting prefetching from those tiles whose individual prefetch accuracy is higher. Over-estimation and under-estimation are caused in those architectures where the underlying LLC is multi-banked and distributed. A prefetcher has complete information on the total number of blocks prefetched if it is a unified LLC. This makes the prefetch accuracy parameter definite for the LLC. But in TCMP, each tile has incomplete information on the amount of prefetching done from the other tiles. Hence, the conventional method of computing prefetch accuracy is imprecise for a multi-banked shared LLC system.

6.2.2 Cache Pollution

Prefetch accuracy is also delusive when the total prefetch request is less (when the denominator is small). For example: prefetch accuracy for 4/5 is 0.8 and prefetch accuracy for 40/100 is 0.4. Less number of prefetch request can be an indicator of the application behavior where the

1. application exhibits a good locality of reference.
2. application's working set completely fits in the cache.
3. prefetcher is not efficient in predicting the cache access patterns.

This chapter focuses on cases 1 and 2 as case 3 can be handled by sophisticated prefetchers. Since cache misses are used for training most of the prefetchers [15, 25, 24], case 1 and 2

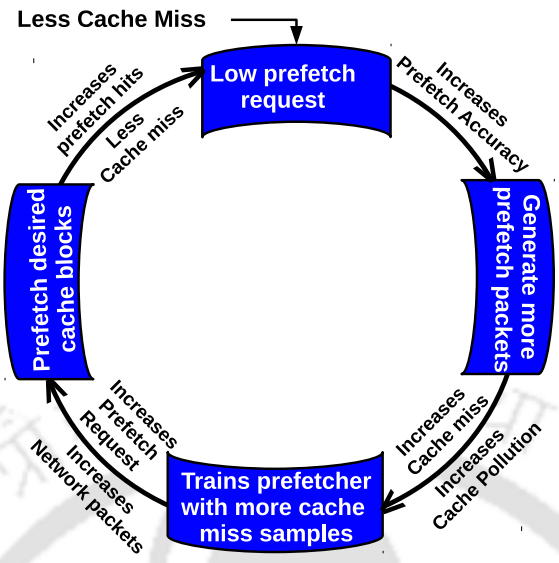

Figure 6.4: False feedback loop generated in TCMPs.

generate lower cache misses. Figure 6.4 shows the creation of false feedback loop in TCMPs due to the conventional prefetch accuracy parameter. The feedback loop is explained in two phases as the application proceeds: (i) Low prefetching phase and (ii) High prefetching phase.

i) Low prefetching phase: The right half of the false feedback loop depicts the low prefetching phase, where the prefetchers are trained with fewer cache misses. Since prefetchers are mostly trained on the miss trace, less cache miss results in training the prefetcher with fewer cache miss samples. Hence, the prefetcher generates fewer prefetch packets from a tile, resulting in increased prefetch accuracy for the application. An increase in prefetch accuracy gives an illusion that the prefetch blocks are experiencing a hit in the cache. Upon receiving such positive feedback on the prefetcher's performance, the prefetcher's aggressiveness increases [74, 75, 76, 113, 114]. In such a scenario, FDP [74] increases the prefetch aggressiveness by 2X where X is the prefetcher's present aggressiveness. But fewer prefetches indicate that the application seldom requires prefetching (less cache miss). Increasing prefetch aggressiveness may generate useless prefetch requests, which may increase cache pollution and NoC contention in TCMPs.

ii) High prefetching phase: The left half of the false feedback loop depicts the high prefetching phase. Initially, the prefetcher generates fewer requests in low prefetching phase. But prefetching for the same application increases after reaching this stage. Cache pollution

caused by unwanted prefetches in the low prefetching phase increases the number of cache misses for the application. An increase in cache miss increases the training samples for the prefetcher. Thus, it increases demand requests and corresponding prefetch requests from the respective tiles to hide the memory access gap. The demand request and prefetch request reduces the number of cache misses for the application. Reduced cache miss turns the application behavior back to the low prefetching phase, creating a false feedback loop.

The false feedback loop occurred because generating fewer cache misses; the conventional prefetch accuracy computation initially misled the prefetcher, creating a domino-effect in TCMPs. Thus, the traditional prefetch accuracy that throttles useless prefetches instead results in generating unwanted prefetches. This shows the unsuitability of the current prefetch accuracy for such architectures. It also indicates that prefetch accuracy is necessary but not sufficient to reduce unwanted prefetches in TCMPs. Another important scenario that may create the false feedback loop is when the prefetcher in a tile performs well to reduce cache misses. In such a scenario, the prefetch accuracy is high in the false feedback loop's low prefetching phase. But if the prefetcher does not cause any cache pollution, the prefetcher never enters the high-prefetching phase, breaking the loop. If the prefetcher performs well but causes cache pollution, it enters the high-prefetching phase.

To understand the false feedback loop's effect, we use FDP [74] and divide applications into (a) prefetch friendly and (b) prefetch un-friendly groups. Prefetch friendly applications are those for which the cache miss reduces when prefetching is enabled, whereas applications for which the cache miss increases are prefetch un-friendly as mentioned in Chapter 3 (Table 3.1). Among the PARSEC benchmarks, *canneal, dedup, freqmine, stream, vips, x264* are prefetch friendly benchmarks, while the rest are prefetch un-friendly benchmarks. For prefetch friendly applications, the problem of over-estimation and under-estimation is more pronounced than the false feedback loop. The false feedback loop is generated for those applications that are suitable for less prefetching or prefetching causes cache pollution. Hence, prefetch un-friendly applications are the ones where the false feedback loop is more pronounced. Figure 6.5 shows a snapshot of *swap* benchmark to show the false feedback loop formed after fast-forwarding for 10M cycles to remove the compulsory misses. In the figure, X-axis shows execution time, and Y-axis shows two parameters: Normalized cache miss (values ranging from 0 - 100) and Prefetch accuracy (in %). On multiplying cache miss by 1000, the actual miss count can be obtained.

The application's trace are observed to correlate the prefetcher's behavior with the cache

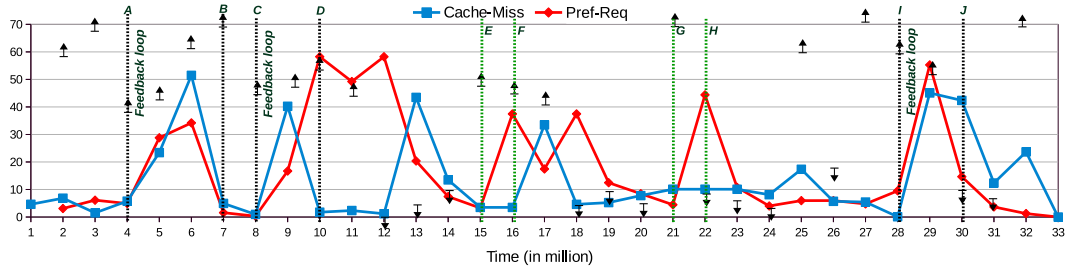

Figure 6.5: Formation of false feedback loop in swap benchmark. Black arrows indicate PA (↑ - PA above threshold, ↓ - PA below threshold). Label (A-J) shows various condition that may form a false feedback loop.

miss. The prefetcher is trained with the cache miss generated in the previous cycle. Based on the observed cache access pattern, blocks are prefetched in the next cycle. Therefore, at interval 0M-1M, no prefetch requests are generated, as shown in the figure. At the end of every time interval, the number of blocks prefetched, corresponding prefetch hits, prefetch lateness, and cache pollution is computed. Based on the computed metrics, FDP decides an appropriate prefetch aggressiveness for the next execution cycle. Different thresholds used by FDP are taken from the original paper [74]. To enter the low prefetching phase, a combination of *less cache miss, less prefetch requests* and *high prefetch accuracy* is a must. A low prefetching phase enters the high prefetching phase if the combination of *high cache misses, high prefetch requests* and *prefetch accuracy* above a threshold is satisfied. Otherwise, the loop breaks. The feedback loop is explained below:

1. (0M-1M): From the figure, we can observe that the cache miss at the end of the 1M cycle is relatively low ($<$ 10K), and the prefetcher is trained with the cache misses.
2. (1M-2M): The initial prefetch aggressiveness used by FDP is 2. For the execution cycle 1M-2M, FDP prefetches cache blocks. The prefetch accuracy obtained at the end of the 2M cycle is 65% (above threshold: 40%) as shown in the figure as an arrow with an increasing symbol. FDP, upon receiving positive feedback on the prefetcher's performance, increases the prefetch aggressiveness by 4 for the next execution cycle.
3. (2M-3M): During this execution cycle, the prefetch accuracy obtained is 76%. But the amount of prefetch requests generated at the end of the execution cycle is less. This may happen if the underlying prefetcher could not find an appropriate cache access pattern in the previous cycle, leading to fewer prefetch requests in the next cycle. Upon receiving positive feedback on the prefetcher's performance and the accuracy being above the threshold, FDP continues with prefetch aggressiveness of 4 for the

next cycle, i.e., 3M-4M.

4. (3M-4M): During this cycle, the prefetch accuracy is 41%. Hence, FDP continues prefetching with the same aggressiveness for the next execution cycle i.e., 4M-5M. This marks the start of the low prefetching phase of the feedback loop and it is shown as point (A) in the figure.
5. (4M-5M): At the end of the execution cycle, the prefetcher generated almost $5.4x$ times more prefetch requests and the number of cache misses also increases by 4.2x than the previous cycle. This is determined by FDP which shows cache pollution at the end of the execution cycle. Thus, FDP reduces prefetch aggressiveness by 2 for the next execution cycle. This interval marks the beginning of high-prefetching phase of the feedback loop.
6. (5M-6M): Increase in cache miss and prefetch accuracy continues until the next cycle.
7. (6M-7M): During this cycle, both prefetch requests and cache misses reduces. The generation of fewer cache misses resulted in the prefetcher to return at the low prefetching phase. Thus, the feedback loop is completed, which is marked as (B) in the figure.

Similarly, two feedback loops are formed, shown as (C-D) and (I-J) in the figure. The figure also shows two cases (E-F) and (G-H) where the prefetcher enters the low prefetching phase, but entering the high prefetching phase fails. For example, at point E, the prefetcher enters the low prefetching phase. But in the next cycle, i.e., (15M-16M), the prefetch requests are high and cache miss reduced. This shows that the prefetcher is successful in lowering cache miss causing no cache pollution. Thus it avoids entering the high prefetching phase. A similar scenario is also observed at (G-H). Experimentally we found that in prefetch un-friendly applications, 36% of prefetch requests are generated, and cache misses increased by 22% during the formation of false feedback loop as described. This statistic is obtained from FDP, which monitors cache pollution, blocks prefetched and prefetch hits obtained.

The difference between Subsection 6.2.1 and Subsection 6.2.2 is that under-estimation or over-estimation problem prohibits an application from benefiting from prefetching. On the other hand, in 6.2.2, the application does not require an aggressive prefetcher but prefetch accuracy being a dicey parameter rather generates useless prefetches. The over-estimation problem and false feedback loop generate unnecessary network packets and resource contention. The under-estimation problem inhibits the system from gaining maximum benefit out of prefetching.

6.3 Proposed Technique

We propose COPE that is added per tile to handle both the issues. In figure, 6.2 and figure 6.3, prefetcher at a tile generates a different amount of prefetch requests from the other tiles. Hence, prefetch requests and hits obtained from different tiles can be counted at per-tile granularity rather than having a common counter for total prefetch requests and prefetch hits. Thus, the prefetch accuracy obtained with per-tile granularity gives an accurate value of tiles from which prefetching should be throttled. Therefore, COPE calculates prefetch accuracy at per-tile granularity called *Individual Prefetch Accuracy (IPA)* to solve under-estimation and over-estimation problem. In TCMP, fewer cache misses indicate less inter-tile communication. Hence, we introduce another parameter *Inter-Tile Communication frequency* (ITC) to throttle useless prefetches. Thus, ITC is used to solve the false feedback loop problem where it avoids misleading the prefetcher in the low prefetching phase of the application. Therefore, the necessary and sufficient case for controlling unwanted prefetches in TCMP is a combination of two parameters: (a) IPA and (b) ITC.

$$\text{IPA}_x[i] = \frac{used\_prefetch_x[i]}{total\_prefetch_x[i]}; \quad \text{where } i \neq x \ \& \ 0 \leq i < N-1 \tag{6.2}$$

6.3.1 Calculation of IPA and ITC

COPE uses two vectors per tile: $IPA_x[0, \ldots, N-1]$ and $ITC_x[0, \ldots, N-1]$, where the TCMP comprises N tiles and $0 \leq x < N$. IPA indicates the accuracy of prefetching blocks from the LLC banks of another tile. For example: At tile-0, $IPA_0[2]$ indicates the accuracy of prefetching blocks from the LLC bank of tile-2 to the L1 cache of tile-0. Equation 6.2 is used to calculate IPA. It uses two additional vectors per tile, $used\_prefetch_x[0, \ldots, N-1]$ and $total\_prefetch_x[0, \ldots, N-1]$ to count useful prefetches and total prefetches, respectively. One bit per prefetch block is used to tag the useful prefetch blocks in the cache. We use $intertile\_comm_x[0, \ldots, N-1]$ to count the total number of misses between the tiles. For example, $total\_prefetch_0[2]$ means the total number of blocks prefetched from tile-2 to tile-0, and $used\_prefetch_0[2]$ means the number of prefetch blocks used in tile-0 out of the total blocks prefetched from tile-2. Similarly, $ITC_0[2]$ counts the rate of communication between tile-0 and tile-2 within a time epoch.

ALGORITHM 2: Prefetch throttling using COPE.

```
1  generatePrefetch_Request(f.dst)
2  begin
       Input: f.dst ⟵ destination tile.
3      //f.src = x
4      list<float> IPA_x: vector of individual prefetch accuracy.
5      list<int> ITC_x: vector of inter-tile communication.
6      if (ITC_x[f.dst] > T_comm) then
7          if (IPA_x[f.dst] > T_acc) || (total_prefetch[f.dst] == 0) then
8              generate prefetch request for current epoch
9          else
10             throttle prefetch request for current epoch
11     else
12         throttle prefetch request for current epoch (avoid low prefetching phase)
```

6.3.2 Throttling Useless Prefetches in COPE

Algorithm 2 is incorporated in each tile-x to throttle useless prefetches. It takes $f.dst$ as an input parameter, and based on the statistics collected from the previous epoch, the prefetch request is either generated or throttled. Since prefetch blocks are fetched from its home-bank, the home-bank/L2 bank is $f.dst$ and the requesting tile is $f.src$. The algorithm is described as follows:

1. COPE divides the complete execution time into equal time epoch, E, and all the vectors are initially initialized with value 0 at the start of every E_i.
2. At every epoch, whenever a prefetch block is fetched from a tile, the corresponding *total_prefetch* counter is incremented. On a demand request, if the block results in prefetch hit, the *used_prefetch* counter is incremented and upon a cache miss the *intertile_comm* entry gets incremented.
3. After an epoch expires, ITC and IPA at each tile are computed and profiled into the respective prefetchers. This is possible because present day prefetchers are programmable and support profiling run-time information [85, 143]. Also, the vectors *used_prefetch*, *total_prefetch* and *intertile_comm* are reset.
4. Within an epoch, whenever prefetcher generates a request, Line $6-12$ in Algorithm 2 determines if the prefetch request is generated or throttled at the source tile itself.

Since the first epoch E_0 has no prefetch statistics, COPE executes Algorithm 2 from the start of E_1. Based on the IPA and ITC values, Line 6 to 12 checks the problem mentioned in Subsection 6.2.1 and 6.2.2. COPE uses two thresholds: T_{acc} and T_{comm} to throttle inaccurate

prefetches. If the communication between the source tile and destination tile for the time epoch is above the threshold, T_{comm} (line 6), prefetch is generated in two cases. First, if the IPA between $f.dst$ and $f.src$ is above T_{acc} or second when the $total\_prefetch$ count is 0 (line 7). The first condition is checked to avoid over-estimation and under-estimation problems. The second condition holds true when in the previous epoch, prefetching has been throttled, but the cache miss count has increased in the current epoch. Since the execution phase changes as the application proceeds, in this case, prefetching should be resumed for accounting if prefetching is useful in the next time epoch. In line $11-12$, prefetches are throttled because ITC is below T_{comm}. This is done to avoid the formation of a false feedback loop for the low prefetching phase.

IPA and ITC are local information collected per tile. Hence, COPE does not produce any extra network traffic. Since the data structure MSHR maintains the missed block information that is outstanding, the ITC vector is incorporated on the MSHRs. The IPA vector is stored into the prefetch engines at each tile and T_{acc}, T_{comm} are decided experimentally as described in Section 6.4.8.

6.3.3 Implicit NoC Packets Generated in TCMP

Since distributed caches has to keep data coherent across all cache levels, we use directory-based MESI-CMP protocol [43] as described in Chapter 2. Whenever a prefetch request is generated from a tile, it is sent to the block's home-bank. If the prefetch block is not present in the home-bank, it is first fetched to the home-bank from the main memory. Upon placing the block in the home-bank, it is served to the requesting tile as packets. This involves a minimum of 12 flits transferred using NoC (request packet from L1 to home-bank + request packet from home-bank to main memory + block fetched from main memory to home-bank + block transfer from home-bank to L1 cache $= 1+1+5+5$). Similarly, when a block is present in S state in its home-bank and receives a prefetch request in M/E state, the maximum number of packets generated to fetch the block is 132 [1]. Thus, in TCMP, fetching a single useless prefetch block may generate at most 132 NoC packets.

Figure 6.6 shows the average number of coherence packets generated by stream prefetcher. In the figure, states are represented as NP: prefetch block is not present in L1 and L2 cache,

[1]Request packet sent from L1 to home-bank + directory in the home-bank sends an invalid request (Inv) to the L1 caches that have a valid copy of the block. Thus, maximum invalidation packets to be sent are equal to (L1 cache - 1) + directory receives acknowledgment (ACK) from such L1 caches. Thus for 8×8 TCMP, Inv+ACK packets generated is at most 2*(#L1 caches - 1) + block transfer from home-bank to L1 cache which is $1+2\times63+5 = 132$

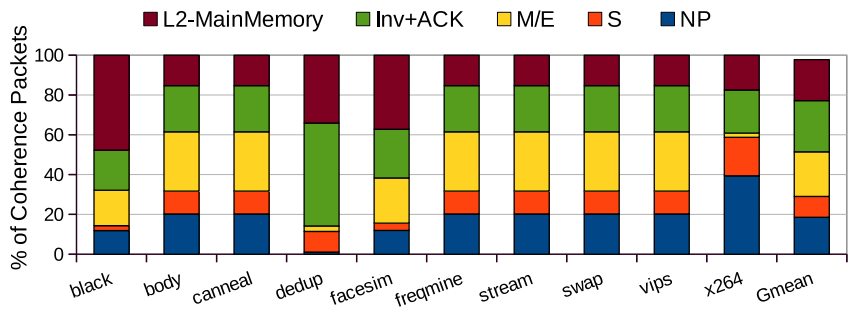

Figure 6.6: Classification of coherence packets generated in stream prefetcher.

Table 6.1: Simulation parameters for COPE

Processor	64, x86 cores OoO superscalar
L1 cache per tile	32KB, 2-way associative, 64B block
L2 cache per tile	256KB, 8-way associative, 64B block
L1 and L2 cache access time	2, 10 cycles
Prefetcher	stream
Prefetch Aggressiveness	2

S: prefetch block is requested in the shared state, Inv+ACK: prefetch block is requested in M/E state and the block is present in S state at L2 cache, L2-MainMemory: prefetch block is fetched from the main memory to its L2 home-bank. From the figure, we can observe that across all the benchmarks, Inv+ACK constitutes around 25.7% of the coherence packets. This is because 22% of the prefetch blocks are requested in the M/E state. Also, 18.5% of the prefetch blocks are in NP state. Hence, such blocks are fetched from the main memory (L2-MainMemory) and constitute around 20.6% of the total coherence packets. This shows that in TCMP, the prefetcher ignorantly congests the network to keep data coherent. However, reducing useless prefetch requests at the source tile directly reduces the implicit NoC packets. COPE indirectly reduces the implicit packets in NoC by throttling useless prefetches.

6.4 Experimental Analysis

COPE is modeled for a 64core TCMP using gem5 [40]. The power consumption in NoC is calculated using Orion 2.0 [128] and the proposed technique is implemented in Verilog HDL and synthesised 90nm technology using Synopsys DC. It is also evaluated on Xilinx Vivado Design suite and tested in ZYNC-7 series FPGA board.

6.4.1 Experimental Setup and Workload Description

The various simulation parameters used in this chapter are mentioned in Table 6.1 It is compared with existing works such as stream prefetcher (Baseline), FDP [74] and AppAware-D [73] because FDP and AppAware-D are exclusively proposed for TCMPs and use the conventional prefetch accuracy parameter to throttle useless prefetch requests. We provide a comprehensive evaluation of all the above techniques using IPC, cache miss-rate, average NoC packet latency, network stall time, prefetch accuracy, and prefetch timeliness. All the results are normalized to the Baseline. We analyzed the sensitivity of COPE using various design parameters: network size, prefetch aggressiveness, threshold values T_{acc}, T_{comm}, and time epoch. In COPE, L1 D cache misses are used to train the prefetcher. At every epoch of $1M$, the ITC and IPA information is profiled into the prefetch engine. This helps the prefetch engine to accommodate variations in the L2 access patterns from respective tiles. We have used PARSEC [41] benchmarks that spawn 64 threads across 64 cores in 8×8 TCMP. Since the threads share data among themselves, an aggressive prefetcher running on a thread may invalidate useful blocks from the L1 cache of another thread. Therefore, the negative effects of useless prefetches are more prominent in multithreaded benchmarks. Chapter 3 (Table 3.1) contains further description of PARSEC benchmarks.

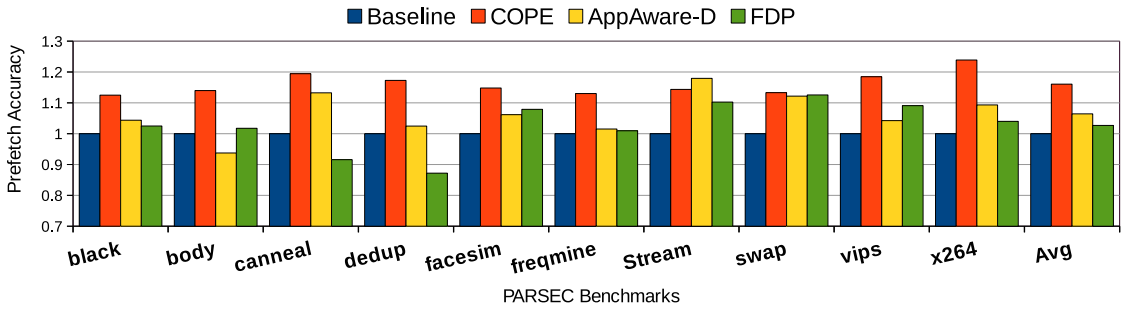

Figure 6.7: Comparison of prefetch accuracy normalized to Baseline.

6.4.2 Effect on Prefetch Accuracy

Figure 6.7 shows prefetch accuracy of various techniques normalized to baseline. The baseline has a prefetch accuracy of 1, and an accuracy greater than 1 shows an increase in the prefetcher's performance. From the figure, we can observe that the prefetch accuracy of COPE is 16% higher than the baseline. In contrast, in AppAware-D, FDP, the prefetch accuracy is 6.44%, and 2.7% higher than the baseline, respectively.

COPE has a higher prefetch accuracy across all the benchmarks except *stream*. In *stream*, COPE performs better than baseline and FDP, but the performance of AppAware-D is better than COPE by 3.6%. One probable reason for this behavior is that the data sharing among the concurrent threads in each tile is less for this benchmark. Hence, ITC is also less which results in less occurrence of feedback loop. Only 7.8% of the total tiles suffer from over-estimation and under-estimation problems in *stream* which is reduced by COPE. As stream's memory access pattern follows a regular fixed pattern and falls under the category of medium prefetch friendly, the access pattern is accurately predicted by stream prefetcher. Thus, AppAware-D rightly prioritizes the prefetch packets, which results in better prefetch accuracy for stream. But the conventional stream prefetcher used in AppAware-D cannot reduce the useless prefetch requests generated in TCMP. Thus, the unwanted NoC packet generated by the useless prefetch request delays both useful and useless prefetches, thereby reducing prefetch timeliness for the benchmark as observed in Figure 6.8.

Black does not communicate heavily with the other tiles. It uses a data parallel model and partitions its data among each thread such that a thread works locally within its partition. This limits inter-tile communication in *black* which is shown in Chapter 3 (Table 3.1) as low data sharing among the threads. Thus, over-estimation and under-estimation are comparatively less for the benchmark. Also, a thread suffering from false feedback cannot invalidate important blocks from other tiles. Therefore, the impact of the false feedback loop is also less. This can be observed in Figure 6.7 that the performance of COPE is least in *black* as compared to the other benchmarks. In some applications such as *canneal, dedup, vips* and *x264*, COPE performs better than the rest of the benchmarks. The percentage improvement of prefetch accuracy in *canneal, dedup, vips* and *x264* is 19.46%, 17.29%, 18.48% and 23.89%, respectively. COPE avoids forming a feedback loop by throttling prefetch requests for prefetch-unfriendly applications when the cache miss (ITC) is less.

6.4.3 Effect on Prefetch Timeliness

Prefetch timeliness also affects prefetch accuracy. Since COPE reduces useless prefetches, it reduces NoC traffic which directly reduces packet contention in the network. This increases the probability of prefetch packets reaching their respective destination tiles on time. Figure 6.8 shows the count of late prefetch blocks (packets) normalized to Baseline. The lower the number of late prefetch blocks, the higher is the prefetch timeliness. We observe that COPE has the least number of late prefetch blocks across all benchmarks, making it the best in

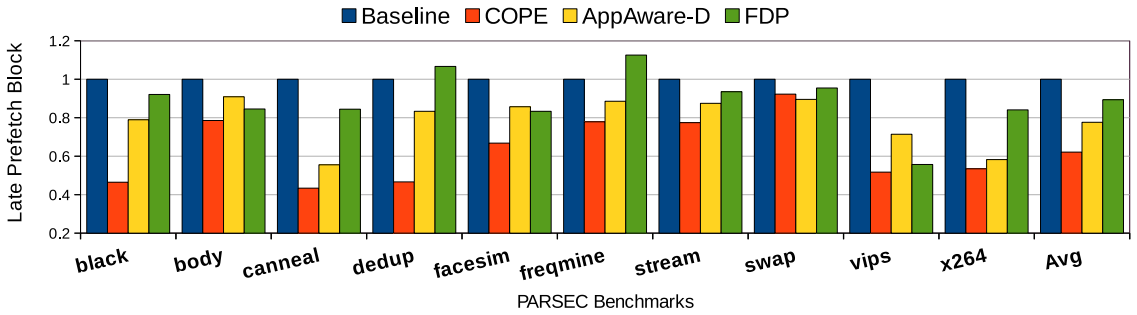

Figure 6.8: Comparison of late prefetch block count normalized to Baseline.

terms of prefetch timeliness. We observe that COPE, AppAware-D, and FDP achieve a reduction of 37.86%, 22.37%, and 10.66%, respectively, in late prefetch block count.

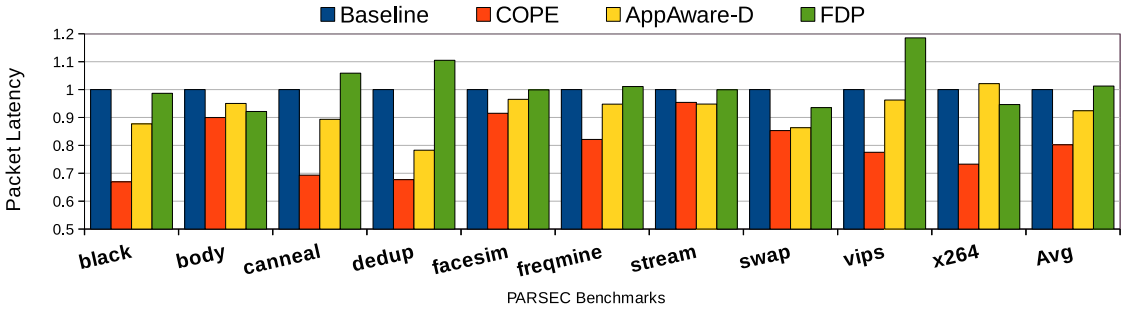

Figure 6.9: Comparison of average packet latency normalized to Baseline.

6.4.4 Effect on Average Packet Latency

We study the effect of COPE on NoC using average packet latency which is a measure of packet traversal rate in NoC. NoC with lower contention may experience lower packet latency. We also compare network load (includes cache miss, cache reply, and prefetch packets). Since link and routers are the primary components of NoC, the increase in network load reduces the routers' processing efficiency. Hence, the packet movement rate decreases, which has a direct impact on IPC of the application. Figure 6.9 shows that COPE reduces average packet latency by 19.78%.

6.4.5 Packet Distribution in NoC

Figure 6.10 shows packet distribution in Stream (Baseline), AppAware-D, FDP and COPE as an average across the PARSEC benchmarks. We can observe that COPE reduces useless

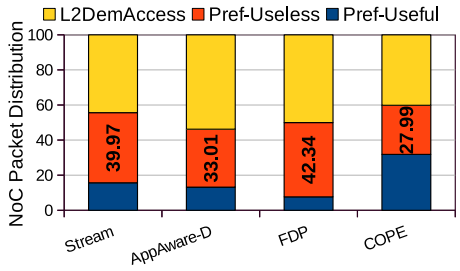

Figure 6.10: Packet distribution in NoC.

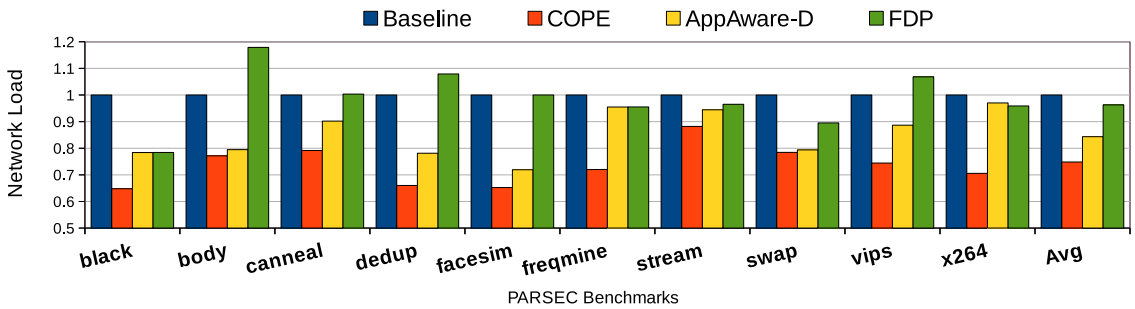

Figure 6.11: Comparison of network load normalized to Baseline.

prefetch packets by 11 percentage points compared to stream. Since COPE throttles useless prefetch blocks, the cache pollution and cache miss reduces, as explained in the next subsection. Thus out of the total prefetch blocks, the number of useful prefetch blocks increases. Due to an increase in useful prefetch blocks, some of the demand blocks are covered by the prefetcher. As a result of this, the number of L2 demand access reduces by 4.2 percentage points. Figure 6.11 shows that COPE, AppAware-D on average, reduces network load by 25% and 16%, respectively. While in FDP, the network load slightly reduces by 3.71%. This figure justifies the increase in average packet latency for benchmarks *canneal, dedup* and *vips* in FDP.

However, COPE's performance is limited by NoC that incurs delay during packet transfer from source tile to destination tile. Figure 6.8 shows that though COPE increases the prefetch timeliness of the underlying prefetcher, it cannot eliminate the late prefetch blocks because of LLC's distributed nature. Hence, NoC plays an important role in inter tile communication. Thus, a useful prefetch block may get converted into useless if it does not reach the requesting tile on-time. However, this is a debatable topic as an ideal prefetcher is nearly impossible to achieve [15]. This is justified as Figure 6.10 shows that around 28% of the prefetch packets are still useless in COPE.

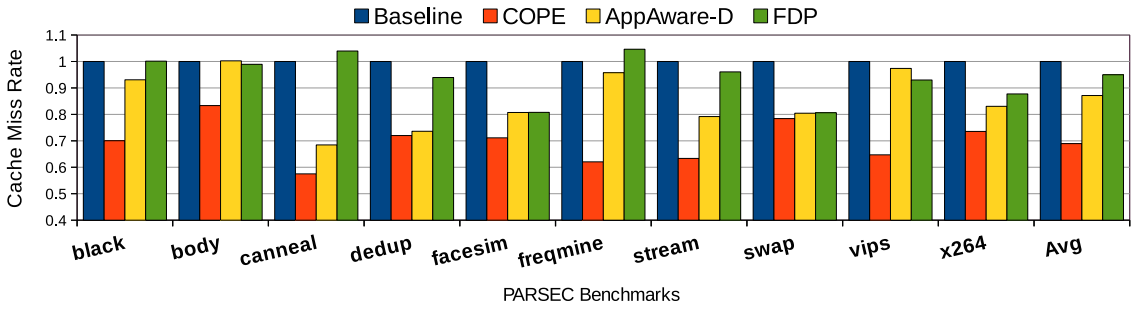

Figure 6.12: Comparison of L1 miss rate normalized to Baseline.

6.4.6 Effect on L1 Cache Miss

Figure 6.12 shows that COPE, AppAware-D and FDP reduces L1 cache miss by 31%, 13% and 5%, respectively compared to the baseline. Prefetch traffic consists of useful and useless prefetches. Useful prefetches help in reducing the network stall time involved during cache misses. Thus, the prefetch block hides the underlying NoC and memory access latency, thereby improving application's performance. Had prefetching being disabled, the useful prefetch blocks would have been fetched as demand blocks. Thus, in such a case, cache miss and AMAT_{TCMP} increases. On the other hand, useless prefetches are unwanted NoC traffic. As a side-effect of useless prefetches, it may increase NoC traffic by evicting important demand blocks from the cache. The implicit NoC traffic (coherence packets) generated by useless prefetches adds up to the overall NoC traffic. Since COPE increases the prefetcher's accuracy and timeliness of the prefetcher, it reduces cache miss and NoC traffic, which would reduce $AMAT_{TCMP}$. This is justified from Figure 6.13 which shows the reduction in network stall time is 20.5% in COPE than the baseline. This is mainly for the reduction of useless prefetching from different tiles. Hence, COPE reduces $AMAT_{TCMP}$ by 24.14% across all the benchmarks.

6.4.7 Effect on Instructions Per Cycle

Table 6.2 shows that COPE increases IPC by 12.63% with maximum IPC improvement of 24.96% in *vips*. Since COPE reduces the false cases of prefetching, it increases the prefetchers' accuracy and timeliness by carefully monitoring the useless prefetch request in TCMPs. This reduces network contention as well as cache pollution. Reduction in network contention resulted in fetching cache blocks faster, thereby reducing network stall time. Therefore, the negative aspects of useless prefetches are avoided in COPE, which

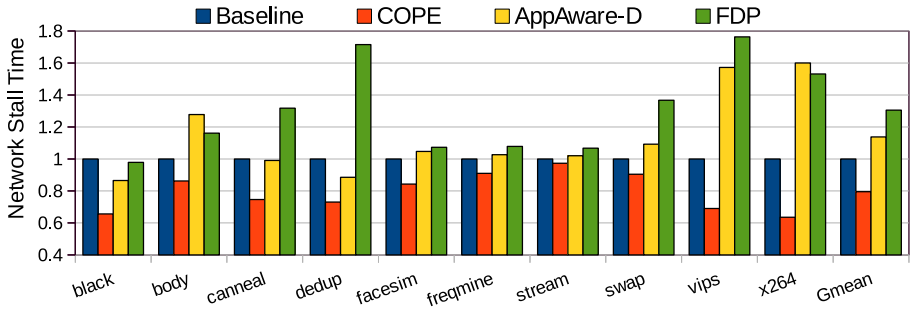

Figure 6.13: Comparison of network stall time normalized to Baseline.

Table 6.2: Effect on IPC (%) in different techniques compared to Baseline.

	black	body	canneal	dedup	facesim	freqmine	stream	swap	vips	x264	Average
COPE	15.62	9.50	26.80	4.83	4.88	5.91	1.67	4.97	24.96	10.26	12.63
AppAware-D	6.66	-10.18	22.10	1.68	-7.32	-2.00	-2.78	1.58	1.02	6.25	2.08
FDP	1.65	4.19	-22.31	0.54	-6.10	-7.55	1.11	2.66	16.32	8.13	0.53

improves the application's IPC. In the table, we can also observe that the IPC improvement in AppAware-D and FDP is a meager 2.08% and 0.58%, respectively.

6.4.8 Sensitivity Analysis

A. Network size: To observe COPE's performance in 4×4 TCMP, we have conducted a sensitivity analysis on the network size. Figure 6.14 shows that COPE reduces the L1-cache miss-rate by 11.93% in a 4×4 TCMP. When compared with 8×8 TCMP, we have observed that the performance of COPE is better at 8×8 than 4×4 TCMP. In a smaller size network, congestion generated on a region due to over-estimation, under-estimation, and false feedback loop disperses faster to the nearby nodes than a large network. Thus, in 8×8 TCMP, COPE reduces the average cache miss rate by 31% as shown in Figure 6.12. This shows that the average reduction of cache miss rate in a larger network of size 8×8 is more than that of a smaller network of size 4×4.

B. Prefetch aggressiveness: Figure 6.15 shows COPE's performance on varying prefetch aggressiveness (degree) from 1 to 16 in multiples of 2. It is represented as $pref\_deg\_x$ where x is the aggressiveness of the prefetcher used throughout the simulation. From the figure, we can observe that COPE's performance is better with prefetch aggressiveness as 2 or 4 with a similar trend in the execution time (least). When prefetch aggressiveness is more than 4, unnecessary prefetch packets are fetched, reducing the prefetcher's efficiency. For

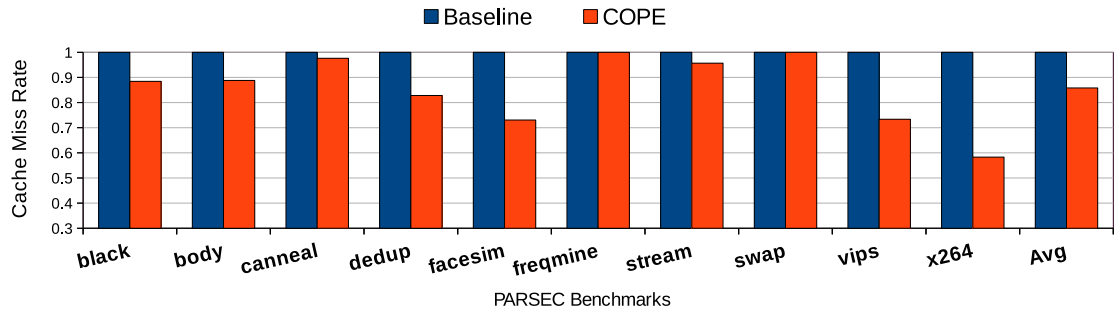

Figure 6.14: Performance of COPE in 4x4 network.

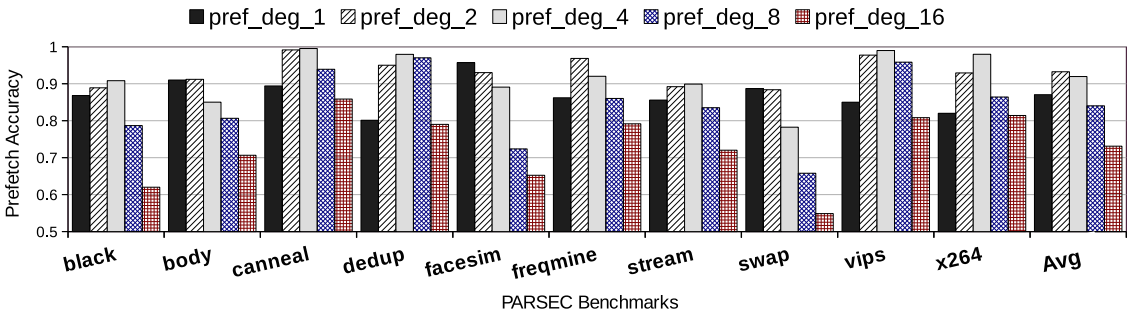

Figure 6.15: Performance of COPE on varying prefetch aggressiveness.

our experiments, we have used the prefetch aggressiveness as 2.

C. T_{acc} and T_{comm}: Table 6.3 shows COPE's behavior on changing the thresholds, T_{acc} and T_{comm}. The value of both the thresholds is varied from 0.2 to 0.5. From the table, we can observe that when T_{acc} and T_{comm} is too low, i.e., 0.2 each, the prefetch accuracy drastically reduces. This is because inaccurate prefetches are not throttled accurately. Hence, the inaccurate prefetches degrade prefetch accuracy. On the other hand, when T_{acc} and T_{comm} is too high, i.e., 0.5 each, COPE restricts the prefetcher in generating any requests. As a result of this, the prefetcher under-performs. We can also observe that COPE performs well when T_{acc} is 0.3 and T_{comm} is either 0.3 or 0.4. For some of the applications, COPE also performed well at T_{acc} as 0.4 and T_{comm} as 0.35 (not shown in the Table). Therefore, we use an average of these cases for our experiments with T_{acc} as 0.35 and T_{comm} as 0.38.

D. Time epoch: We have also experimented with different epoch intervals for COPE: 10^3, 10^4, 10^6, and 10^8. When epoch is 10^3, COPE became very restrictive as not enough

Table 6.3: Analysis of prefetch accuracy (PA) on varying T_{acc} and T_{comm}.

T_{acc}	T_{comm}	Overall PA	T_{acc}	T_{comm}	Overall PA
0.2	0.2	0.548	0.4	0.2	0.838
0.2	0.3	0.629	**0.4**	**0.3**	**0.946**
0.2	0.4	0.658	0.4	0.4	0.825
0.2	0.5	0.752	0.4	0.5	0.725
0.3	0.2	0.816	0.5	0.2	0.571
0.3	**0.3**	**0.932**	0.5	0.3	0.684
0.3	**0.4**	**0.928**	0.5	0.4	0.528
0.3	0.5	0.783	0.5	0.5	0.473

statistics are collected, thereby throttling most of the prefetches. On the other hand, when the epoch interval is 10^8, the cache miss increased compared to the baseline due to cache pollution. COPE produces optimal performance gain with an epoch interval of 10^6.

6.4.9 Hardware Analysis

Table 6.4 gives a review of the extra hardware required to implement COPE in an 8×8 TCMP. The baseline consists of caches (L1 and L2) per tile and a stream prefetcher. The prefetcher contains four streams with two prefetches per stream. Each prefetch entry consists of a valid bit, tag bits (8 bits), prefetch request type (load/store), and stride (6 bits) that sums up to be 2B. Hence, the prefetcher requires a total of $16B$ per tile. COPE requires four sets of integer vectors and one floating pointer vector with single precision for IPA. Since each epoch is 1 million cycles, a 20-bit register is enough to avoid overflow for the integer vectors. Two additional 32-bit registers are used to hold T_{acc}, and T_{comm}, and a prefetch bit per L1 tag-entry is used to tag the prefetch blocks in the cache. The hardware overhead is dependent on TCMP size. For 2×2, 4×4 NUCA, the hardware overhead of COPE is just $0.04, 0.09\%$, respectively. On the other hand, FDP has an additional hardware requirement of 1102B per tile (512B/L2 cache entry for storing prefetch-bit per tag, 512B for pollution filter, 26B for the static thresholds, 36B for nine counters, and 16B for per MSHR entry). As compared to FDP, COPE has 88.3%, 73%, and 12.2% less area overhead for 4 cores, 16 cores, and 64 core TCMP, respectively.

The LUT overhead involved in the proposed design's addition has a logic overhead of around 0.316% and area overhead of 0.29% for a 8×8 TCMP. We observed that the implementation of the extra logic for COPE does not violate the timing constraints of 1ns (the NoC runs at 1GHz). This is noticed as both the design has the same positive slack. Hence, COPE does not affect the timing design and it is not on the critical path of

Table 6.4: Additional hardware overhead of COPE per tile for 8x8 TCMP.

Cache size per tile: **256KB (L2 bank) + 32*2KB (L1) = 320KB**	
Stream prefetcher per tile: **16B**	
Component in COPE	Total Bytes
IPA	4B x 64
ITC	20bits x 64
intertile_comm	20bits x 64
used_prefetch	20bits x 64
total_prefetch	20bits x 64
T_{acc} & T_{comm}	4B x 2
prefetch-bit per tag-entry in L1 cache	512 blocks = 64B
Total Bytes for COPE:	968B
Hardware overhead per tile compared to the baseline: **0.29%**	

the processor. COPE requires an additional power of 4mW due to the switching activity involved in the counters. On the other hand, the extra logic's static power consumption is dominated by the static power consumption of the L1 cache. Therefore, the new design does not affect the static power consumption of the cache. Synthesizing our proposed technique in hardware showed that the overhead of additional counters is very negligible in terms of memory overhead, sustainable in terms of logic (LUT), and safe in terms of timing and power constraints. The power results from Orion 2.0 [128] showed that the dynamic power consumption of NoC reduces by 18.56% as compared to the baseline. This makes COPE a feasible design criterion for all architectures with a shared banked LLC like TCMP.

6.5 Chapter Summary

In this chapter, we revisited prefetching in TCMPs and identified that the conventional prefetch accuracy generates false positive cases of prefetching that increase useless prefetches in the system. To resolve the issue, we propose a light-weight technique, COPE, that redefines prefetch accuracy for TCMP architectures. It uses two parameters: inter-tile communication frequency and prefetch accuracy on per-tile granularity to reduce the useless prefetches. COPE also reduces the dynamic power consumption of NoC by reducing useless prefetch packets generated from the false cases of prefetching. On experimenting COPE with existing technique, it significantly improves system performance.

CHAPTER

7

FlitZip: Effective Packet Compression for NoC Packets

This chapter proposes an efficient on-chip packet compression technique to reduce network load in NoC based TCMPs. The compression technique uses a lightweight compressor and decompressor module to reduce the area and power consumption in NoC. For measuring its efficacy, we compare the proposed compression technique with state-of-the-art packet compression techniques.

7.1 Introduction

The previous chapters dealt with improving $AMAT_{TCMP}$ by throttling useless prefetch requests and reducing prefetcher-caused cache pollution. In TCMP, the time required to service a cache miss is a major concern, and it is greatly dependent upon how fast NoC services the reply packets. In this thesis, we reduce network stall time using two techniques: prefetching [15, 14], and on-chip packet compression [17, 21, 22, 32, 33, 34]. Reduction in cache pollution reduces NoC packets which in turn reduces the network stall time. Hence, the average memory access time, $AMAT_{TCMP}$ reduces. On the other hand, packet compression exploits the data redundancy within network packets, shrinking their size. Reduction in packet size reduces network traffic, wastage of link bandwidth, and NoC power consumption. Reduction in the network traffic reduces the network stall time, thereby reducing the average memory access time required while fetching cache blocks from a different tile. Also, NoC accounts for around 28% of tile-power in Intel Teraflop chip [29] and 36% in MIT RAW chip [5]. Hence, the power consumed by NoC is also taken care of by on-chip packet compression techniques.

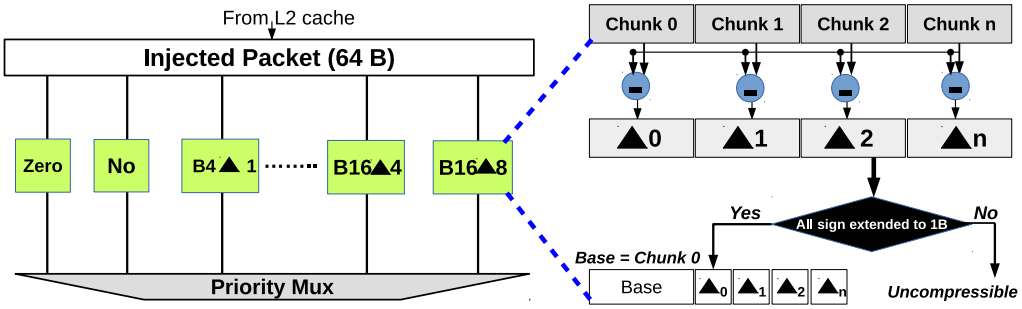

Figure 7.1: Compressor unit in existing delta compression in NoC.

As mentioned in Chapter 1, that packet compression requires an efficient compressor and decompressor units that is provided by delta compression techniques [17, 21, 22]. It uses the fact that the relative deviation between the data values in a cache block varies slightly. Hence, a part of the cache block (X bytes) can be represented as a common base of P bytes and rest of the block is represented as an array of differences: $\{\Delta_1, \Delta_2, \Delta_3, \ldots, \Delta_n\}$ where $n = X/P$, from the base. Packets are compressed at Network Interface (NI) only. Caches and DRAM are unaware of any underlying packet compression and function as a conventional system. Figure 7.1 shows eleven combinations of de/compressors used in NoΔ and DISCO [17, 21, 22]: B16Δ8, B16Δ4, B16Δ2, B16Δ1, B8Δ4,. . ., B4Δ1. The term BxΔy means that the packet is compressed with a base of size x bytes and Δ_i of y bytes. Since a 64 byte packet can be represented as four 16 *byte* chunks or eight 8 *byte* chunks or sixteen 4 *byte* chunks, the base in these cases are of size 16 bytes (B16), 8 bytes (B8) and 4 bytes (B4), respectively. For a particular base say B8, the size of Δ can be represented using 1 byte (B8Δ1), 2 bytes (B8Δ2) or 4 bytes (B8Δ4).

We study the existing delta compression techniques [17, 21] in general and uncover a few of its limitations here. (1) The size of base ($4 \leq B \leq 16$) and Δ ($1 \leq \Delta \leq 8$) are always represented in bytes. Hence, existing techniques can never compress a packet with a smaller base (< 4 bytes) or Δ (< 1 byte). Consider a 64 bytes packet with data consisting of repeated 0xFF values. The compressed packet would contain a base and Δ_i is 0, which is minimally represented in a byte. But 0 can be represented in less than a byte, which is not possible in the existing techniques. (2) The de/compressor units are bulky and are not area and power efficient. (3) The metadata of a compressed packet is represented using an encoding table containing all combinations of (B, Δ), encoding bits and priority bits. The size of the table is 250 bytes[1]. (4) The base of the compressed packet is appended in the

[1]Encoding table stores base (max 16 bytes), Δ (max 8 bytes), encoding (4bits) and priority (4bits) at each tile.

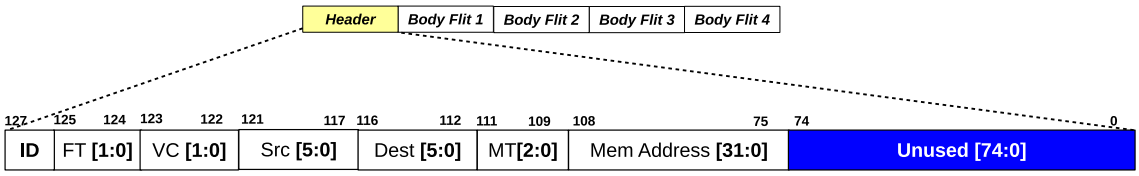

Figure 7.2: Content of head flit.

body flits. Hence if $B16$ is used for compression, the minimum packet size is 16 bytes plus Δs. (5) Also, only the encoding scheme of a compressed packet is stored in the head flit. But a head flit has around 50% unused bits as shown in Figure 7.2. Thus, the remaining bits are underutilized. Thus, we propose a novel and lightweight lossless packet compression technique, *FlitZip*, that compresses packets at flit-level granularity.

The key contributions of the chapter are as follows.

1. We identify the limitations of existing packet compression techniques [17, 21, 22].
2. We propose FlitZip that identifies data patterns within a flit and compresses packets on a per-flit basis, thereby achieving a higher flit compression ratio.
3. We propose a lightweight de/compressor units that provides significant savings in area and power.
4. We perform extensive experiments with 397 workloads from PARSEC [41] and SPEC [42] benchmarks to validate our claims.

The chapter is organized as follows. The motivation behind this work is presented in Section 7.2. Section 7.3 presents the proposed technique, FlitZip followed by experimental analysis in Section 7.4 and the chapter is finally summarized in Section 7.5.

7.2 Motivation

The motivation behind FlitZip is drawn from three observations as explained below.

1. Unused bits in the head flit: Figure 7.2 shows a head flit in 8x8 TCMP with a 128 link bandwidth. The system uses 4GB main memory with a 64B packet size. A request packet (REQ) is represented using one head flit, and a reply packet (REP) is represented using a head flit and four body flits. As shown in the figure, the head flit consists of various control information such as packet number (ID), source tile (Src): that generates cache miss, destination tile (Dest): block's home-bank, flit type (FT): head/body/tail, VC number: internal buffer in a router used to store a flit, Message Type (MT): REQ/REP/coherence packets and missed block address (MEM Address). Out of the total bits in the head flit,

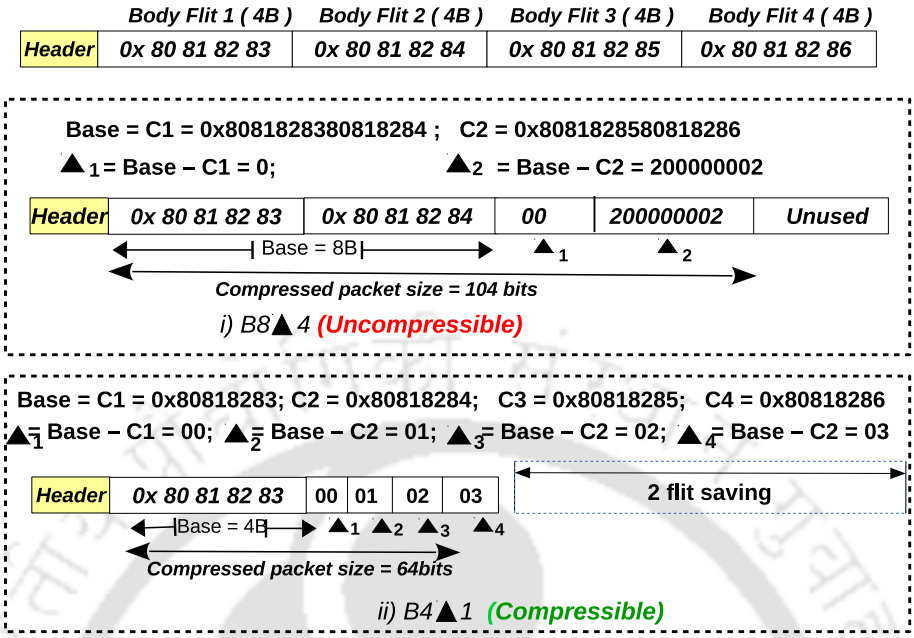

Figure 7.3: Simultaneous execution in NoΔ and DISCO.

75 bits remain unused. Kindly note that if any of the field changes, the size of the unused portion also changes.

Existing packet compression technique that uses delta compression such as NoΔ (nodelta) [17] utilizes only 4 bits from the unused portion to store the encoding information of a compressed packet. Thus, the remaining unused portion is not utilized efficiently. If the unused portion can store additional metadata information of a compressed packet, we can achieve a highly compressed packet than the existing delta compression techniques. This serves as the first motivation in proposing FlitZip.

2. Simultaneous computation in existing delta compression techniques is costly: Figure 7.1 shows that the state-of-the-art delta compression technique uses different de/compressor modules that are executed simultaneously on an incoming packet to determine whether a packet is compressible or not. Figure 7.3 shows a representative example of simultaneous computation in B8Δ4 and B4Δ1 modules. The example assumes a packet size of 16 *bytes*, and each flit size is 4 *bytes*. We can observe that only the last 4 bits from the LSB differ among the flits. Figure 7.3(i) shows the computation in $B8\Delta4$ module. With base B8, the packet is divided into two chunks of 8 bytes: C1 and C2. NoΔ uses the first chunk, C1 as the base, and Δ_i is computed by subtracting each chunk from the base: Δ_1 is 0, and Δ_2 is 200000002. However, with $B8\Delta4$, the compressed packet size does not reduce the flit size as Δ_2 cannot be represented in 4 *bytes*. Thus, the packet is incompressible

with $B8\Delta4$. Similarly, Figure 7.3(ii) shows that computation for B4Δ1. Out of all (B, Δ) combinations, the packet is compressible with B4Δ1 only with two flits saving. Thus, existing delta compression techniques select one best by applying all the combinations on the incoming data packet to achieve the maximum compression ratio.

Though simultaneous computation gives the best compression ratio, these modules include several adders, buffers, etc., in the de/compressor unit. This is a challenge for an area and energy-efficient design. For example, to calculate Δ_i, the bit-width of the subtractors in the de/compressor units must be equal to the base (in bytes). Since the base size varies between 16 bytes to 4 bytes, the existing techniques require bulky de/compressor units for efficient packet compression. In addition to these, multiple modules that check the sign extension of Δ_i and the number of subtractors are equal to n where $n = \frac{Block\ size}{Base\ size}$. This motivated us to propose lightweight de/compression units, which led to the proposal of FlitZip.

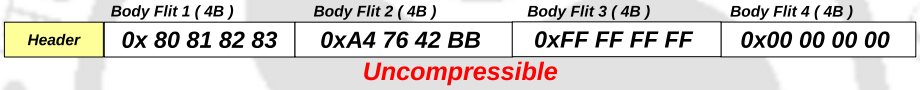

Figure 7.4: Example showing delta compression is unable to compress.

3. Existing delta compressions cannot reduce redundancy in packets effectively: As the size of (B, Δ) is in bytes, it enforces to represent even the smallest value for Δ in a byte. Moreover, the first chunk of a packet is used as a base for compression. Hence, if the first chunk varies widely from rest of the chunks, existing delta compression techniques cannot compress the packet. We call this as *intra-packet* data pattern. This restricts packet compression as explained using Figure 7.4. If the base used is either B8 (0x80818283A47642BB) or B4 (0x80818283), the packet is incompressible. Hence, the packet is incompressible using any combination of (B, Δ). But from the figure, we can observe that data within flits 1, 3, and flit 4 shows low deviations.

Need for a new compression technique: In Figure 7.4, we can observe that if each flit's data value is divided as 1 byte chunks, the variations within the chunks are very less. The chunks of flit 1 (80, 81, 82, 83) differs by 2 bits only while the chunks of flit 3 (FF, FF, FF, FF) and flit 4 (00, 00, 00, 00) are all the same. On the other hand, the chunks of flit 2 ($A4$, 76, 42, BB) differ by 8 bits. Since the existing technique searches for intra-packet data patterns, despite having regular patterns in all the flits except flit 2, the packet is

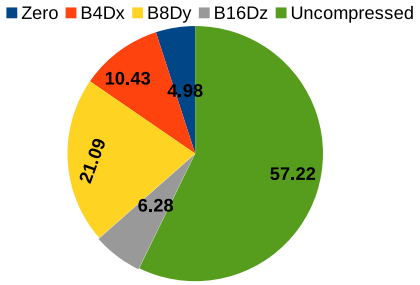

Figure 7.5: Data patterns observed in existing techniques [17, 21, 22]. In B4Dx, B8Dy and B16Dz, x=1/2, y=1/2/4; z=1/2/4/8.

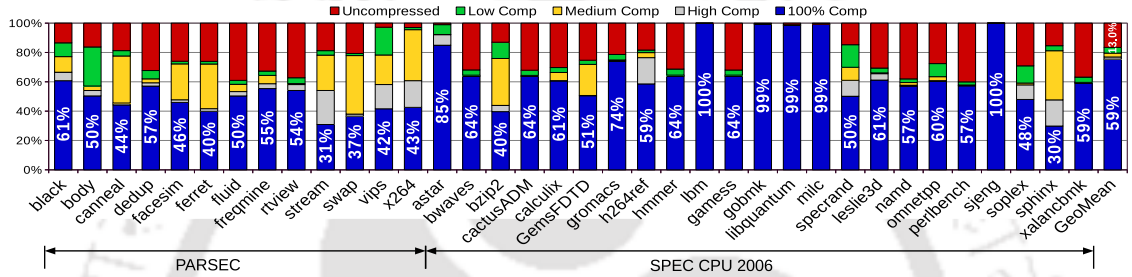

Figure 7.6: Data patterns observed in FlitZip for packet compression. Each flit is divided into 1B chunks and data among the chunks are used to determine the compressibility rates.

incompressible. This is the third motivation behind proposing FlitZip, where data patterns are observed within each flit separately. We call it as *intra-flit* data patterns.

Figure 7.5 shows the percentages of packets that are compressible using different bases in NoΔ for PARSEC and SPEC 2006 benchmarks. We can observe that 10.43%, 21.09% and 6.28% of the packets are compressible with B4, B8 and B16 only 57.22% of the flits are not compressible. Among these packets, only 4.98% of them contain only zeroes, which are compressed by the Zero module. Figure 7.6 shows intra-flit data pattern (marked with different colors) in PARSEC [41] and SPEC CPU2006 [42] benchmarks. 100% Comp (blue) indicates that the data within a flit is the same. High Comp (grey), Medium Comp (yellow) and Low Comp (green) indicate that the intra-flit differences (di) can be represented using $1 \leq di \leq 2$ bits, $3 \leq di \leq 4$ bits, and $5 \leq di \leq 6$ bits, respectively. Uncompressed (red) are those flits whose intra-flit difference varies widely $di \geq 7$. Hence, the lesser bits required to represent di, the higher is the flit compression ratio. From the figure, we can observe that 52.2% flits have the same value, while 3.2%, 4.36% and 5.05% of flits fall in the category of High, Medium, and Low Comp, respectively. And only 21% flits are uncompressible.

Rationale: Figure 7.6 shows the relevance and suitability of compression in flit level granularity as compared to the existing techniques. We can conclude from the figure that

there may arise fewer intra-packet data patterns in existing delta compression techniques, making it incompressible. However, the same packet may have intra-flit data patterns. Thus, FlitZip, exploits intra-flit data patterns for packet compression.

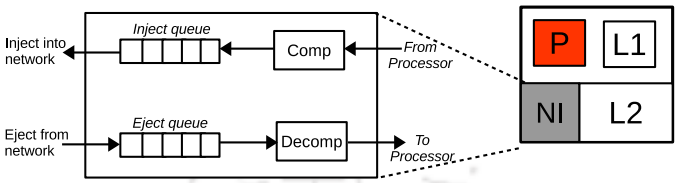

Figure 7.7: Abstract view of compressor and decompressor units in FlitZip.

7.3 Proposed Technique

Figure 7.7 shows an abstract view of a tile when FlitZip is incorporated. Unlike state-of-the-art techniques [17, 21, 22], only a single compressor and decompressor module is added in the NI of each tile. As shown in the figure, FlitZip searches for intra-flit data patterns within a packet when stored in the Inject queue. Upon finding a pattern, the flits of a packet are compressed separately with different bases $(B_1, B_2, ..., B_n)$. Hence, each flit is represented as a set of differences, di from an optimal base, B. If the size of di is less than a byte, the flit is compressible. Thus, the proposed technique performs flit wise compression, as the name suggests *FlitZip*. To indicate whether a flit is compressed or not, a three bit encoding, E, is used. In FliZip, only the di's of each flit is copied to the compressed packet, and for an uncompressed flit, the original flit content is added together to constitute the compressed packet. The base and encoding of each flit form the metadata used by the decompressor to regenerate the original packet. Hence, if the compressed packet consists of fewer flits than the original packet, FlitZip compresses it. Since packets are transferred on a flit-by-flit basis, a compressed packet is flitisized conventionally before injecting into the network. Each flit's metadata is stored in the head flit's unused portion to achieve a highly compressed packet.

Similarly, when a packet is ejected out of the network and stored in the eject queue, the head flit is investigated. From the head flit, encoding bits and base is retrieved for each flit. For compressed flits, the base is added with the differences to regenerate the original flit that are reorganized to form the original packet. Throughout the chapter, FlitZip uses di to represent intra-flit differences, while the existing delta compression techniques use Δ to represent intra-packet differences. Subsection 7.3.1 and Subsection 7.3.2 explains the compression and decompression of FlitZip in details.

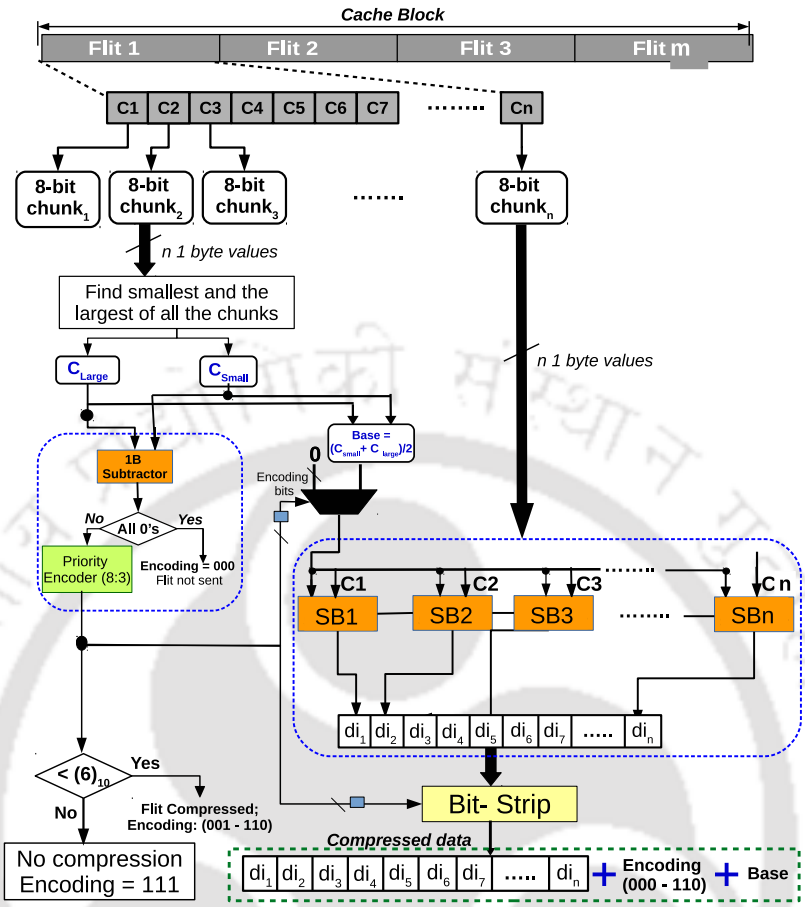

Figure 7.8: Internal structure of compressor; packet consists of m flits each of size n bytes.

7.3.1 Internal Structure of FlitZip Compressor

Figure 7.8 shows the internal structure of FlitZip compressor. Initially, the compressor determines the original flit boundaries, which are multiples of the link bandwidth. Each flit is then divided into 1 byte chunks ($C_1 \ldots C_n$). Since the largest and smallest values bound the chunks, the *Base* is computed as an average of the smallest (C_{small}) and largest (C_{large}) chunk value. The range of data within a flit is determined with a priority encoder that takes C_{small} and C_{large} as input and outputs the encoding bits (001-111). Section 7.3.3 explains the encoding in details. If the largest and smallest chunk is the same, it indicates that the intra-flit data pattern is the same and thus 100% compressible (encoding = 000). For other cases, differences (di) are calculated by subtracting the Base from each chunk. Using equation 7.1, it determined whether a flit is compressible or not. If di can be represented within 6 bits, the flit is compressible; otherwise, incompressible. A total of n subtractors are used for this purpose that takes the chunks and either *Base* or 0 as inputs to calculate

the differences, di ($di = Base - C$). Since di can be either positive or negative, an extra bit is used to indicate the sign as shown in Equation 7.2. When compression is possible (encoding = 001 to 110), one of the inputs to the subtractors (SB1, ..., SBn) is always the Base. However, when compression is not possible (encoding = 111), the multiplexer sends all zeroes instead of the Base in order to keep the uncompressed flit intact. To select either Base or 0, the select line of MUX is determined using the output of the priority encoder. The signal is high when the priority encoder output is within 001 to 110 and for output 111, the signal is low. A small combinational circuit (blue rectangle near the MUX) converts the encoder output into a select line.

$$max_{di} = max\{|di_1|, |di_2|, \ldots, |di_n|\}; n = \frac{flit\ size}{1byte}. \tag{7.1}$$

$$sizeof(max_{di}) = \lceil log_2(max_{di}) \rceil + 1. \tag{7.2}$$

$$\begin{aligned} ComPS = Chunks \times \{(max(|di_1|) + 1) + \ldots + \\ (max(|di_F|) + 1)\}; \\ F = numberof flits; Chunks = n. \end{aligned} \tag{7.3}$$

To represent di in minimum bits, a *Bit-Strip* module is used that strips off the extra bits from each of the di's. Therefore, the priority encoder output is also required to enable or disable the Bit-Strip module. It is enabled when the signal is high (001-110) and disabled when it is low (111). When the module is disabled, no bits are stripped from the chunks, thereby avoiding flit compression. The compressor finally outputs a Base, Encoding bits, and an array of differences, di's for a compressed flit and original data values for an uncompressed flit. Equation 7.3 determines the size of a compressed packet (ComPS).

7.3.2 Internal Structure of FlitZip Decompressor

Figure 7.9 shows the internal structure of FlitZip decompressor that consists of a MUX and n-subtractors. Initially, the decompressor decodes the stored metadata in an incoming head flit. The base and encoding of each flit are extracted to determine the flit boundaries. Using Equation 7.4, the boundary of each flit is obtained in the compressed packet. Flits with encoding 000 are regenerated by copying the base by the number of chunks. For other cases, the module takes di_1, ..., di_n from the incoming flit and the MUX sends either base (for encoding 001 to 110) or 0's (for encoding 111, the flits are uncompressed) as inputs. Hence, the select line of the MUX is determined from the encoding bits. Each di is obtained from

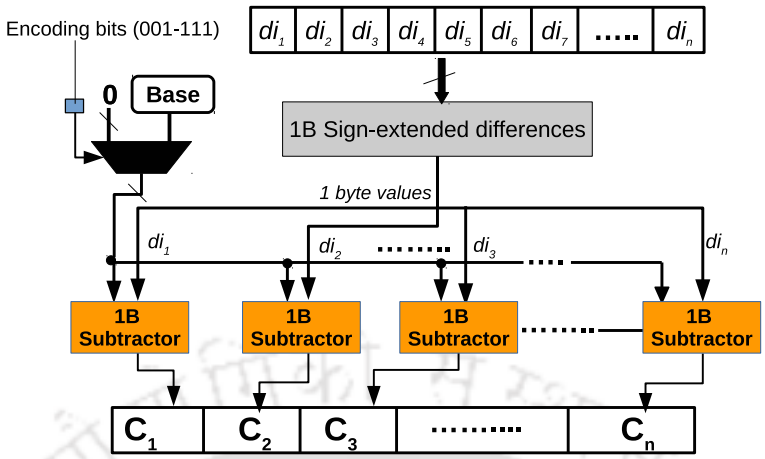

Figure 7.9: Internal structure of decompressor where n is flit size in bytes.

the encoding bits and is sign-extended to 1 byte values. Example: For a flit size of 4 bytes ($Chunks = 4$), if flit 1 has an encoding of 011, each di is of 3 bits. Thus, flit 1 consists of first 12 bits ($3 \times Chunks \times 1$) where bits $[0, 1, 2]$ corresponds to di_1, $[3, 4, 5]$ corresponds to di_2 and so on in the compressed packet. The subtractors take the sign-extended di's and the base as inputs and finally output the original flit, reorganized to generate the original packet. Since in FlitZip, a packet received after decompression is the same as that of the original packet, it is classified as lossless compression.

$$\begin{aligned} flit\ boundary_i &= Encoding\ bits \times Chunks \times i; \\ i = flit\ number, 1 \leq i &\leq \frac{packet\ size}{link\ bandwidth} \end{aligned} \tag{7.4}$$

Table 7.1: Encoding table in FlitZip.

Number of bits for di	Encoding bits	Compressed-flit size (in bits)
≥7	**111**	-No-
6	110	(6+1)*16 = 112
5	101	(5+1)*16 = 96
4	100	(4+1)*16 = 80
3	011	(3+1)*16 = 64
2	010	(2+1)*16 = 48
1	001	(1+1)*16 = 32
0	000	0 (All Same; including zeroes)

7.3.3 Encoding Table

Table 7.1 shows the encoding table of FlitZip that is stored at the NI of each tile. The third column is not a part of the table, and the compressed flit size is calculated using the

original flit size and link bandwidth as 16 bytes each. The table consists of eight entries where each entry is a 2-tuple: <*di-bits, encoding bits*>. Encoding bits store the size of *di* of each compressed flit used by the decompressor to determine the flit boundaries in a compressed packet. Three encoding bits can sufficiently represent a total of 2^3 encodings, out of which 111 and 000 have a special meaning. 111 indicates that the flit is uncompressed, and 000 indicates that the flit chunks are the same. Rest combination indicates various encodings used for different *di*-bits. For example: if *di* requires 5 bits for a compressed flit, the encoding is 101.

7.3.4 Metadata Storage in Head Flit

FlitZip stores the metadata of a compressed packet in the corresponding head flit's unused location. As shown in Figure 7.2, for 128 bit link bandwidth, 75 bits from the head flit is unused. For a 64B packet (4 body flits), each flit requires a metadata of 11 bits (Base = 8 bits, Encoding = 3 bits). Hence, a total of $44(11 * 4)$ bits from the head flit with bit location [74:31] is used to store all the flits' metadata. Bits [74: 64] contain the encoding bits and base of flit 1 out of which bits [74:72] are the encoding bits, and the next 8 bit is the base. Similarly, bits ranging from [63:53], [52:42] and [41:31] contains the metadata for flit 2, 3 and 4, respectively. Thus, storing the metadata in the head flit results in achieving a higher flit compression ratio than NoΔ.

7.3.5 Illustrative Example of FlitZip De/compression:

Figure 7.10 and Figure 7.11 shows the working methodology of FlitZip. In the figure, the original packet and flit size is 16 bytes and 4 bytes, respectively. The left half of the figure shows the same flit as in Figure 7.4 that NoΔ could not compress. Flit 1 has four chunks: C1 = 80, C2 = 81, C3 = 82 and C4 = 83 out of which C_{small} is 0x80 and C_{large} is 0x83. Therefore, base B1 is 0x81 (10000001) and *di* requires 3 bits (2+1 sign bit). Thus, flit 1 is compressible and the encoding is 011. Flit 2 is not compressible as *di* requires 8 bits. Therefore, the encoding for flit 2 is 111. For flits 3 and 4, each flit has same data values. These flits are not transmitted and the (encoding, base) used for flit 3, 4 is (000, 11111111) and (000, 00000000), respectively. Figure 7.10(B) shows the content of the compressed packet using FlitZip where di_j^i is the *jth* difference of flit *i*. The encoding and base of each compressed flit is stored in the head flit as shown in Figure 7.10(C). Thus, FlitZip compresses the packet in figure 7.4 by 50% which NoΔ is unable to compress.

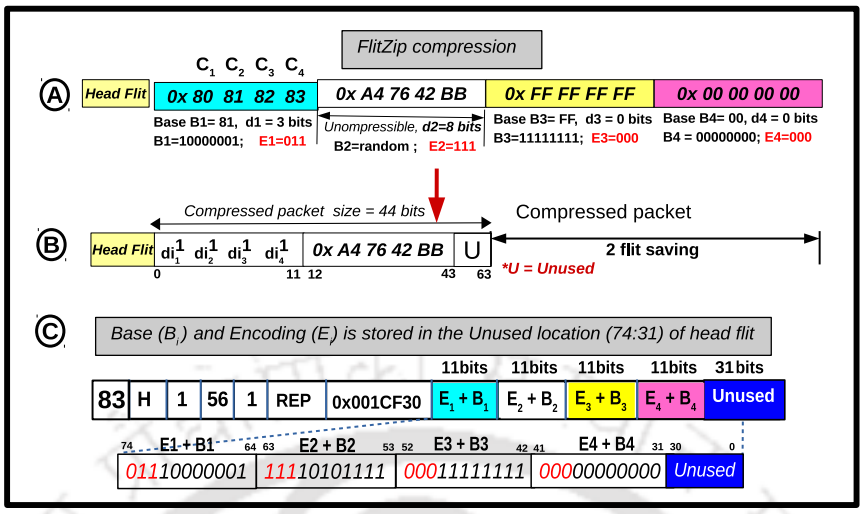

Figure 7.10: Example of FlitZip compression.

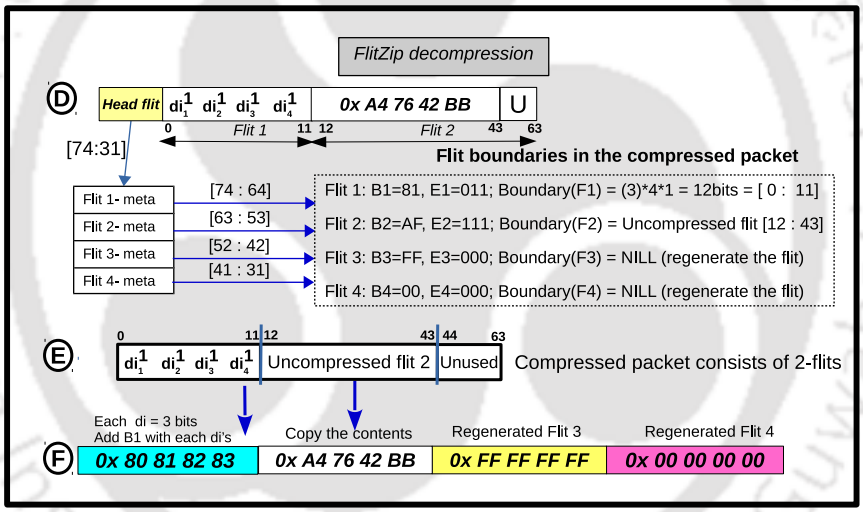

Figure 7.11: Example of FlitZip decompression.

Figure 7.11 shows FlitZip decompression. As shown in the figure 7.11(D), from the head flit, the metadata of the compressed packet is extracted [74:31]. Bits [74, 73, 72] is 011 which means that the number of bits used for *di* in flit 1 is 3. Thus, flit 1 is of 12 bits $(3 \times Chunks \times 1)$. As shown in Figure 7.11(E), in the compressed packet the flit ranges from [0 : 11]. The base extracted from the head flit is added with di_i^1 to generate the original flit. For flit 2, bits [65:63] from the head flit is 111. Hence the flit boundary is [12 : 43] (original flit size = 32 bits). On the other hand, for flit 3 and 4, the encoding bits are 000. Hence, both the flits are regenerated using the base that is copied by the number of chunks. The decompressed flit is shown in Figure 7.11(F) which is same as that of Figure 7.10(A). Thus,

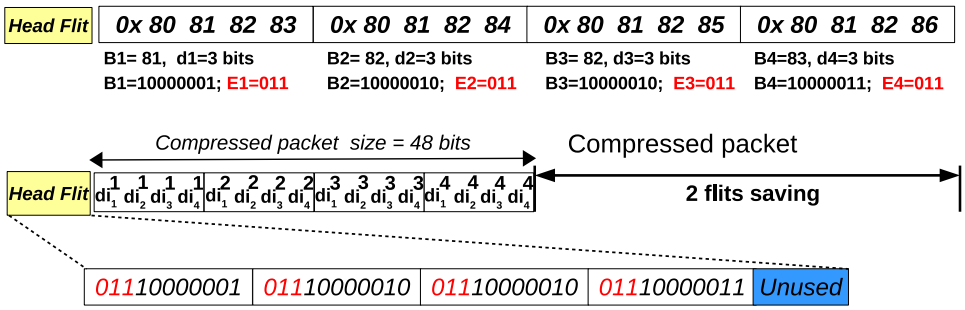

Figure 7.12: Illustration of FlitZip mechanism and savings achieved.

FlitZip is a lossless compression that can effectively reconstruct the original packet from a compressed packet.

7.3.6 Advantage of FlitZip:

Unlike the existing delta compression techniques [17, 21, 22] that uses multiple bases (4B, 8B and 16B), FlitZip uses a constant base of size 1B. Hence, it avoids the requirement of multiple de/compression units with varying bases. This addresses the second problem of simultaneous computation, as mentioned in Section 7.2. Also, FlitZip can compress all packets that NoΔ and DISCO can, and in addition to that, FlitZip compresses some extra packets that they cannot. Hence, it can compress more packets than the existing techniques, as shown in Figure 7.12. The figure is the same as that of Figure 7.3 where existing delta compression techniques compresses the packet using B4Δ1 with the same flit compression ratio of 50%. Experimentally we found that FlitZip can compress 27% of additional network packets compared to existing techniques across all the benchmarks.

7.4 Experimental Analysis

FlitZip and state-of-the-art techniques (NoΔ [17], ZeroCompr [32], FPC [34] and DISCO [21]) are modelled in gem5 [40] and the NoC component is modelled in garnet 2.0 [134] which is closely integrated with gem5. The details regarding the simulators are already discussed in chapter 3. The de/compressor units have been tested for power, area, and timing estimations by Verilog implementation and synthesizing in Synopsys Design compiler.

Table 7.2: Simulation parameters for FlitZip.

Processor	4/16/64, x86 OoO superscalar
L1 cache per tile	32KB, 4-way associative, 64B block
L2 cache per tile	1MB, 8-way associative, 64B block
L1 and L2 cache access time	2, 10 cycles

7.4.1 Experimental Setup and Workload Description

The simulation parameters used to model the techniques are mentioned in Table 7.2 and Table 3.3. We used various network sizes (2x2, 4x4 and 8x8) and experimented using 13 benchmarks from PARSEC [41] and 384 workload mixes for 64 cores from SPEC 2006 benchmarks [42] to analyze the performance of FlitZip. The baseline used is a system with no packet compression. FlitZip is further compared with existing techniques NoΔ, ZeroCompr, FPC and DISCO. For performance metrics, we have used flit count, weighted speedup, flit compression ratio, packet latency, bandwidth utilization and packet queuing latency.

Table 7.3: Workload mixes created from SPEC benchmark where xB_i - yB_j means x% of B_i and y% of B_j are used to create the mix, 1 $\leq$ i,j $\leq$3.

M1, M2, M3	B1 (all), B2 (all), B3 (all)
M4 , M5	0.75B1 - 0.25B2, 0.75B1 - 0.25B3
M6, M7	0.75B2 - 0.25B3, 0.75B3 - 0.25B2
M8, M9	0.75B3 - 0.25B1, 0.75B2 - 0.25 B1
M10, M11	0.5B1 - 0.5B2, 0.5B1 - 0.5B3
M12	0.5B2 - 0.5B3
M13,	0.25B1 - 0.5B2 - 0.25B3
M14,	0.25B1 - 0.5B3 - 0.25B2
M15	0.25B2 - 0.5B1 - 0.25B3
M16	Random combination

Workload Characterization: We have used real workloads from SPEC CPU 2006 and PARSEC benchmark suite to evaluate the performance of FlitZip. SPEC benchmarks are classified into three categories ($B1$, $B2$ and $B3$) based on their MPKI from L1 cache. B1 (MPKI $<$ 0.25) contains *gromacs, sjeng, bwaves, hmmer, calculix, gobmk, cactusADM, namd, sphinx* and *h264ref*. B2 (0.25 $\leq$ MPKI $<$ 0.5) consists of *astar, specrand, mcf, milc, omnetpp* and *gamess*, and B3 (MPKI $\geq$ 0.5) consists of benchmarks *bzip2, lbm, leslie3d, soplex, GemsFDTD, perlbench, xalancbmk* and *libquantum*. Using these benchmarks, we

create workloads by combining them in different ratios and shown in Table 7.3. The table shows 16 different workload mixes ($M1 \ldots M16$) generated by mixing B1, B2 and B3 in different ratios. The benchmarks in each workload mixes are mapped to 64 cores in 24 different ways as shown in Figure 3.3. Hence, a total of 384 different workloads for 64-core and 128 for 16-core are created to evaluate the effectiveness of FlitZip. The workloads are fast forwarded for 10 billion instructions, warm up for next 1 billion instructions and then executes for next 2 billion instructions for collecting the statistics. The results plotted for each Mi is the geometric mean of 24 different ways of workload mapping. For PARSEC benchmarks we have used *black, body, canneal, dedup, facesim, ferret, fluid, freqmine, rtview, stream, swap, vips* and x*264*.

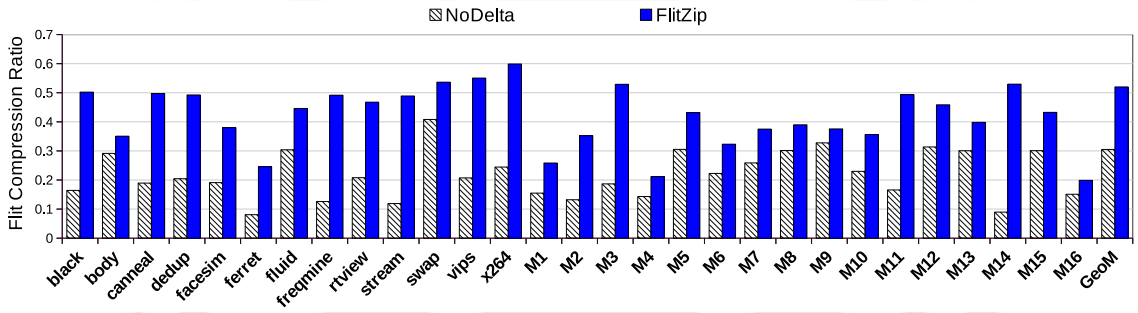

Figure 7.13: Comparison of flit compression ratio in PARSEC and SPEC workloads.

7.4.2 Effect on Flit Compression Ratio

Figure 7.13 shows the FCR achieved by FlitZip and NoΔ. It is calculated using Equation 3.2 (mentioned in Chapter 3). On average (geometric mean), FlitZip achieves an FCR of 0.52, whereas FCR for NoΔ is 0.30 only. Kindly note that the packet count in both the technique remains the same. It only reduces the number of flits within a packet, thereby reducing the packet size. Figure 7.14 shows the normalized flit count after applying FlitZip compression. From the figure, we can observe that FlitZip effectively reduces flit count in the network by 42.53% whereas NoΔ reduces the flit count by 16% as compared to the baseline. FlitZip achieves higher FCR than NoΔ as shown in the figure with the highest/lowest compression ratio of 0.59 / 0.44 in x*264* and *fluid* for PARSEC benchmark and 0.52 / 0.19 in *M3,M14* and *M16* for SPEC CPU 2006 workload mixes.

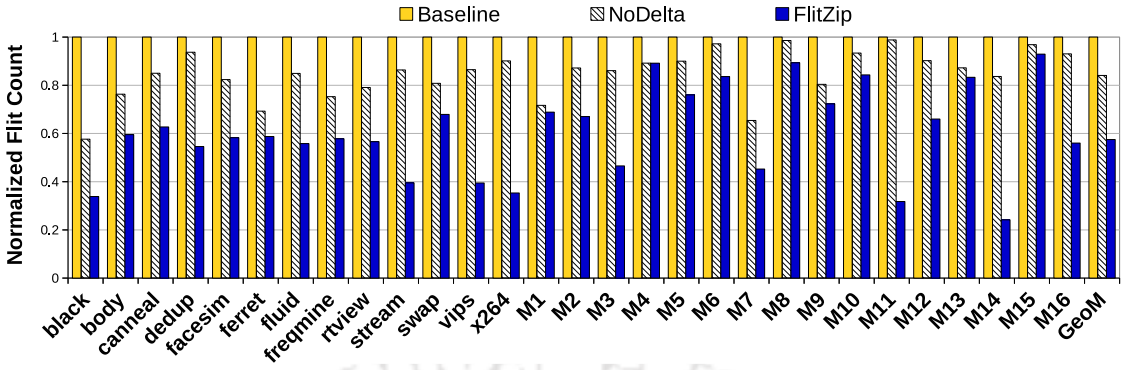

Figure 7.14: Comparison of flit count normalized to Baseline.

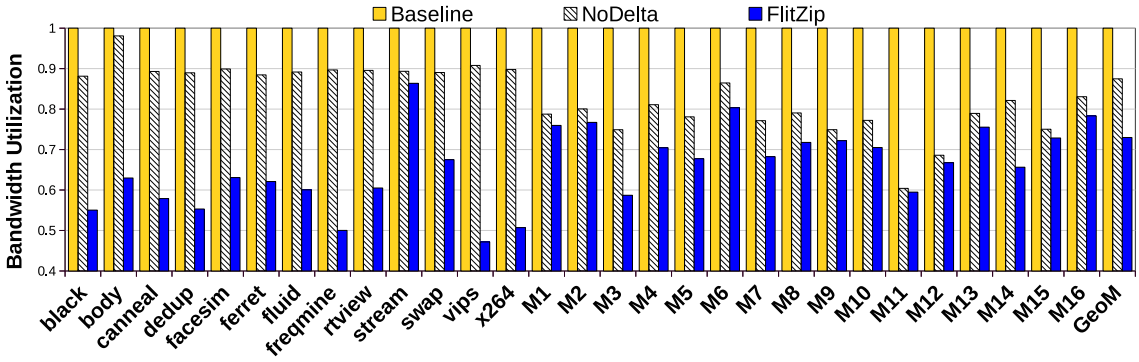

Figure 7.15: Comparison of bandwidth utilization normalized to Baseline.

7.4.3 Effect on Bandwidth Utilization

In NoC, on-chip bandwidth is an essential component used for carrying flits from one router to the other. An increase in the flit compression ratio reduces the number of flits transferred between the tiles. This reduces the usage of on-chip bandwidth, thereby resulting in fewer network activity and transistor switching. Thus, an efficient packet compression technique also reduces the NoC power consumption. Figure 7.15 shows a comparison of normalized bandwidth utilization in NoΔ and FlitZip w.r.t. baseline. As compared to the baseline, on average, FlitZip and NoΔ reduce the bandwidth utilization by 27% and 12.5%, respectively, to transfer flits across the tiles. Among the PARSEC and SPEC 2006 benchmarks, x*264* (50%) and M3 or M14 (41%) workload mixes have the lowest bandwidth utilization rate, respectively. Workload mixes M3 and M14 consists of B3 benchmark mixes (c.f. Table 7.3) which is a combination of *lbm-libquantum-soplex-GemsFDTD-bzip2-xalancbmk* benchmarks. Since these combinations achieve a higher flit compression ratio than NoΔ, it helps in compressing packets efficiently, thereby generating less flit count.

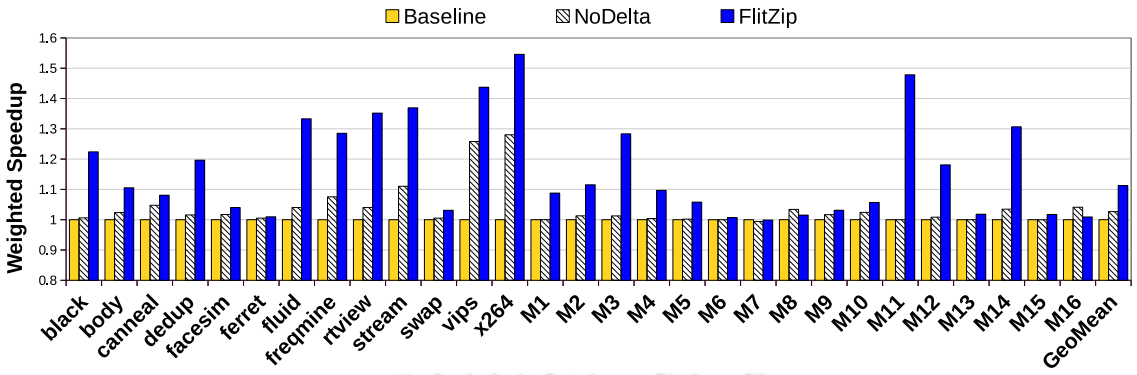

Figure 7.16: Comparison of weighted speedup normalized to Baseline.

7.4.4 Effect on Weighted Speedup

As shown in Figure 7.16, FlitZip provides a better trade-off between performance (speedup) and bandwidth utilization as it provides 12% performance improvement with a 39.87% reduction in flit count when compared to baseline. Also, when compared with NoΔ, the weighted speedup of FlitZip increases by 8.35%. The primary goal of FlitZip is to reduce redundancies within packets such that the request and reply packets are transmitted faster to the requesting tile and reduces NoC power consumption. Since FlitZip achieves a higher flit compression ratio than NoΔ, a packet takes less time to reach the core. It results in faster instruction execution, thereby increasing the system throughput. Among the PARSEC and SPEC 2006, x*264*, and the mix *M11, M3, M14* are the major performance gainers as compared to the baseline and NoΔ.

7.4.5 Average Packet Latency

Figure 7.17 shows the normalized reduction in packet latency (PL) for various benchmarks. In PARSEC and SPEC CPU 2006 mixes, the packet latency reduces by 13.57% and 14.4%, respectively. Since FlitZip reduces network traffic, the time taken by a packet to reach the requesting tile decreases. Hence, FlitZip reduces the packet latency by 19.28% as compared to the baseline.

7.4.6 Average Packet Queuing Latency

Since FlitZip efficiently reduces the flit count within a packet, the queuing time of the packet reduces. Figure 7.18 shows that FlitZip reduces the queuing latency by an average of 13.3% and 7.15% (geometric mean) compared to baseline and NoΔ, respectively. Since

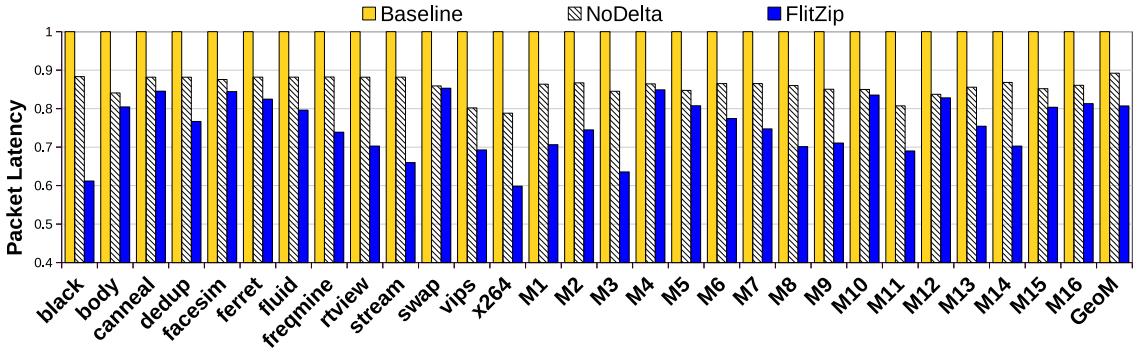

Figure 7.17: Comparison of packet latency (in cycles) normalized to Baseline.

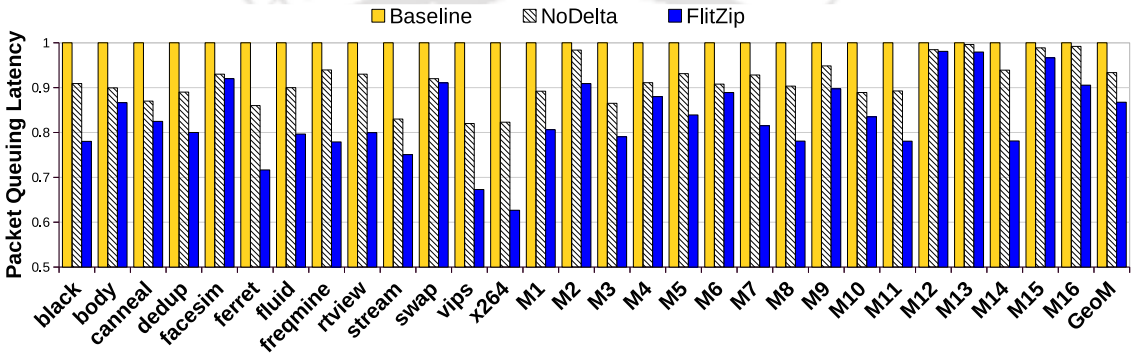

Figure 7.18: Comparison of packet queuing latency (in cycles) normalized to Baseline.

network stall time is defined as the amount of time required by a packet to travel from the requesting tile to the destination tile, it involves both packet latency and packet queuing latency. FlitZip efficiently reduces both the parameters, thereby reducing the network stall time. Reduction in network stall time directly reduces $AMAT_{TCMP}$ by 31.29%.

7.4.7 Sensitivity Analysis

We have also analyzed the sensitivity of FlitZip to various parameters such as network size, base size, block size and NoC bandwidth.

A. Change in Network Size:

Table 7.4 shows the performance of FlitZip as compared to NoΔ and baseline with varying network size between 4 cores (2x2 NoC) to 64 cores (8x8 NoC). For comparison we have used the parameters weighted speedup, packet latency and bandwidth utilization because an efficient compression technique achieves higher reduction in network flits that further lowers

Table 7.4: Comparison of various metrics with varying network size normalized to baseline and NoΔ (values in percentages). WS: Weighted Speedup, PL: Packet Latency, BU: Bandwidth Utilization.

	WS	PL	BU
2x2 NoC (4 cores : 64 SPEC workloads + 13 PARSEC benchmarks)			
NoΔ	5.88 (20.45 / 0.66)	8.65 (18.2 / 9.62)	12.9 (25 / 1.5)
Baseline	9.86 (45.54 / 0.01)	13.21 (27.4 / 10)	21 (39.1 / 11.2)
4x4 NoC (16 cores:128 SPEC workloads + 13 PARSEC benchmarks)			
NoΔ	7.08 (25 / 1.01)	10 (20 / 12)	15.2 (34.5 / 1.18)
Baseline	11.05 (49.76 / 0.086)	15.76 (31.3 / 10.2)	22.5 (45.8 / 11.27)
8x8 NoC (64 cores : 384 SPEC workloads + 13 PARSEC benchmarks)			
NoΔ	8.35 (28.03 / 1.12)	9.46 (21.82 / 11.78)	16.56 (48.02 / 1.52)
Baseline	12 (54.56 / 0.1)	19.28 (38.8 / 14.4)	27 (50 / 13.6)

the average packet latency and bandwidth utilization in the network. Thus, packets can travel faster in the network, thereby increasing weighted speedup for the application. Each entry is in the form of A (B / C) where A is the geometric mean over all the workloads, and B and C is the highest and lowest respective values in percentages. The table can be interpreted as follows. For network size of 2x2, the average performance of FlitZip across all the workload is 5.88% compared to NoΔ with the highest value as 20.45% and the lowest as 0.66% in terms of WS. Similarly, for the metric WS, the average performance of FlitZip is 9.86% compared to Baseline with the highest value as 45.54% and the lowest as 0.01%. Accordingly, the remaining entries can also be interpreted. From the table, we can observe that on average FlitZip performs better across different NoC sizes as compared to NoΔ and baseline. However, we can also observe that the performance of 8x8 NoC is better than 4x4, which in turn is better than 2x2. This is because the bigger the network diameter, the larger is the time taken by a flit to reach its destination (as the end to end hop count increases). Also, the number of applications running in 8x8 NoC is 4 times more than that of a 4x4 NoC, which generates more traffic to the network. Thus, FlitZip effectively reduces the flit count in the network, thereby reducing the overall traffic.

B. Change in Block size, Link bandwidth and Memory size:

Compression in FlitZip is restricted by the number of unused bits in the head flit. FlitZip can compress a packet if it contains a maximum of $\left\lfloor \frac{(Unused\_Bits)}{(11\ bits(E+B))} \right\rfloor$ number of flits. Since the number of unused bits for a standard 128 bit link width is 75, FlitZip can compress a packet if it contains a maximum of 6 body flits. Table 7.5 shows FlitZip compression for different packet size when the NoC link bandwidth is 128 bits and 256 bits, respectively. From the

Table 7.5: Sensitivity analysis with varying block size and NoC link bandwidth.

Link bandwidth	Block/packet size	#Flits	Metadata bits
128 bits (16B), 75 unused bits	16 B	1	11
	32 B	2	22
	48 B	3	33
	64 B	4	44
	80 B	5	55
	96 B	6	66
256 bits (32B), 203 unused bits	32 B	1	11
	64 B	2	22
	96 B	3	33
	128 B	4	44
	256 B	8	88
	512 B	16	176

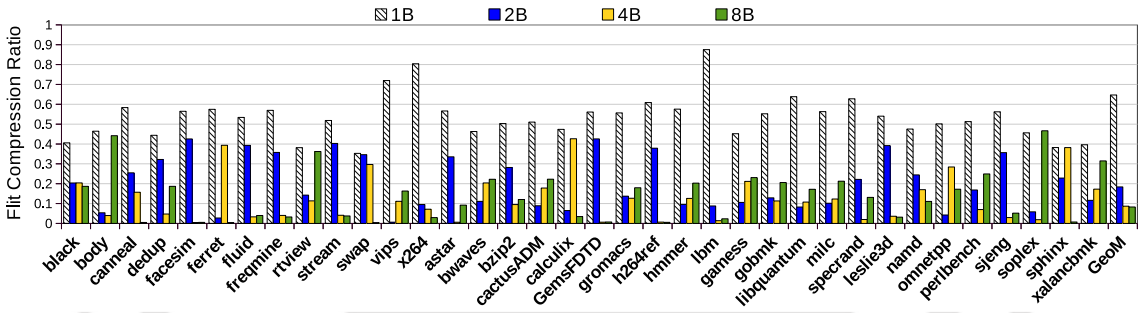

Figure 7.19: Sensitivity analysis with varying base sizes for 8x8 NoC. [Packet size: 64 bytes and flit size: 16 bytes.]

table, we can observe that for 128 bit link bandwidth and 75 unused bits in the head flit, a packet size (cache block) of 32 bytes is divided into two flits. Since each flit requires 11 bits to store the metadata information, the total bits utilized from the unused location of head flit is 22 (11×2). Similarly, we can also observe that for 256 link bandwidth, FlitZip can compress packets of size ranging from 16 bytes to 512 bytes. The standard cache block size being $64B$, FlitZip can sufficiently compress a packet/block.

C. Change in Base Size:

Figure 7.19 shows the flit compression ratio achieved by FlitZip when the base size varies between 1 byte to 8 bytes. From the figure, we can observe that FlitZip can compress maximum flit across all the benchmarks when the base size is 1 byte. Since integer/floating-point data are in multiples of bytes, considering the lower byte granularity helps FlitZip find maximum patterns within a flit. On the other hand, NoΔ uses varying bases whose size is in multiple of bytes. The benefit of using constant base size is to avoid multiple de/compressor modules. The figure also indicates that the probability of finding patterns

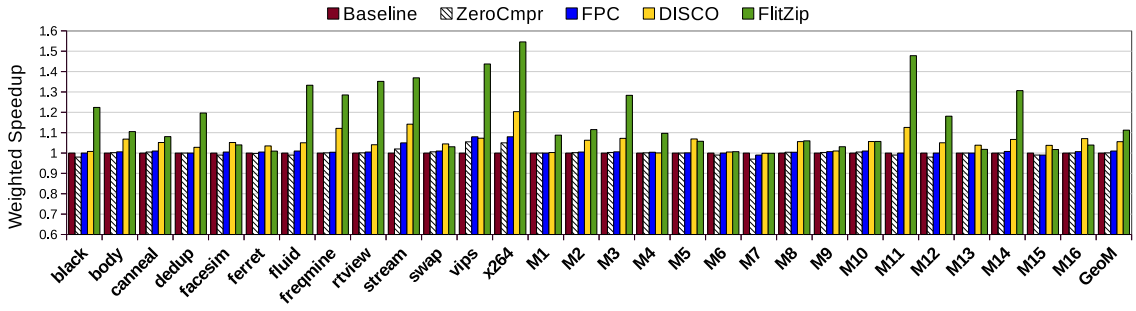

Figure 7.20: Performance of FlitZip compared to ZeroCompr, FPC and DISCO normalized to Baseline.

on a larger base size ($> 1byte$) is less than that of 1 byte. Thus, FlitZip uses a base of size one byte for flit compression.

D. Comparison of FlitZip with Other Techniques

We have also implemented ZeroCompr [32], FPC [34] and DISCO [21] in 8x8 NoC for evaluating the performance of FlitZip. Figure 7.20 shows the average performance of FlitZip with respect to the existing techniques. On average, FlitZip performs better than all the techniques. This is well explained because only 4.98% of all packets contains only zeroes averaged across all the benchmarks. Hence, ZeroCompr compresses only a small number of packets inspite of having different patterns within a packet (Figure 7.6). On the other hand, in FPC, the encoding table has a restricted count of eight patterns only. Also, packet de/compression in FPC requires indexing into the encoding and decoding table, which is time consuming (5 cycles). This degrades the performance of FPC as compared to FlitZip. DISCO router with NoΔ/BDI compression for NoC packets, on the other hand, performs better than ZeroCompr and FPC, but the performance degrades as compared to FlitZip. This is because DISCO router selects those packets for compression that experience a larger waiting time in the router's internal buffers. So it avoids compressing those packets with the least waiting time even if the packet content has a well-defined pattern. Though the selection criteria are advantageous in avoiding the extra de/compression latency for the compressed packets, the count of compressed packets is limited compared to FlitZip.

7.4.8 Hardware Analysis

We have also implemented FlitZip in hardware using $45nm$ technology.

A. Timing aspects of FlitZip De/compressors:

In FlitZip, a compressor circuit requires two cycles and a decompressor circuit requires one cycle. Since the head flit is the first to reach the requesting tile, the body flits can be decompressed as soon as they arrive at the network interface of the requesting tile. In zero-network load, consecutive flits arrive at a router after every cycle, and the router requires two additional cycles to process it, thereby requiring a total of three cycles per body flit (1 cycle-link traversal and 2 cycle-processing at the local router). Hence, the decompression latency for all body flits is hidden by the flits' queuing latency except for the last flit. The last flit requires one cycle to decompress, thereby reducing the decompression latency to a single clock cycle with a single decompression circuit. The number of compressors that FlitZip can use is a design choice that depends on whether it is used for a low-power or a low-latency system. A single compressor can be used for a low power system where the flits are compressed serially one after another. In such a scenario, compression may require more cycles. On the other hand, for a system with low latency, multiple compressors can be used that can compress all the flits in parallel [18]. In our experiment, we have used four compressors to compress a four flit packet, in parallel.

B. Power and Area aspects of FlitZip De/compressors:

FlitZip does not change the router microarchitecture and it is similar to that of Figure 2.4. The de/compressor units are added to the NI. Since FlitZip uses a single static base size of 1 byte, it avoids multiple compression modules, unlike NoΔ. The chunks, C_i and base being of equal size (1 byte), FlitZip avoids extra bit padding for computing the differences, di_i. In addition to these, the size of the base and difference (di) is $1B$ each. Hence, the bit-width of the subtractor units reduces from $16B$ to $1B$ in FlitZip. The power required for sign-extending, i.e., fan-in AND and OR gates, is also avoided in the FlitZip compressor as the chunks and base are of equal size, unlike NoΔ. It greatly reduces the size and dynamic power consumption of the de/compressor unit, thereby making compression and decompression energy efficient. Also, the encoding table at each tile requires 48 bits (6 bits*8 = 6B) which is 97.6% less storage than the encoding table used in NoΔ per tile.

The de/compressor unit in FlitZip, being an additional component added in the NI, requires an extra power of $13.75mW$ while NoΔ has $36.48mW$ of power consumption. Hence, FlitZip reduces power by 62.3% as compared to NoΔ. Also, the combined area of the de/compressor module in FlitZip and NoΔ is $0.0028mm^2$ and $0.006mm^2$, respectively. Thus

the area of the de/compressor module in FlitZip is also reduced by 53.33% than that of NoΔ. The reduction in area and power consumption in FlitZip is obtained because NoΔ uses multiple de/compression modules for various (Base, Δ) sizes executed in parallel on a packet, and each module consumes additional power, which is completely avoided in FlitZip. Moreover, efficient reduction in the flit count resulted in carrying fewer flits in the NoC. Hence, FlitZip reduces the dynamic power consumption in the network by 16.68% as compared to NoΔ. The de/compression circuit in FlitZip being very simple and power efficient, FlitZip can become a better packet compression technique for NoC.

7.5 Chapter Summary

In this chapter, we observed the shortcomings in existing delta compression techniques and found that the existing techniques cannot compress a packet even though the flits have repetitive data patterns. Towards this, we proposed FlitZip that searches for data patterns within each flit and compresses them separately. This helped in achieving a higher flit compression ratio than the existing techniques. We have also proved that FlitZip is efficient in terms of area and power consumption by proposing a lightweight compressor and decompressor units.

CHAPTER

8

Conclusion and Future Directions

In this thesis, we have shown that the performance of TCMP can be improved by reducing the memory wall problem using two orthogonal techniques: prefetching and on-chip packet compression. However, the existing prefetching techniques are not suitable for TCMP architectures, as discussed in Chapter 1. Thus, we have revisited prefetching techniques for TCMPs in this thesis. Also, the existing on-chip packet compression techniques cannot compress packets efficiently. Hence, we also propose a novel packet compression technique. The summary of the thesis is mentioned in the next section. Section 8.2 provides insights into the possible future works in a similar direction.

8.1 Summary of Thesis

Figure 8.1 shows the summary of the contributions made towards the thesis. As shown in the figure, the thesis aims to improve system performance in TCMP architectures by reducing $AMAT_{TCMP}$ required to fetch cache blocks from a different tile. The primary focus is on proposing efficient prefetching techniques for TCMP architectures by reducing cache pollution. Cache pollution generates additional NoC traffic that congests the network and further delays the on-chip packet transmission. Thus, cache pollution directly contributes to NST. It can be reduced either by using an intelligent prefetch block placement strategy or throttling the useless prefetch requests. Both these techniques can also be combined to make prefetching efficient. In the former category, we have proposed two different contributions: ECAP and ZPP. In the latter category, we proposed proposed COPE to reduce prefetcher-caused cache pollution. Another technique that can reduce the network stall time is on-chip packet compression. Packet compression reduces packet size, which

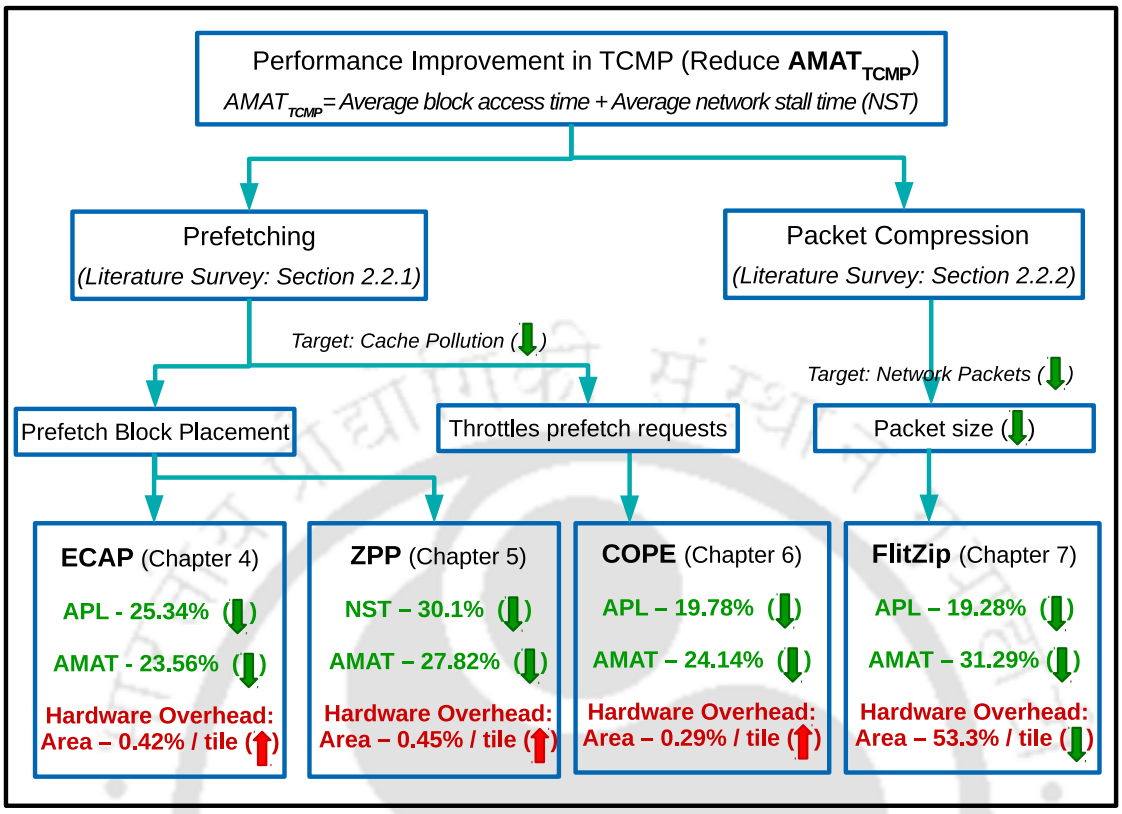

Figure 8.1: Summary of the thesis. In the figure, performance of each contribution is compared to the baseline. Detailed description of each contribution is mentioned in the respective chapters. [Average Packet Latency (APL), Network Stall Time (NST), AMAT. ⇓: parameter decreases, ⇑: parameter increases]

results in a faster packet transmission rate in the network. Thus, it reduces NST, thereby reducing AMAT. Though there exist on-chip packet compression techniques, the existing ones cannot reduce network packets efficiently at flit level. Hence, in this thesis, we have proposed a novel packet compression technique, FlitZip. The experimental framework used to model each contribution is described in Chapter 3.

In Figure 8.1, the performance of each contribution and the additional hardware requirement are shown in different colors. ECAP reduces the APL and AMAT by 25.34%, 23.56%, respectively, with an area overhead of 0.42% per tile. ZPP, on the other hand, reduces NST and AMAT by 30.1%, 27.82%, respectively, with a hardware overhead of 0.45% per tile. The performance of ZPP is better described by NST rather than APL. COPE reduces APL and AMAT by 19.78%, 24.14%, respectively, by incurring an area overhead of 0.29%. FlitZip reduces APL and AMAT by 19.28%, 31.29%, respectively, and reduces the area overhead by 53.3% per tile compared to state-of-the-art technique NoΔ [17].

8.2 Future Work

The contributions in the thesis can be extended in many ways. Some of the possible works that can be explored as future directions are summarized as follows.

- In chapter 4, ECAP reduces prefetcher-caused cache pollution by placing the prefetch blocks in neighboring tiles running light applications. However, placing the prefetch block and requesting them from the neighboring tiles generate additional NoC packets, increasing network traffic for the region. As a part of future work, the additional packets generated by ECAP can be reduced by exploring a piggybacking techniques.
- In chapter 5, ZPP removes prefetch-caused cache pollution by placing prefetch blocks in the stale block's locations of the local L2 bank. ZPP can be further experimented by placing the prefetch blocks in the neighboring tiles' L2 bank.
- In chapter 6, COPE redefines prefetch accuracy in the context of TCMP architectures to throttle prefetcher-caused NoC traffic. COPE can be further explored by grouping the tiles such that while throttling useless prefetch requests, it uses prefetch accuracy for a group of tiles rather than individual tiles.
- In Chapter 7, FlitZip compresses on-chip packets at flit-level granularity. The packets are generated as a result of an L1 cache miss. It can be further experimented by compressing the L2 cache miss and reply packets that travel to the main memory through NoC.
- In the thesis, we propose TCMP-aware prefetch block placement strategies, prefetch throttling mechanism, and on-chip packet compression technique. Hence, there is a need to propose a TCMP-aware prefetcher that incorporates all these techniques to make prefetching efficient for TCMP architecture.

Bibliography

[1] G. E. Moore, "Cramming More Components onto Integrated Circuits," *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 33–35, 2006.

[2] S. Kaxiras and M. Martonosi, "Computer Architecture Techniques for Power-Efficiency," *Synthesis Lectures on Computer Architecture*, vol. 3, no. 1, pp. 1–207, 2008.

[3] K. Olukotun, *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*, 1st ed. Morgan and Claypool Publishers, 2007.

[4] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F.-L. Qiao, W. Zhao, X. Yin, C. Hou, C. Zhang, W. Ge, J. Zhang, Y. Wang, C. Zhou, and G. Yang, "The Sunway TaihuLight Supercomputer: System and Applications," *Science China. Information Sciences*, vol. 59, pp. 1–16, 07 2016.

[5] M. B. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal, "Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures," in *9th International Symposium on High-Performance Computer Architecture*, 2003, pp. 341–353.

[6] C. Kim, D. Burger, and S. W. Keckler, "An Adaptive, Non-uniform Cache Structure for Wire-delay Dominated On-chip Caches," *SIGARCH Computer Architecture News*, vol. 30, no. 5, pp. 211–222, 2002.

[7] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler, "A NUCA Substrate for Flexible CMP Cache Sharing," in *19th International Conference on Supercomputing*, 2005, pp. 31–40.

[8] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook, "TILE64 - Processor: A 64-Core SoC with Mesh Interconnect," in *IEEE International Solid-State Circuits Conference - Digest of Technical Papers*, 2008, pp. 88–598.

[9] A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. Liu, "Knights Landing: Second-Generation Intel Xeon Phi Product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, 2016.

[10] T. Bjerregaard and S. Mahadevan, "A Survey of Research and Practices of Network-on-chip," *ACM Computer Survey*, vol. 38, no. 1, p. 1âĂŞes, Jun 2006.

[11] L. Benini and G. De Micheli, "Networks on Chips: a New SoC Paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, 2002.

[12] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *ACM SIGARCH Computer Architecture News*, vol. 23, no. 1, pp. 20–24, 1995.

[13] N. P. Jouppi, "Improving Direct-mapped Cache Performance by the Addition of a Small Fully-associative Cache and Prefetch Buffers," in *17th International Symposium on Computer Architecture*, 1990, pp. 364–373.

[14] B. Falsafi and T. F. Wenisch, *A Primer on Hardware Prefetching.* Morgan & Claypool Publishers, 2014.

[15] S. Mittal, "A Survey of Recent Prefetching Techniques for Processor Caches," *ACM Computing Surveys*, vol. 49, no. 2, pp. 35:1–35:35, 2016.

[16] Y. Zhang, Y. Yuan, D. Feng, C. Wang, X. Wu, L. Yan, D. Pan, and S. Wang, "Improving Restore Performance for In-Line Backup System Combining Deduplication and Delta Compression," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 10, pp. 2302–2314, 2020.

[17] J. Zhan, M. Poremba, Y. Xu, and Y. Xie, "NoΔ: Leveraging Delta Compression for End-to-End Memory Access in NoC based Multicores," in *19th Asia and South Pacific Design Automation Conference*, 2014, pp. 586–591.

[18] G. Pekhimenko, V. Seshadri, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "Base-Delta-Immediate Compression: Practical Data Compression for On-chip Caches," in *21st International Conference on Parallel Architectures and Compilation Techniques*, 2012, pp. 377–388.

[19] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis, "MemZip: Exploring unconventional benefits from memory compression," in *20th International Symposium on High Performance Computer Architecture*, 2014, pp. 638–649.

[20] R. Kanakagiri, B. Panda, and M. Mutyam, "MBZip: Multiblock Data Compression," *ACM Transaction on Architecture Code Optimization*, vol. 14, no. 4, pp. 42:1–42:29, 2017.

[21] Y. Wang, Y. Han, J. Zhou, H. Li, and X. Li, "DISCO: A Low Overhead in-network Data Compressor for Energy-Efficient Chip Multi-Processors," *Design Automation Conference*, pp. 1–6, 2016.

[22] Y. Wang, H. Li, Y. Han, and X. Li, "A Low Overhead In-Network Data Compressor for the Memory Hierarchy of Chip Multiprocessors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 6, pp. 1265–1277, 2018.

[23] C. Zhang and S. A. McKee, "Hardware-only Stream Prefetching and Dynamic Access Ordering," in *14th International Conference on Supercomputing*, 2000, pp. 167–175.

[24] E. Bhatia, G. Chacon, S. Pugsley, E. Teran, P. V. Gratz, and D. A. Jiménez, "Perceptron-Based Prefetch Filtering," in *46th International Symposium on Computer Architecture*, 2019, pp. 1–13.

[25] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti, "Path Confidence based Lookahead Prefetching," in *49th International Symposium on Microarchitecture*, 2016, pp. 1–12.

[26] J. Kim, E. Teran, P. V. Gratz, D. A. Jiménez, S. H. Pugsley, and C. Wilkerson, "Kill the Program Counter: Reconstructing Program Behavior in the Processor Cache Hierarchy," in *Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 737–749.

[27] S. Pakalapati and B. Panda, "Bouquet of Instruction Pointers: Instruction Pointer Classifier-based Spatial Hardware Prefetching," in *47th International Symposium on Computer Architecture*, 2020, pp. 118–131.

[28] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently Prefetching Complex Address Patterns," in *48th International Symposium on Microarchitecture*, 2015, pp. 141–152.

[29] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar, "A 5-GHz Mesh Interconnect for a Teraflops Processor," *IEEE Micro*, vol. 27, no. 5, pp. 51–61, 2007.

[30] M. Buckler, W. Burleson, and G. Sadowski, "Low-Power Networks-on-Chip: Progress and remaining challenges," in *International Symposium on Low Power Electronics and Design*, 2013, pp. 132–134.

[31] A. Ben Achballah, S. Ben Othman, and S. Ben Saoud, "Problems and Challenges of Emerging Technology Networks-on-Chip: A Review," *Microprocessors and Microsystems*, vol. 53, pp. 1–20, 2017.

[32] R. Das, A. K. Mishra, C. Nicopoulos, D. Park, V. Narayanan, R. Iyer, M. S. Yousif, and C. R. Das, "Performance and Power Optimization through Data Compression in Network-on-Chip Architectures," in *High Performance Computer Architecture*, 2008, pp. 215–225.

[33] S. Ogg and B. Al-Hashimi, "Improved Data Compression for Serial Interconnected Network on Chip through Unused Significant Bit Removal," in *19th International Conf, on VLSI Design*, Jan 2006.

[34] P. Zhou, B. Zhao, Y. Du, Y. Xu, Y. Zhang, J. Yang, and L. Zhao, "Frequent Value Compression in Packet-Based NoC Architectures," in *Asia and South Pacific Design Automation Conference*, 2009, p. 13âĂŞ18.

[35] D. Deb, J. Jose, and M. Palesi, "Performance Enhancement of Caches in TCMPs using Near Vicinity Prefetcher," in *32nd International Conference on VLSI Design and 18th International Conference on Embedded Systems*, 2019.

[36] D. Deb, J. Jose, and M. Palesi, "ECAP: Energy-Efficient Caching for Prefetch Blocks in Tiled Chip Multiprocessors," *IET Computers Digital Techniques*, vol. 13, no. 6, pp. 417–428, 2019.

[37] D. Deb and J. Jose, "ZPP: Zero Pollution Prefetcher for NUCA based MultiCore Architecture," *CSE, IITG Archive (Ready to be submitted).*

[38] D. Deb, J. Jose, and M. Palesi, "COPE: Reducing Cache Pollution and Network Contention by Inter-Tile Coordinated Prefetching in NoC-Based MPSoCs," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 26, no. 3, pp. 1–31, 2021.

[39] D. Deb, R. M.K., and J. Jose, "FlitZip: Effective Packet Compression for NoC in MultiProcessor System-on-Chip," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 1, pp. 117–128, 2022.

[40] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 Simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[41] C. Bienia, "Benchmarking Modern Multiprocessors," Ph.D. dissertation, Princeton University, 2011.

[42] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[43] M. Dalui and B. K. Sikdar, "An Efficient Test Design for CMPs Cache Coherence Realizing MESI Protocol," in *Progress in VLSI Design and Test*, 2012, pp. 89–98.

[44] P. Michaud, "Best-Offset Hardware Prefetching," in *High Performance Computer Architecture*, 2016, pp. 469–480.

[45] S. Palacharla and R. E. Kessler, "Evaluating Stream Buffers as a Secondary Cache Replacement," *SIGARCH Comput. Archit. News*, vol. 22, no. 2, p. 24âĂŞ33, 1994.

[46] J. Lira, C. Molina, and A. González, "HK-NUCA: Boosting Data Searches in Dynamic Non-Uniform Cache Architectures for Chip Multiprocessors," in *International Parallel & Distributed Processing Symposium*, 2011, pp. 419–430.

[47] J. Liptay, "Structural Aspects of the System/360 Model 85 II: The Cache," *IBM Syst. J.*, vol. 7, pp. 15–21, 1968.

[48] M. V. Wilkes, "Slave Memories and Dynamic Storage Allocation," *IEEE Transactions on Electronic Computers*, vol. EC-14, no. 2, pp. 270–271, 1965.

[49] L. A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[50] J. Archibald and J.-L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Trans. Comput. Syst.*, vol. 4, no. 4, p. 273âĂŞ298, 1986.

[51] W. Dally, "Virtual-channel flow control," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 2, pp. 194–205, 1992.

[52] Y. J. Yoon, N. Concer, M. Petracca, and L. Carloni, "Virtual Channels vs. Multiple Physical Networks: A Comparative Analysis," in *47th Design Automation Conference.*, 2010, pp. 162–165.

[53] P. Kogge, *The Architecture of Pipelined Computers.* Taylor & Francis, 1981.

[54] J. Lee, N. B. Lakshminarayana, H. Kim, and R. Vuduc, "Many-Thread Aware Prefetching Mechanisms for GPGPU Applications," in *International Symposium on Microarchitecture*, 2010, pp. 213–224.

[55] J. Lee, H. Kim, and R. Vuduc, "When Prefetching Works, When It Doesn't, and Why," *ACM Trans. on Architecture and Code Optimization*, vol. 9, no. 1, pp. 2:1–2:29, 2012.

[56] P. Conway and B. Hughes, "The AMD Opteron CMP NorthBridge architecture: Now and in the future," in *IEEE Hot Chips 18 Symposium*, 2006, pp. 1–30.

[57] J. Tendler, J. Dodson, J. S. Fields, H. le, and B. Sinharoy, "POWER4 System Microarchitecture," *IBM Journal of Research and Development*, vol. 46, no. 1, pp. 5–25, 2002.

[58] Tien-Fu Chen and Jean-Loup Baer, "Effective Hardware-based Data Prefetching for High-Performance Processors," *IEEE Transactions on Computers*, vol. 44, no. 5, pp. 609–623, 1995.

[59] Y. Chen, H. Zhu, and X. Sun, "An Adaptive Data Prefetcher for High-Performance Processors," in *International Conference on Cluster, Cloud and Grid Computing*, 2010, pp. 155–164.

[60] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "An Efficient Temporal Data Prefetcher for L1 Caches," *IEEE Computer Architecture Letters*, vol. 16, no. 2, pp. 99–102, 2017.

[61] F. Golshan, M. Bakhshalipour, M. Shakerinava, A. Ansari, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Harnessing Pairwise-Correlating Data Prefetching With Runahead Metadata," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 130–133, 2020.

[62] C. Zhang, Y. Zeng, J. Shalf, and X. Guo, "RnR: A Software-Assisted Record-and-Replay Hardware Prefetcher," in *International Symposium on Microarchitecture*, 2020, pp. 609–621.

[63] M. J. Liao and J. Sampson, "D-SOAP: Dynamic Spatial Orientation Affinity Prediction for Caching in Multi-Orientation Memory Systems," in *International Symposium on Microarchitecture*, 2020, pp. 581–595.

[64] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "IMP: Indirect Memory Prefetcher," in *International Symposium on Microarchitecture*, 2015, p. 178âĂŞ190.

[65] S. Somogyi, T. F. Wenisch, A. Ailamaki, and B. Falsafi, "Spatio-Temporal Memory Streaming," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, p. 69âĂŞ80, 2009.

[66] S. H. Pugsley, Z. Chishti, C. Wilkerson, P. Chuang, R. L. Scott, A. Jaleel, S. Lu, K. Chow, and R. Balasubramonian, "Sandbox Prefetching: Safe Run-time Evaluation of Aggressive Prefetchers," in *High Performance Computer Architecture*, 2014, pp. 626–637.

[67] O. Ozturk, S. W. Son, M. Kandemir, and M. Karakoy, "Prefetch Throttling and Data Pinning for Improving Performance of Shared Caches," in *ACM/IEEE Conference on Supercomputing*, 2008, pp. 1–12.

[68] S. Ainsworth and T. M. Jones, "Software Prefetching for Indirect Memory Accesses: A Microarchitectural Perspective," *ACM Trans. Comput. Syst.*, vol. 36, no. 3, 2019.

[69] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham, "Effective Stream-based and Execution-based Data Prefetching," in *18th International Conference on Supercomputing*, 2004, pp. 1–11.

[70] Y. Ishii, M. Inaba, and K. Hiraki, "Access Map Pattern Matching for Data Cache Prefetch," in *Supercomputing*, 2009, pp. 499–500.

[71] M. Shakerinava, M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Multi-Lookahead Offset Prefetching," 2019.

[72] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo Spatial Data Prefetcher," in *International Symposium on High Performance Computer Architecture*, 2019, pp. 399–411.

[73] N. C. Nachiappan, A. K. Mishra, M. Kandemir, A. Sivasubramaniam, O. Mutlu, and C. R. Das, "Application-Aware Prefetch Prioritization in On-Chip Networks," in *International Conference on Parallel Architectures and Compilation Techniques*, 2012, pp. 441–442.

[74] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt, "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers," in *High Performance Computer Architecture*, 2007, pp. 63–74.

[75] E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Techniques for Bandwidth-Efficient Prefetching of Linked Data Structures in Hybrid Prefetching Systems," in *15th International Symposium on High Performance Computer Architecture*, 2009, pp. 7–17.

[76] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt, "Coordinated Control of Multiple Prefetchers in Multi-core Systems," in *42nd IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 316–326.

[77] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning," in *44th IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 374–385.

[78] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Prefetch-Aware Shared-Resource Management for Multi-core Systems," in *38th International Symposium on Computer Architecture*, 2011, pp. 141–152.

[79] J. Yu and P. Liu, "A Thread-Aware Adaptive Data Prefetcher," in *32nd International Conference on Computer Design*, 2014, pp. 278–285.

[80] W. Heirman, K. D. Bois, Y. Vandriessche, S. Eyerman, and I. Hur, "Near-side Prefetch Throttling: Adaptive Prefetching for High-performance Many-core Processors," in *27th International Conference on Parallel Architectures and Compilation Techniques*, 2018, pp. 28:1–28:11.

[81] P. Yedlapalli, J. Kotra, E. Kultursay, M. Kandemir, C. R. Das, and A. Sivasubramaniam, "Meeting Midway: Improving CMP Performance with Memory-Side Prefetching," in *22nd International Conference on Parallel Architectures and Compilation Techniques*, 2013, pp. 289–298.

[82] S.-w. Liao, T.-H. Hung, D. Nguyen, C. Chou, C. Tu, and H. Zhou, "Machine Learning-based Prefetch Optimization for Data Center Applications," in *High Performance Computing Networking, Storage and Analysis*, 2009, pp. 56:1–56:10.

[83] V. Seshadri, S. Yedkar, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Mitigating Prefetcher-Caused Pollution Using Informed Caching Policies for Prefetched Blocks," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, pp. 51:1–51:22, 2015.

[84] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, "The Evicted-address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing," in *21st International Conference on Parallel Architectures and Compilation Techniques*, 2012, pp. 355–366.

[85] V. Jiménez, R. Gioiosa, F. J. Cazorla, A. Buyuktosunoglu, P. Bose, and F. P. O'Connell, "Making Data Prefetch Smarter: Adaptive Prefetching on POWER7," in *21st International Conference on Parallel Architectures and Compilation Techniques*, 2012, pp. 137–146.

[86] V. Jimenez, A. Buyuktosunoglu, P. Bose, F. P. O'Connell, F. Cazorla, and M. Valero, "Increasing Multicore System Efficiency through Intelligent Bandwidth Shifting," in *International Symposium on High Performance Computer Architecture*, 2015, pp. 39–50.

[87] C. Wu, A. Jaleel, M. Martonosi, S. C. Steely, and J. Emer, "PACMan: Prefetch-Aware Cache Management for High Performance Caching," in *44th IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 442–453.

[88] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Prefetch-Aware Shared-Resource Management for Multi-Core Systems," in *International Symposium on Computer Architecture*, 2011, pp. 141–152.

[89] Albericio Jorge, Gran Rubén, Ibáñez Pablo, Viñals Víctor Llabería and Jose María, "ABS:A Low-cost Adaptive Controller for Prefetching in a Banked Shared Last-level Cache," *ACM Trans. on Architecture and Code Optimization*, vol. 8, no. 4, p. 19, 2012.

[90] A.Aziz, M. Cireno, and E. B. et al., "Balanced Prefetching Aggressiveness Controller for NoC-based Multiprocessor," in *27th Symposium on Integrated Circuits and Systems Design*, 2014, pp. 1–7.

[91] N. C. Nachiappan, A. K. Mishra, M. Kandemir, A. Sivasubramaniam, O. Mutlu, and C. R. Das, "Application-Aware Prefetch Prioritization in On-Chip Networks," in

International Conference on Parallel Architectures and Compilation Techniques, 2012, pp. 441–442.

[92] J. Lee, H. Kim, M. Shin, J. Kim, and J. Huh, "Mutually Aware Prefetcher and On-Chip Network Designs for Multi-Cores," *IEEE Transactions on Computers*, vol. 63, no. 9, pp. 2316–2329, 2014.

[93] T. Sherwood, S. Sair, and B. Calder, "Predictor-directed Stream Buffers," in *International Symposium on Microarchitecture*, 2000, pp. 42–53.

[94] Y. Huang, Z. Gu, J. Tang, M. Cai, J. Zhang, and N. Zheng, "Reducing Cache Pollution of Threaded Prefetching by Controlling Prefetch Distance," in *International Parallel and Distributed Processing Symposium Workshops*, 2012, pp. 1812–1819.

[95] M. Cavus, R. Sendag, and J. J. Yi, "Informed Prefetching for Indirect Memory Accesses," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 1, 2020.

[96] A. Naithani, J. Feliu, A. Adileh, and L. Eeckhout, "Precise Runahead Execution," in *International Symposium on High Performance Computer Architecture*, 2020, pp. 397–410.

[97] A. Jain and C. Lin, "Linearizing Irregular Memory Accesses for Improved Correlated Prefetching," in *46th International Symposium on Microarchitecture*, 2013, p. 247âĂŞ259.

[98] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Domino Temporal Data Prefetcher," in *International Symposium on High Performance Computer Architecture*, 2018, pp. 131–142.

[99] M. Bakhshalipour, S. Tabaeiaghdaei, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Evaluation of Hardware Data Prefetchers on Server Processors," *ACM Comput. Surv.*, vol. 52, no. 3, pp. 1–29, 2019.

[100] N. Neves, P. TomÃąs, and N. Roma, "Compiler-Assisted Data Streaming for Regular Code Structures," *IEEE Transactions on Computers*, vol. 70, no. 3, pp. 483–494, 2021.

[101] N. C. Crago, M. Stephenson, and S. W. Keckler, "Exposing Memory Access Patterns to Improve Instruction and Memory Efficiency in GPUs," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 4, 2018.

[102] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Practical off-chip Meta-data for Temporal Memory Streaming," in *International Symposium on High Performance Computer Architecture*, 2009, pp. 79–90.

[103] X. Lu, R. Wang, and X. H. Sun, "APAC: An Accurate and Adaptive Prefetch Framework with Concurrent Memory Access Analysis," in *International Conference on Computer Design*, 2020, pp. 222–229.

[104] L. M. AlBarakat, P. V. Gratz, and D. A. Jiménez, "SB-Fetch: Synchronization Aware Hardware Prefetching for Chip Multiprocessors," in *International Conference on Supercomputing*, 2020, pp. 1–12.

[105] L. M. AlBarakat, P. V. Gratz, and D. A. JimÃľnez, "MTB-Fetch: Multithreading Aware Hardware Prefetching for Chip Multiprocessors," *IEEE Computer Architecture Letters*, vol. 17, no. 2, pp. 175–178, 2018.

[106] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial Memory Streaming," in *33rd International Symposium on Computer Architecture*, 2006, pp. 252–263.

[107] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block Prediction &; Dead-block Correlating Prefetchers," in *28th International Symposium on Computer Architecture*, 2001, pp. 144–154.

[108] D. A. Varkey, B. Panda, and M. Mutyam, "RCTP: Region Correlated Temporal Prefetcher," in *International Conference on Computer Design*, 2017, pp. 73–80.

[109] S. Kondguli and M. Huang, "Division of Labor: A More Effective Approach to Prefetching," in *International Symposium on Computer Architecture*, 2018, pp. 83–95.

[110] P. Xiang, Y. Yang, M. Mantor, N. Rubin, and H. Zhou, "Revisiting ILP Designs for Throughput-Oriented GPGPU Architecture," in *International Symposium on Cluster, Cloud and Grid Computing*, 2015, pp. 121–130.

[111] X. Zhuang and H. S. Lee, "Reducing Cache Pollution via Dynamic Data Prefetch Filtering," *IEEE Transactions on Computers*, vol. 56, no. 1, pp. 18–31, 2007.

[112] V. Selfa, J. Sahuquillo, M. E. GÃşmez, and C. GÃşmez, "Efficient Selective Multicore Prefetching under Limited Memory Bandwidth," *Journal of Parallel and Distributed Computing*, vol. 120, pp. 32 – 43, 2018.

[113] S. H. Pugsley, Z. Chishti, C. Wilkerson, P. Chuang, R. L. Scott, A. Jaleel, S. Lu, K. Chow, and R. Balasubramonian, "Sandbox Prefetching: Safe Run-time Evaluation of Aggressive Prefetchers," in *20th International Symposium on High Performance Computer Architecture*, 2014, pp. 626–637.

[114] F. Dahlgren, M. Dubois, and P. Stenstrom, "Sequential Hardware Prefetching in Shared-Memory Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 7, pp. 733–746, 1995.

[115] M. Torrents, R. Martínez, and C. Molina, "Facing Prefetching Challenges in Distributed Shared Memories for CMPs," *J. Supercomput.*, vol. 72, no. 4, pp. 1453–1476, 2016.

[116] M. Wu, Y. Pei, L. Yu, T. Chen, X. Lou, and T. Zhang, "WAP: The Warp Feature Aware Prefetching Method for LLC on CPU-GPU Heterogeneous Architecture," in *International Conference on HPCC/SmartCity/DSS*, 2016, pp. 414–421.

[117] J. Chen, X. Tao, Z. Yang, J. Peir, X. Li, and S. Lu, "Guided Region-Based GPU Scheduling: Utilizing Multi-thread Parallelism to Hide Memory Latency," in *International Symposium on Parallel and Distributed Processing*, 2013, pp. 441–451.

[118] N. D. E. Jerger, E. L. Hill, and M. H. Lipasti, "Friendly fire: Understanding the Effects of Multiprocessor Prefetches," in *International Symposium on Performance Analysis of Systems and Software*, 2006, pp. 177–188.

[119] C. Navarro, J. Feliu, S. Petit, M. E. GÃşmez, and J. Sahuquillo, "Bandwidth-Aware Dynamic Prefetch Configuration for IBM POWER8," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 8, pp. 1970–1982, 2020.

[120] M. Cireno, A. Aziz, and E. Barros, "Temporized Data Prefetching Algorithm for NoC-based Multiprocessor Systems," in *27th International Conference on Application-specific Systems, Architectures and Processors*, 2016, pp. 235–236.

[121] R. Boyapati, J. Huang, P. Majumder, K. H. Yum, and E. J. Kim, "APPROX-NoC: A Data Approximation Framework for Network-On-Chip Architectures," in *44th International Symposium on Computer Architecture*, 2017, pp. 666–677.

[122] Y. Chen and A. Louri, "An Approximate Communication Framework for Network-on-Chips," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 6, pp. 1434–1446, 2020.

[123] Y. Chen, M. F. Reza, and A. Louri, "DEC-NoC: An Approximate Framework Based on Dynamic Error Control with Applications to Energy-Efficient NoCs," in *International Conference on Computer Design*, 2018, pp. 480–487.

[124] H. Kim, B. Grot, P. V. Gratz, and D. A. JimÃĺnez, "Spatial Locality Speculation to Reduce Energy in Chip-Multiprocessor Networks-on-Chip," *IEEE Transactions on Computers*, vol. 63, no. 3, pp. 543–556, 2014.

[125] H. Kim, P. Ghoshal, B. Grot, P. V. Gratz, and D. A. Jiménez, "Reducing Network-on-Chip Energy Consumption through Spatial Locality Speculation," in *5th International Symposium on Networks-on-Chip*, 2011, pp. 233–240.

[126] H. Kim and P. Gratz, "Leveraging Unused Cache Block Words to Reduce Power in CMP Interconnect," *IEEE Computer Architecture Letters*, vol. 9, no. 1, pp. 33–36, 2010.

[127] V. Y. Raparti and S. Pasricha, "DAPPER: Data Aware Approximate NoC for GPGPU Architectures," in *IEEE/ACM International Symposium on Networks-on-Chip*, 2018, pp. 1–8.

[128] A. B. Kahng, B. Li, L. Peh, and K. Samadi, "ORION 2.0: A Fast and Accurate NoC Power and Area Model for Early-Stage Design Space Exploration," in *Design, Automation Test in Europe Conference Exhibition*, 2009, pp. 423–428.

[129] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Cacti 6.0: A Tool to Model Large Caches," *HP Laboratories*, pp. 22–31, 2009.

[130] Jiang *et al.*, "A Detailed and Flexible Cycle-Accurate Network-on-Chip Simulator," in *Performance Analysis of Systems and Software*, 2013, pp. 86–96.

[131] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[132] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt, "The M5 Simulator: Modeling Networked Systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, 2006.

[133] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's General Execution-Driven

Multiprocessor Simulator (GEMS) Toolset," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, p. 92âĂŞ99, 2005.

[134] N. Agarwal, T. Krishna, L.-S. Peh, and N. Jha, "GARNET: A Detailed On-Chip Network Model inside a Full-System Simulator," in *International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 33–42.

[135] Kun Luo, J. Gummaraju, and M. Franklin, "Balancing Thoughput and Fairness in SMT Processors," in *International Symposium on Performance Analysis of Systems and Software*, 2001, pp. 164–171.

[136] J.-K. Peir, S.-C. Lai, S.-L. Lu, J. Stark, and K. Lai, "Bloom Filtering Cache Misses for Accurate Data Speculation and Prefetching," in *16th International Conference on Supercomputing*, 2002, pp. 189–198.

[137] N. Beckmann and D. Sanchez, "Modeling Cache Performance beyond LRU," in *International Symposium on High Performance Computer Architecture*, 2016, pp. 225–236.

[138] S. M. Khan, Y. Tian, and D. A. Jimenez, "Sampling Dead Block Prediction for Last-Level Caches," in *43rdl IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 175–186.

[139] P. Faldu and B. Grot, "Leeway: Addressing Variability in Dead-Block Prediction for Last-Level Caches," in *26th International Conference on Parallel Architectures and Compilation Techniques*, 2017, pp. 180–193.

[140] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith, "AC/DC: An Adaptive Data Cache Prefetcher," in *13th International Conference on Parallel Architecture and Compilation Techniques*, 2004, pp. 135–145.

[141] D. Deb, J. Jose, S. Das, and H. K. Kapoor, "Cost Effective Routing Techniques in 2D Mesh NoC using On-Chip Transmission Lines," *Journal of Parallel and Distributed Computing*, vol. 123, pp. 118 – 129, 2019.

[142] R. Bera, A. V. Nori, O. Mutlu, and S. Subramoney, "DSPatch: Dual Spatial Pattern Prefetcher," in *Microarchitecture*, 2019, pp. 531–544.

[143] H. W. Cain and P. Nagpurkar, "Runahead Execution vs. Conventional Data Prefetching in the IBM POWER6 Microprocessor," in *IEEE International Symposium on Performance Analysis of Systems Software*, 2010, pp. 203–212.

List of Publications

Publications from Thesis Work:

Refereed Journals:

1. **Dipika Deb**, Rohith M.K., and John Jose, "FlitZip: Effective Packet Compression for NoC in MultiProcessor System-on-Chip ", *IEEE Transactions on Parallel and Distributed Systems (IEEE TPDS)*, Volume 33, Issue 1, pp. 117-128, January 2022.
   DOI: 10.1109/TPDS.2021.3090315

2. **Dipika Deb**, John Jose, and Maurizio Palesi, "COPE: Reducing Cache Pollution and Network Contention by Inter-tile Coordinated Prefetching in NoC-based MPSoCs", *ACM Transactions on Design Automation and Electronics Systems (ACM TODAES)*, Volume 26, Issue 3, pp. 1-31, February 2021.
   DOI: 10.1145/3428149

3. **Dipika Deb**, John Jose, and Maurizio Palesi, "ECAP: Energy Efficient CAching for Prefetch Blocks in Tiled Chip MultiProcessors", *Journal of IET Computers & Digital Techniques (IET-CDT)*, Volume 13, Issue 6, pp. 417-428, August 2019.
   DOI: https://doi.org/10.1049/iet-cdt.2019.0035

4. **Dipika Deb**, and John Jose, "ZPP: Zero Pollution Prefetcher for NUCA based ManyCore Architecture", *Major revision submitted to ACM TACO.*

Refereed Conferences:

5. **Dipika Deb**, John Jose and Maurizio Palesi, "Performance Enhancement of Caches in TCMPs using Near Vicinity Prefetcher", *32nd International Conference on VLSI Design and 18th International Conference on Embedded Systems (VLSID)*, 2019, pp. 13-18.
   DOI: *10.1109/VLSID.2019.00021*

Publications other than Thesis Work:

1. **Dipika Deb**, Shirshendu Das, John Jose, and Hemangee K. Kapoor, "Cost Effective Routing Techniques in 2D Mesh NoC using On-Chip Transmission Lines", *Elsevier Journal of Parallel and Distributed Computing (Elsevier JPDC)*, Volume 123, pp. 118-129, January 2019.
   DOI: `https://doi.org/10.1016/j.jpdc.2018.09.009`

2. S. Chakraborty, **Dipika Deb**, D. Buragohain and H. K. Kapoor, "Cache Capacity and its Effects on Power Consumption for Tiled Chip Multi-Processors ", *International Conference on Electronics and Communication Systems (ICECS)* , 2014, pp. 1-6.
   DOI: *10.1109/ECS.2014.6892719*

3. **Dipika Deb**, Kashyap Matoo, "Caching Strategy for Prefetch Blocks in TCMPs", ***[Best Poster Award]*** *In Student Research Symposium of 25th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2018

4. *Dipika Deb*, Harsh Motwani, John Jose, "CLAP: Correlated LookAhead Delta Prefetcher", *CSE, IITG Archive (Ready to be submitted).*

Vitae

Dipika Deb joined the Ph.D. programme in the Department of Computer Science and Engineering at Indian Institute of Technology Guwahati, India in July 2016. In IIT Guwahati, she was affiliated with MARS Lab in the Department of Computer Science and Engineering. She received her Master of Technology in Computer Science and Engineering from Indian Institute of Technology Guwahati, India in 2016. Prior to that, she received her Bachelor of Technology in Computer Science and Engineering from Tezpur University in 2013. Her Ph.D. work focuses on improving the performance of tiled multicore architectures using prefetching and NoC packet compression. Her current research interests also include multicore architecture, security issues in TCMP, DNN accelerators, GPUs, and neuromorphic computing.

Contact Information

E-mail: dipika131991@gmail.com

Phone: +91-9435875127

Website: http://www.iitg.ac.in/stud/d.dipika/