DATA TRAFFIC REDUCTION BY EXPLOITING DATA CRITICALITY WITH A COMPRESSOR IN GPU

A Thesis

by

JAEGUEN BYOUN

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2018

Major Subject: Computer Science

ABSTRACT

Graphics Processing Units (GPUs) have been predominantly accepted for various general purpose applications due to a massive degree of parallelism. The demand for large-scale GPUs processing an enormous volume of data with high throughput has been rising rapidly. However, the performance of the massive parallelism workloads usually suffer from multiple constraints such as memory bandwidth, high memory latency, and power/energy cost. Also a bandwidth efficient network design is challenging in large-scale GPUs.

In this research, we focus on mitigating network bottlenecks by effectively reducing the size of packets transferring through an interconnect network so that the overall system performance improves.

The unused fraction of each L1 data cache block across a variety of benchmark suits is initially investigated to see inefficient cache usage. Then, categorizing memory access patterns into several types we introduce essential micro-architectural enhancements to support filtering out unnecessary words in packets throughout the reply path. A compression scheme (Dual Pattern Compression) adequate for packet compression is exploited to effectively reduce the size of reply packets. We demonstrate that our scheme effectively improves system performance. Our approach yields 39% IPC improvement across heterogeneous computing and text processing benchmarks over the baseline cooperating with DPC. Comparing this work with DPC, we achieved 5% IPC improvement for the overall benchmark suits and 20% IPC increase for favorable workloads to this scheme.

DEDICATION

To my family and the Republic of Korea Army.

ACKNOWLEDGMENTS

I would sincerely like to thank my advisor, Professor Eun Jung Kim, for her guidance, support, and advice during this research. Her computer architecture course gave me the basic concepts and inspiration to do this work.

I am grateful to Professor Duncan M. "Hank" Walker, Jiang Hu for being my thesis committee and their valuable feedback.

I would also like to thank all my colleagues in the High Performance Computing Laboratory as well as in the Republic of Korea military. They have helped me not only enhance my knowledge but also encourage me throughout the course of this study. Special thanks to Kyung Hoon Kim who provided me with the assistance and materials to do this work.

I would like to acknowledge the Republic of Korea Army for providing this wonderful opportunity to explore this work.

Last but not least, I am deeply grateful to my family for their consistent support and love.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supervised by a thesis committee consisting of my advisor, Dr. Eun Jung Kim of the Department of Computer Science and Engineering, Dr. Duncan M. "Hank" Walker of the Department of Computer Sicence and Engineering, and Dr. Jiang Hu of the Department of Electrical and Computer Engineering.

All work for the thesis was completed by the student, in collaboration with Kyung Hoon Kim of the Department of Computer Science and Engineering

Funding Sources

Graduate Study was supported by a scholarship from the Republic of Korea Army.

NOMENCLATURE

DPC	Dual Pattern Compressor
FDR	Full Data Request
PDR	Partial Data Request
CA	Consistent Memory Accesses
ICA	Inconsistent Memory Accesses
FM	First Miss
SM	Subsequent Miss
HI	Hit but Invalid
DF-MAN	Data Filtering with Manipulation method
DF-TRUC	Data Filtering with Truncation method

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

TABLE Page

1. INTRODUCTION

1.1 The Characteristics of GPUs

GPUs have been extensively utilized in a variety of general purpose applications due to their powerful computing capability. Recent applications require an even more powerful computation ability to deal with a huge volume of data with higher throughput. The rising demands have been satisfied by the continuing development of GPUs. In fact, an NVIDIA Fermi GPU (GTX480) released in 2010 started with 480 cores and the recently released NVIDIA Titan-XP incorporates 3840 cores. Even these GPUs are not enough for rapidly evolving AI applications that tackle large datasets, so multiple GPUs are often used together to facilitate faster processing. Therefore, it is paramount to design a large-scale GPU to support higher degrees of parallelism. However, the performance of the massive parallelism workloads usually suffer from multiple constraints such as architectural imbalanced design, memory bandwidth (bandwidth wall [1]), high memory latency [2], and power/energy cost.

1.1.1 Execution Model of GPU

Contemporary GPUs architecture contains a scalable array of multithreaded *Streaming Multi-processors*(SMs in NVIDIA terminology) [3]. A GPU program is invoked by the host CPU and the launched program on a GPU device is called a kernel. The host refers to the CPU and its memory. The device denotes the GPU and its memory.

A thread is defined as a sequential execution unit. In GPU programming, all threads execute the same sequential program and they run in parallel. SMs are developed to run numerous threads concurrently. The *Single Instruction, Multiple Threads*(SIMT) architectural model is applied to manage such a large amount of threads. A group of threads is known as a threads block. Threads within a block can cooperate to exchange data with light-weight synchronization. SMs create, schedule, and execute threads in groups of 32 parallel threads known as *warps*. An array of thread blocks are also named as a grid. When a grid is given to an SM, it makes them into several warps.

Warps are scheduled by a warp scheduler to be executed in each SM.

1.1.2 Programming Model of GPU

CUDA is a parallel computing platform and application programming interface(API) model developed by NVIDIA [3]. It enables NVIDIA GPUs to implement programs written with C, C++, and other languages. The CUDA threads are executed by a physically separate device. Both the host and the device have their own memory spaces in DRAM. The host initiates a data transaction to store necessary data for a parallel program in the device. Once the transaction is finished the host launches a kernel, then in turn the device begins computations in parallel with the execution model. After the host finishes the kernel execution, it copies the computed results back to the host. In this way, CUDA application codes execute on the host while the parallel code executes on the device.

1.1.3 Many-to-few-to-many Traffic Pattern

In GPU architecture, many SMs typically communicate with fewer MCs. Over the previous decades the computational units significantly scale up and MCs are still fewer than SMs due to on-chip pin bandwidth limitations [4].

There are two types of request categories, read requests and write requests. While write requests need to get the reply back with only an acknowledgement message, read requests require the heavier replies with the requested data coming from lower memory. It creates a traffic imbalance between requests and replies [5, 6]. From the traffic imbalance property, a higher volume of read replies creates bottlenecks in the reply network by stalling MCs.

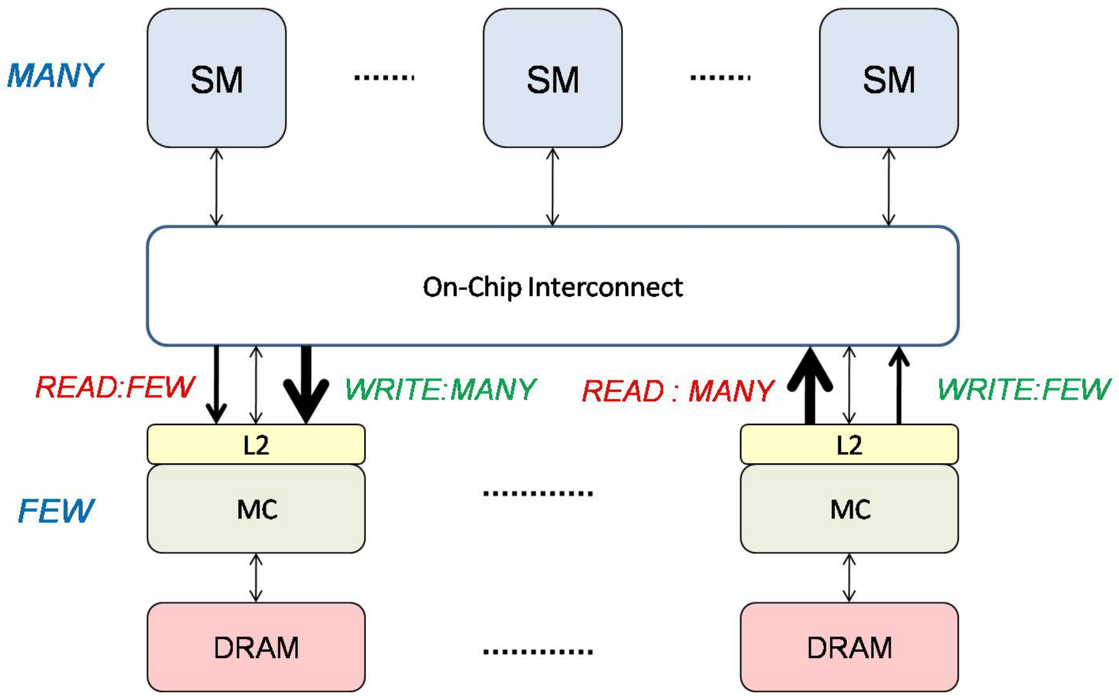

Figure 1.1: Overview of Many-to-few-to-many Property

Figure 1.1 depicts the Many-to-few-to-many properties in GPUs. Both architectural and traffic asymmetry cause overall system performance degradation. Thus, it is necessary to put a special treatment in this design.

1.1.4 On-chip Memory Types

Each shader core (SM) in GPUs has a load-store unit consisting of 4 different memory regions : data, texture, shared, constant. The L1 data cache deals with global memory that has a global scope and lifetime of an allocated program and local memory that has a local scope to each SM. Shared memory is utilized for efficiently managing global memory data between threads and its size can be dynamically partitioned with L1 data cache. For example, L1 data cache takes 16KB while shared memory is reserved 48KB, and vice versa. Constant memory is used for constants that cannot be compiled into the program. Texture memory is for general purpose computation to handle special cases such as fast interpolation on multi-dimensional arrays. Since global memory is primarily considered main memory, global memory requests transmitted through an interconnect network are chiefly discussed in this work.

1.1.5 Inefficient Usage of L1 Data Cache

While some application types, such as image processing workloads, can naturally benefit from the existing architecture due to regular memory access patterns, other types, irregular workloads, need a more sophisticated approach to address frequent branches and memory divergences that access memory irregularly [7].

The irregular access patterns to a L1 data cache block lead to the inefficient usage of the cache since a miss brings the entire range of data without being fully utilized until the eviction of a cache block. If replies transferring through an interconnect network have a synthetically reduced size such that partial necessary portion of 128 bytes that is really requested data from a SM instead of the entire cache block the traffic volume on the reply path can be alleviated.

1.2 Compression Techniques

Data compression techniques have been widely adopted as an instrumental way for improving network bandwidth and they show drawbacks depending on where they are applied. Most hardware compression algorithms fall into two types of schemes, dictionary-based compression schemes, and pattern-based compression schemes.

The dictionary-based compression schemes encode data words into the corresponding short ones in the dictionary. Such algorithms are effective in encoding large data blocks and files, but the complex synchronization and significant overheads for managing dictionary limits the applicability for packet compression.

For the pattern-based compression schemes, data is compressed based solely on the occurrences of predefined patterns. This simple characteristic is inherently suitable for packet compression.

1.3 Approach

Given these observations, the goal of this work is to mitigate the network bottlenecks so that the overall system performance improves by effectively reducing the size of packets transferring through an interconnect network between many SMs and fewer MCs.

First, we observe how L1 cache blocks are inefficiently accessed at runtime according to multiple application scopes (e.g., heterogeneous computing, and text processing workloads) and characterize such memory access patterns into several types according to accessing size and consistency.

Second, we propose essential micro-architectural enhancements to support filtering out unnecessary portions of each reply from the L2 cache. Our filtering mechanism mainly considers L1 data cache read misses.

Third, we introduce and compare two different techniques for reducing reply packet size. The size of a criticality aware reply packet is reduced by one of two data reduction methods collaborating with a simple but powerful compressor DPC.

Fourth, we design a memory request prediction technique to carefully determine whether the entire data or the partial critical data should be delivered from the lower memory.

Finally, we evaluate and analyze the proposed scheme across various applications in terms of performance, NoC traffics on the way of reply, ,compression ratio, and the impact of the request controller.

2. REQUEST PATTERN CHARACTERIZATION

2.1 Actual Usage of L1 Data Cache

In regular memory access pattern applications, a warp typically demands a single wide 128-byte memory request to the L1 data cache. This request smoothly accesses 32 consecutive 32-bit data exploiting the coalescing hardware in each SM if the accessing cache block is valid. However, heterogeneous applications with irregular memory access patterns can behave differently compared to traditional memory access pattern applications as a result of powerful massive thread level parallelism capabilities. We characterize two aspects of memory requests. First, the entire 128 byte data in a L1D cache block is not fully utilized until the block is evicted. We categorize memory access patterns into two types according to memory request size: partial data request (PDR) and full data request (FDR). Second, we classify multiple memory requests on the same cache block into two types depending if memory accesses are consistent on a region of a block: consistent access (CA) and inconsistent access (ICA).

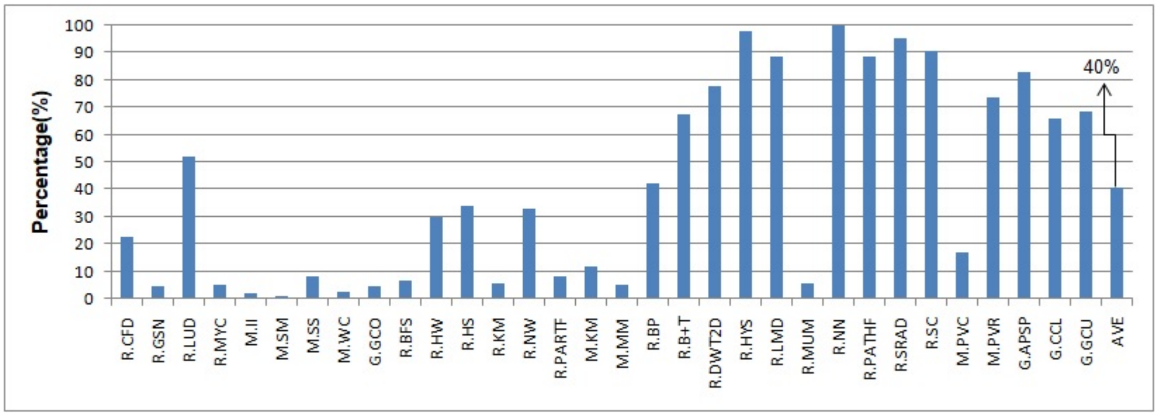

Figure 2.1: Averaged Percentage of Actual L1D Cache Block Usage until Eviction.

Figure 2.1 shows that the portion of cache block used on average across heterogeneous computing and text processing type applications. We simulate a large-scale GPGPUs which has 56 SMs, 8 MCs, and a 2D mesh topology interconnect. This scheme is implemented by a cycle-accurate GPU simulator, GPGPU-Sim 3.2.2 [8]. We take Rodinia benchmark [9], Graph benchmark suits for heterogeneous computing applications and Mars benchmark suits [10] for text processing applications. The benchmark leukocyte in Rodinia is intentionally not evaluated because global read memory requests are not observed although other types of memory requests occur. As the figure depicts, cache blocks are not efficiently used while they are serving in general. Only a portion of each cache block is utilized until eviction. On average, 40 percent of each cache block is used until eviction across all benchmarks. While R.HYS, R.NN, R.SRAD present a relatively efficient serving ratio of L1D cache (more than 90 percent actual usage), R.CFD, R.GSN, R.MYC, M.II, M.SM, M.SS and others (the left group workloads of the figure) present a remarkably poor level of L1D cache usage. This observation motivates the light weighted replies throughout the course of an interconnect if the unnecessary fraction of data is ignored just before they are injected into the interconnect.

2.2 Memory Access Patterns

We view memory access patterns on the L1D cache according to 32-byte granularity for more investigation. Accordingly, a sub-block of a cache block is defined 32-byte size (4 sub-blocks of each cache block in total). For instance, if a memory request on an L1D cache block needs only 4 byte size out of 128 bytes, we consider that the corresponding one sub-block (32 bytes) is accessed. While the cache block is still serving (not evicted), another memory request less than 32 bytes demands a different region of the block, then two sub-blocks (64-byte size) are acknowledged. Figure 2.2 plots how this memory access pattern measurement operates.

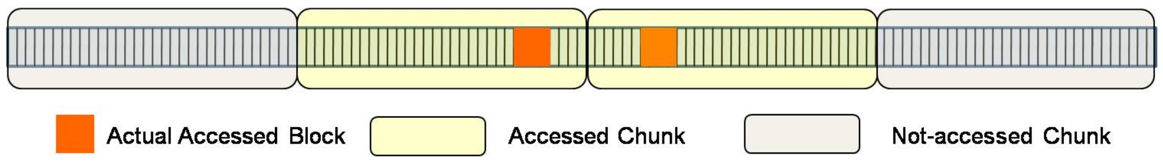

Figure 2.2: An Illustration of Memory Access Pattern Measurement.

In the following requests on the same block (the block is still not evicted), if one of the already accessed sub-blocks which is not exactly accessed is requested we acknowledge the same memory region access of the request.

We explore L1D cache block access patterns in detail at runtime in sub-blocks as well. Figure 2.3 shows the averaged percentage of accessed L1D cache blocks region while cache blocks are on the line. 32 byte access implies that only 1 sub-block is accessed until its eviction. 64 byte access, likewise, explains that two sub-blocks are used. 96 bytes are three sub-blocks, and 128 bytes are the entire usage of cache blocks.

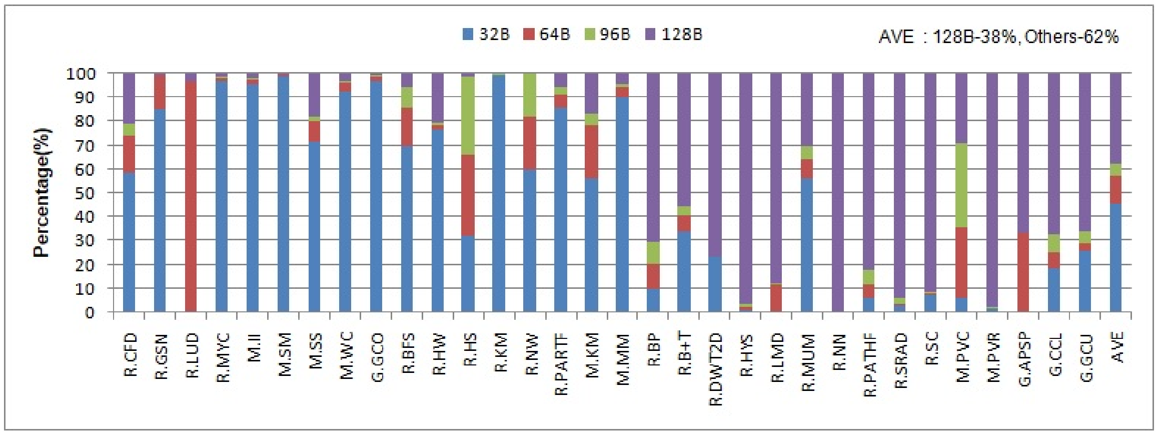

Figure 2.3: Averaged Percentage of L1D Cache Accessed Region until Eviction.

Some heterogeneous computing applications such as R.GSN, R.MYC, , R.PARTF, and G.GCO, show one sub-block memory access pattern until their eviction is dominant. In text processing applications, most applications except M.PVC, and M.PVR consist of 32-byte-size memory access on the L1D cache. This phenomenon is mainly because the necessary data requested by a warp is sparse. Heterogeneous computing applications often incur frequent branches and memory divergences [11] [12] and text processing applications are often interested in the data region around keywords.

While some applications showing the partial memory access pattern can benefit from a filtering critical data mechanism, other applications indicating full memory access patterns cannot take the advantage. Therefore, we need to further investigate the memory access patterns beneficial to us in order to develop a deeper insight. Memory accesses can be grouped into two types : consistent access (CA), and inconsistent access (ICA). With the CA type, multiple memory accesses on a same memory address are always contiguous regardless of access sizes. For the CA example, a memory request by a warp needs 2 sub-blocks (64 byte request). The corresponding reply from either L2 or DRAM is completed through the interconnect and is successfully filled with the data to that block. Over time, the cache block is accessed (cache hits) multiple times demanding only the filled partial data of the cache block until the cache is evicted.

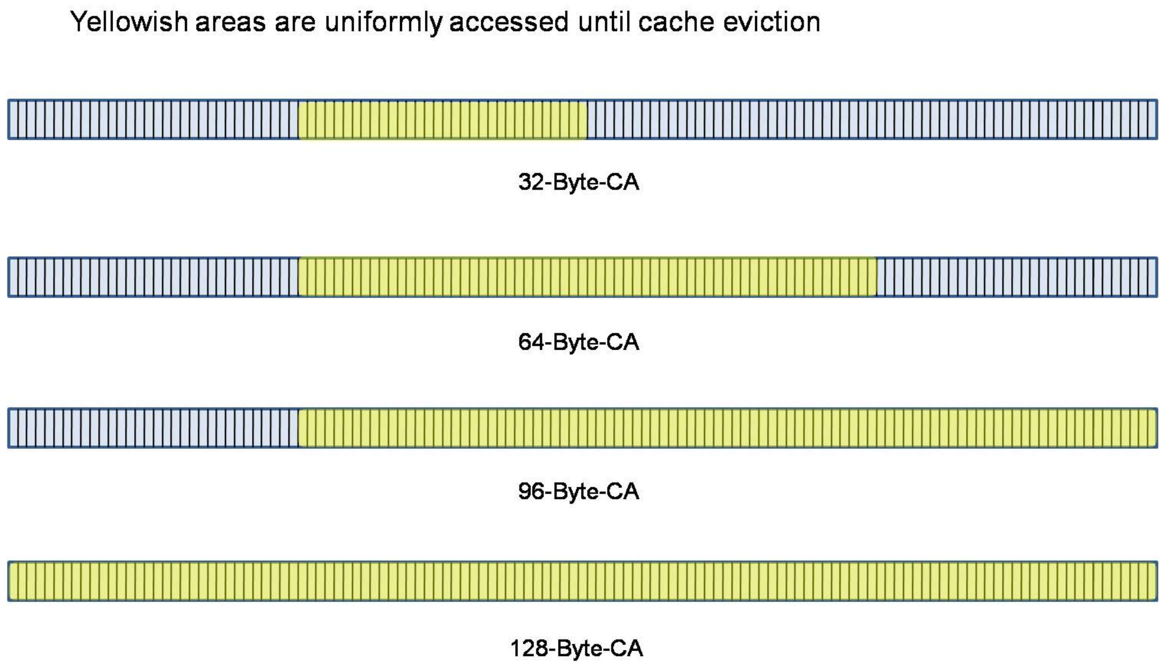

Figure 2.4: CA Type Examples

Figure 2.4 illustrates the CA types according to a memory access degree. Depending on the uniformly accessed size of a cache block by multiple memory requests, the CA can be classified into 4 subtypes, 32 bytes, 64 bytes, 96 bytes, and 128 bytes. On the other hand, the ICA can have

3 subtypes, 64 bytes, 96 bytes, and 128 bytes. 32 byte size is not contemplated for ICA because accessing only 32 bytes on a cache line until its eviction implies it is contiguously accessed only by one sub-block. Unlike the CA type, the ICA requests memory region irregularly. Once a cache line is filled with a portion of 128 bytes, another access to the cache block needs another region other than the filled area of the cache. Figure 2.5 illustrates the ICA type examples.

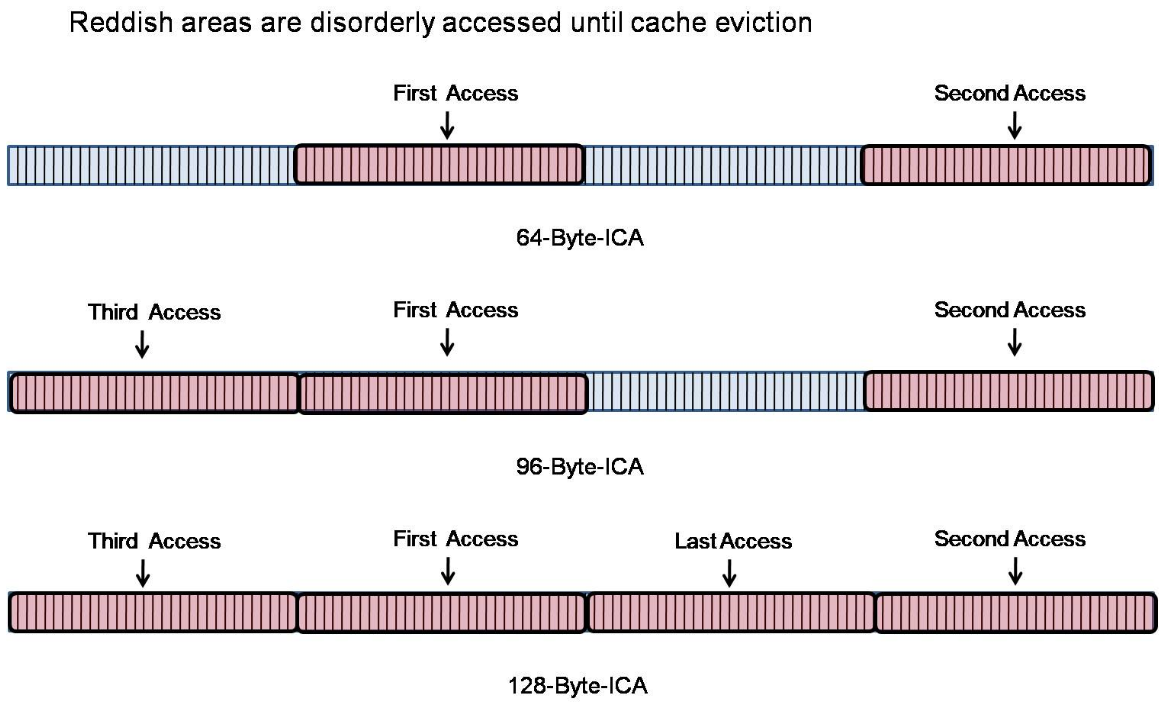

Figure 2.5: ICA (Inconsistent Access) Type Examples

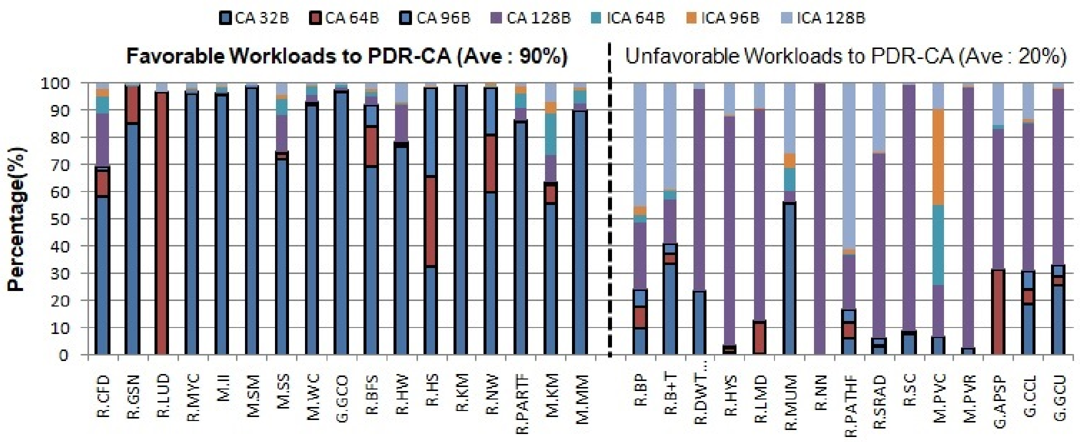

Figure 2.6: Memory Access Pattern According to CA, ICA

Figure 2.6 denotes the ratio of each CA and ICA across the introduced applications. Highlighted with bold represents partial CAs (32B, 64B, 96B). We can benefit only from favorable benchmarks to partial CAs. Since using cache blocks fully signifies cache blocks are used efficiently we cannot take advantage of the 128B CA type. Also, it is almost impossible to expect in advance a memory access behavior with ICA types. Therefore, we categorize the benchmarks into two groups according to partial CA patterns. The workloads in the left of the figure show a promising potential to improve performance. They present on average 90 percentage of partial CAs. Thus, it is reasonable if a mechanism is employed that fetches only partial data supposed to be utilized from the on-chip L2 cache it can alleviate the network bottleneck of the interconnect so that the overall system performance improves.

3. DATA FILTERING

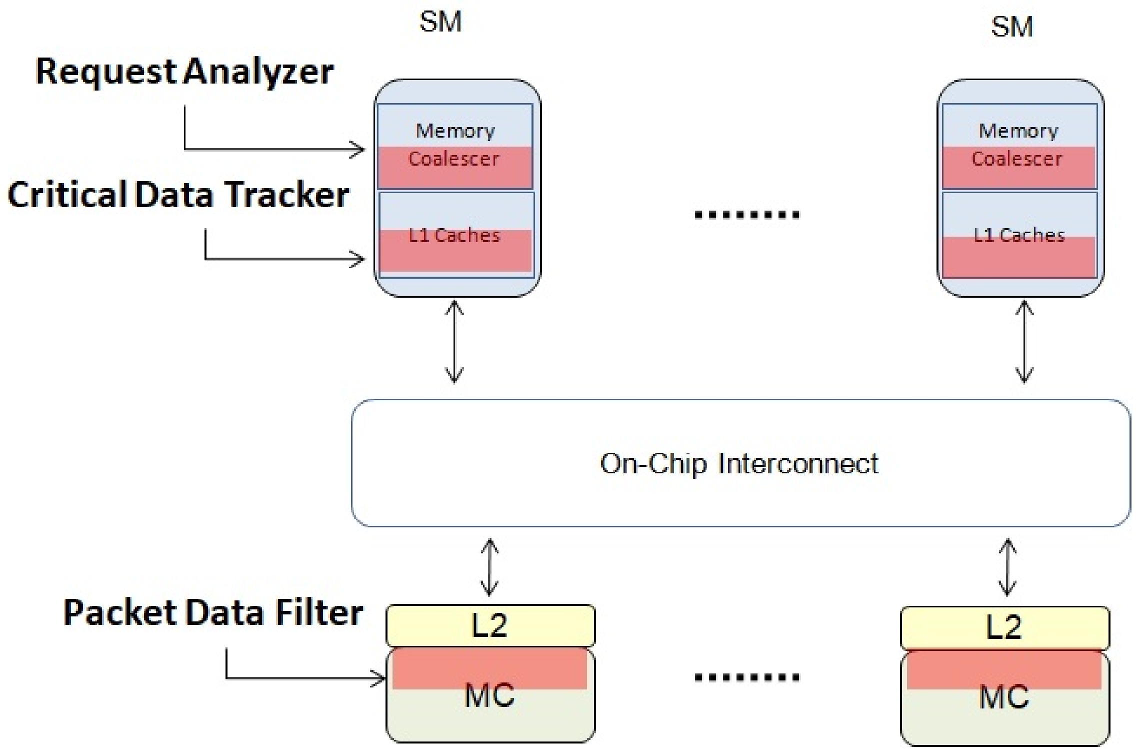

Figure 3.1: Illustration of Noncritical Words Detection and Data Filtering Mechanism

A global memory miss on the L1 data cache is sent to an MC. The packet is de-packetized into flits to be transferred through an on-chip network interface. The request is stored in a request queue when the queue has an free space. When the data request is completed by the L2 cache or DRAM, the data is stored in a reply queue and its reply packet is returned back to the requesting SM through the network interface. Our non-critical words detection and data filtering scheme consists of 3 main components as Figure 3.1 shows: Request Analyzer, Critical Data Tracker, and Packet Data Filter. Request Analyzer captures critical words and resides in the memory coalescing unit. Critical words are defined as requested data by a warp in this work. Critical Data Tracker keeps track of the existing data in a cache block by referring to its bitmap extension. Packet Data Filter is incorporated between MCs and an interconnect to refine reply packets creating smaller sized packets with an already developed compressor, DPC.

3.1 Request Analyzer

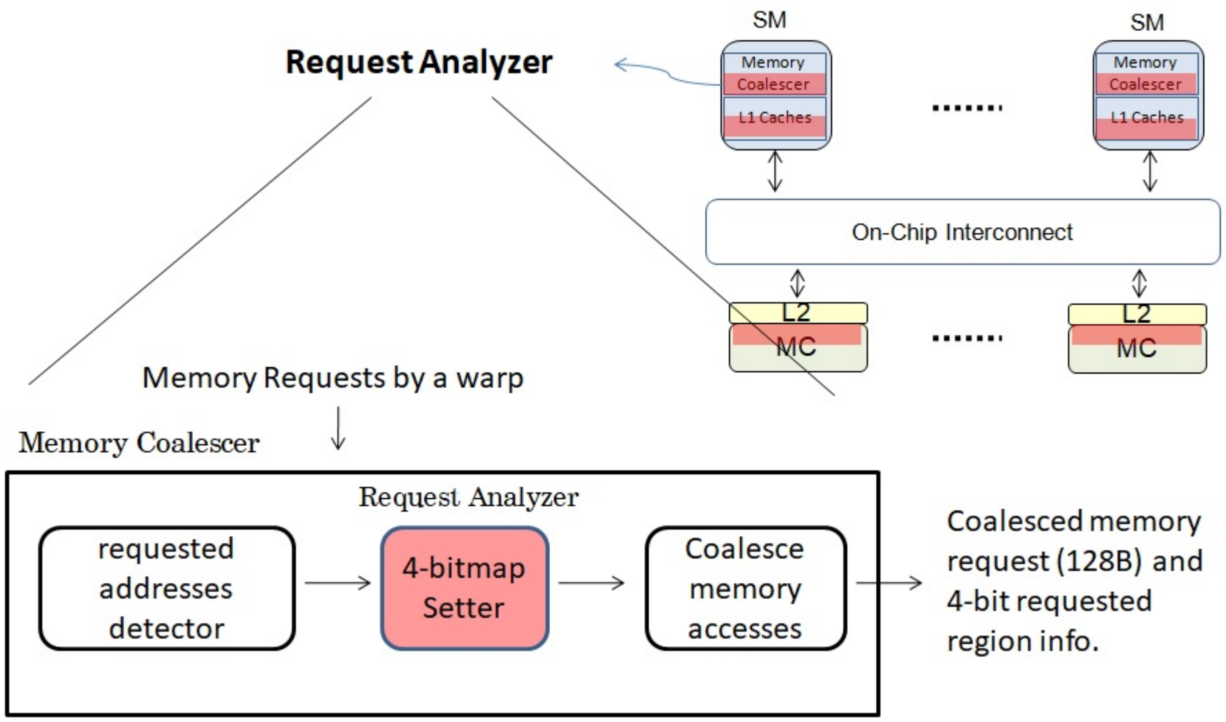

Figure 3.2: Illustration of Request Analyzer

In this data filtering mechanism, a memory request stores not only an accessing cache block address but also the *N*-bitmap where the *i*th bit indicates whether the *i*th data region (32-Byte granularity is used by default) is needed by a warp. The *N*-bitmap is computed based on the accessing byte information from the memory coalescing unit in an SM. The *N*-bitmap makes sure that the data region corresponding to ones of the *N*-bitmap are critical. Since 32 byte granularity is taken into account, a *4*-bitmap setter is introduced in each memory coalescing unit. A *4*-bitmap setter is situated as Figure 3.2 shows. Therefore, the output of the coalescing unit is a 128 byte coalesced memory request and a *4*-bit critical requested memory region.

3.2 Critical Data Tracker

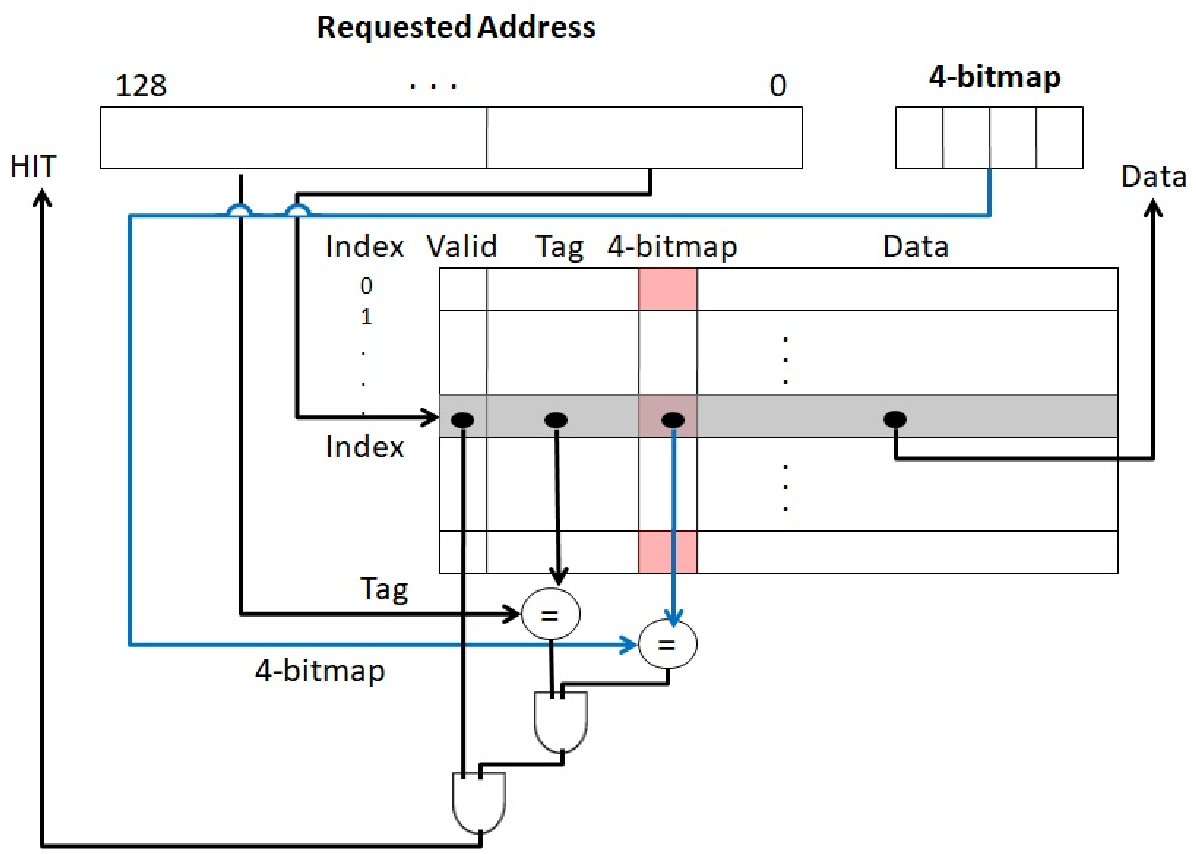

Figure 3.3: Illustration of Critical Data Tracker

Each cache block has an extension *4*-bitmap. A tag of each cache block contains the *4*-bitmap as a field so that it can distinguish which data regions are requested. When the system is initialized, all data of cache blocks are empty (valid bits are set to 0). When data is loaded into a particular block the corresponding valid bit is set to 1. If a memory request tries to fetch the data in a cache block it looks up the corresponding index in the cache array then compares the tag to make sure the appropriate block is found. When the status of a valid bit in the found cache block is set at 1, the data can be read successfully (cache hit). The critical data tracker takes one more step to make sure the finding data is in the cache block before checking validity of a cache block. It is depicted with blue lines in Figure 3.3. After the tag matching process, the *4*-bitmap field needs to be checked to see whether the requested data is in the cache block. The incoming *4*-bitmap information from the memory coalescing unit is compared with the *4*-bitmap in the found cache block. If the two have the same value, and the valid bit in the block is set at 1 the memory request is verified as a cache hit. If one of those multiple steps(indexing, tag matching, *4*-bitmap comparison, valid bit checking) fails, the memory request is a cache miss.

3.3 Newer Definition of Cache Miss

	Type	Description
Miss	First Miss (FM)	misses that fail indexing or tag matching in cache array.(normal miss)
	Subsequent Miss (SM)	following misses on the already requested block by prior misses demanding insufficient data.
Hit	Hit but Invalid (HI)	cache block found but containing insufficient data.

Table 3.1: Newer Definition of Cache Miss

While from the baseline architecture subsequent requests on the same cache block do not need to be taken into account, our proposed scheme requires sending multiple requests even on the already requested cache block. For instance, if a request needs partial data of a cache block, subsequent requests on the same cache block may need another partial data. As a result, we newly define cache miss as Table 3.1 plots.

For a First Miss(FM) case, they behave in the exactly same way as the baseline cache misses. However, subsequent requests possibly occur on the same cache block when the foremost cache miss requires insufficient data while the corresponding reply of the first is coming. Following memory requests on the same address while the first is being completed are defined as Subsequent Miss (SM).

We can also define another type of miss stemming from a cache hit. A cache block is successfully found, but there is a possibility the block has not enough data since we only filled a fraction of the block ahead of time. A memory request hitting the cache block realizes the necessary data is not in the block. Therefore, the request also needs to be sent to the lower memory as the original miss despite the existence of the cache. This type of miss is defined as Hit but Invalid (HI). Although HI cases can also have subsequent requests on the already accessed cache block, it is not taken into account since subsequent requests on HI can create significant system performance degradation.

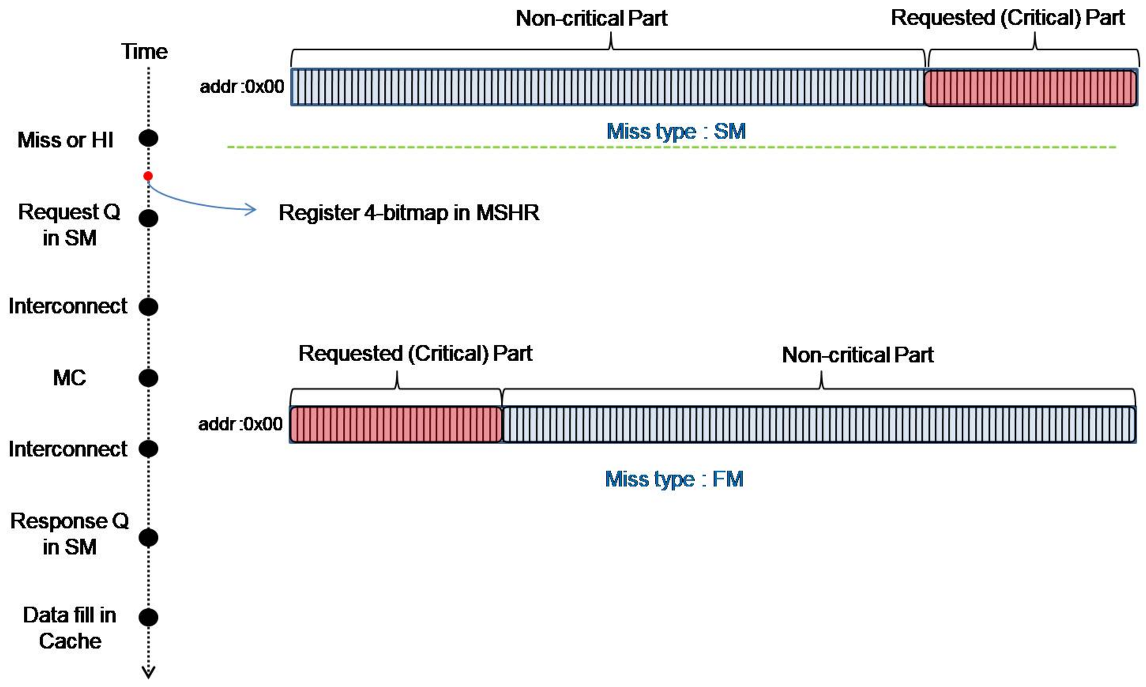

Figure 3.4: Example of Newer Definition of Cache Miss.

Figure 3.4 shows an example of a newer definition of cache miss. The first memory request (FM) initially looks up the cache array and it is turned out as a cache miss. The demanded data by that FM is the first 32 bytes (1 sub-block). The FM request registers an entry in the corresponding MSHR. Prior to filling the data from the first request, if another memory request on the same address with the first (SM) needs a different region of the FM it also needs to be sent through the interconnect to bring the demanded data since the first requested only the first 32 bytes while the second requires the last 32 bytes. Therefore, it is required to have a special treatment to support multiple requests on the already requested address.

3.3.1 Cache-Miss Case

To support cache miss case, we add a *4*-bitmap field to each MSHR as cache blocks. When an FM request is stored in an MSHR, its *4*-bitmap is stored in the allocated entry. When a subsequent memory request accesses a data region not set by the *4*-bitmap in the entry, the request is sent to an MC unlike the baseline where subsequent requests are always blocked. The bit value related to the new accessing regions in the *4*-bitmap are set to ones.

We make sure that two possible SM cases are properly handled by the data filtering mechanism, these occur according to the temporal locality between multiple requests. First, a subsequent request can arrive in an MC before the reply packet of the prior request is sent to the interconnect. The request is merged into the first request in the data-filtering table by ORing its *4*-bitmap to the *4*-bitmap created by the first request. Second, a sub-request may arrive in an MC, whereas the reply packet of the first request is already delivered through the interconnect. Since the entry created by the first request is already de-allocated in the table, our data filtering mechanism naturally considers the request as if it is the first request. When a reply packet arrives in an SM, the corresponding bit value of the *4*-bitmap field in the MSHR entry is flipped. All 0s of the *4*-bitmap implies there is no pending requests. Finally, a filling to a cache block is made. If there are pending requests by SMs, the filled data in the cache block is not valid until all SMs being processed fill the cache block. Otherwise, the cache block turns to the valid status to serve.

3.3.2 Cache-Hit Case

Although a memory request may hit a cache block, the block may not actually contain the necessary data incurred by a warp. Our mechanism detects this case when the *4*-bitmap is not a subset of the *4*-bitmap in the accessed tag. Then, it is necessary to re-access the cache block from the L2 cache. We consider this as a cache miss case by registering it to the MSHR and sending a request to an MC. Its reply is handled as discussed in Cache-Miss Case.

3.4 Packet Data Filter

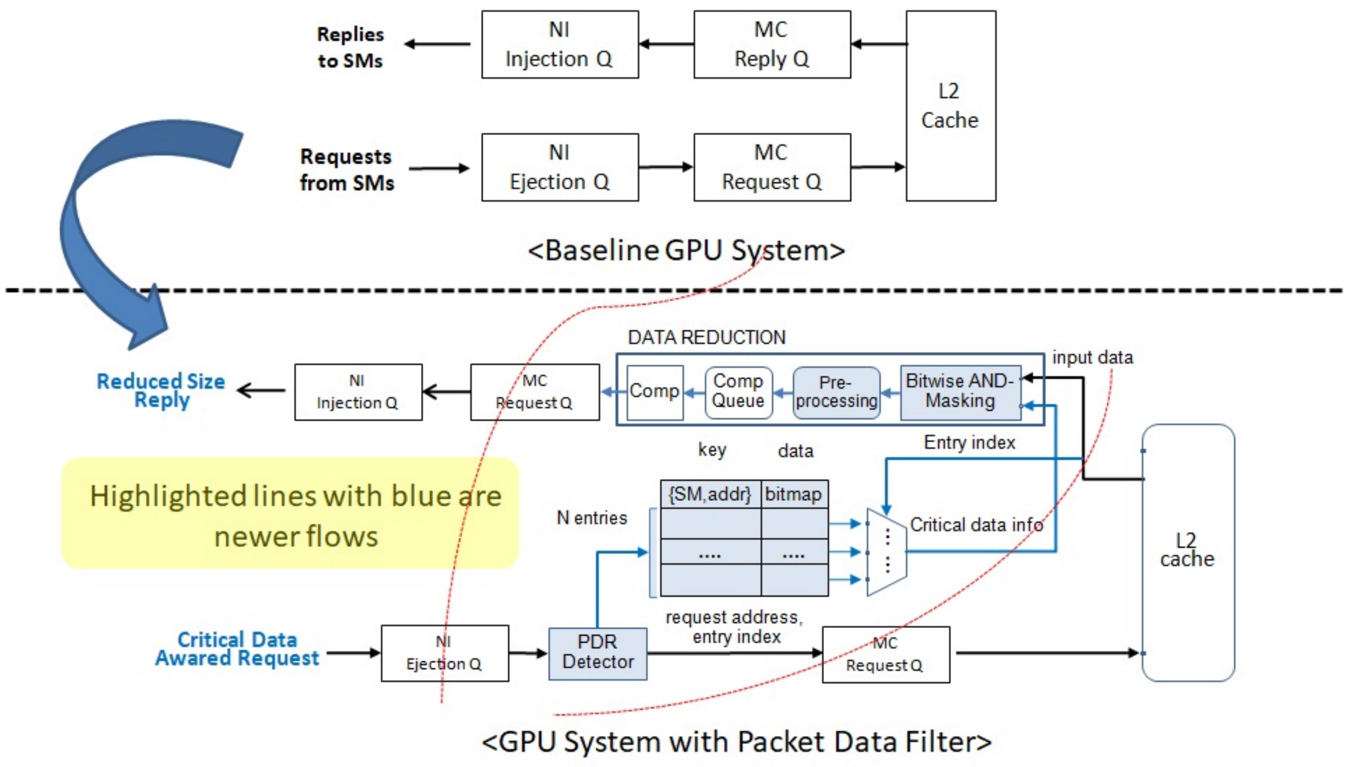

Figure 3.5: Illustration of Packet Data Filter

Our filtering mechanism is integrated between an MC and an interconnect as illustrated in Figure 3.5. When a global read request is sent to the L2 cache, the PDR detector determines whether the request needs full bytes or partial bytes. If the *4*-bitmap in the request is set to all ones, the detector classifies it as an FDR, and otherwise a PDR. A PDR is registered in the data-filtering table where an SM id and accessing address are used as a key and the *4*-bitmap information in the request is stored as data. When a reply data comes from the L2 cache with its corresponding index, the data filtering mechanism takes the *4*-bitmap from the table entry pointed by the index. Finally, the mechanism reduces the reply packet size by compressing only the critical data region.

3.4.1 Data Reduction

3.4.1.1 Packet Compression

The requested critical portion of each request can be identified by the *4*-bitmap of the table after being processed from L2 Cache. In order to effectively reduce the reply packet size it is essential to employ a compressor before replies are sent back through the interconnect. We first studied several compression algorithms. Subsequently, we studied the simple and appropriate compression scheme for packet compression, DPC. Then DPC is applied to the Data Filtering mechanism. Data compression is an instrumental approach for improving effective network bandwidth by reducing the size of packets (i.e. payload) before they are sent through an interconnect. Although a number of compression schemes have been introduced, they entail limitations on their applicability to the packet compression. The dictionary-based compression schemes encode data words into the corresponding short codes in a dictionary [13, 14, 15]. By compacting frequently appearing data words, they obtain high compressibility but are not suitable for packet compression due to deficient scalability, complex dictionary synchronization issues between compressors and decompressors, and high latency overhead. In packet compression, all *N* nodes in a network compress packets and each receiver node decompresses the packets coming from other *N-1* nodes. To neatly restore them, a decompressor in each receiver node needs to maintain *N-1* dictionaries. Synchronizing dictionaries between a compressor and a decompressor also requires an expensive hardware cost. The bulk data transfer for a dictionary created at the training phase is necessary from a compressor to a decompressor [13, 14] or synchronization protocols for a dictionary updated during runtime should be introduced [15, 16]. The hardware dependency on a dictionary in compressor and decompressor inherently creates a serialized process for each input. In particular, a compressor with long latency overhead adversely becomes a bottleneck for the next waiting packets in a highly dense network.

On the other hand, the pattern-based compression schemes encode data purely based on the occurrences of predefined patterns on the data words. Due to this simplicity, they are inherently suitable for packet compression by reducing latency overhead [17, 18]. Thus, we take pattern-based algorithm, Dual Pattern Compression, for our packet compression. We make the compressed data after the data filtering transparent to SMs by compressing and decompressing.

3.4.1.2 Dual Pattern Compression Algorithm

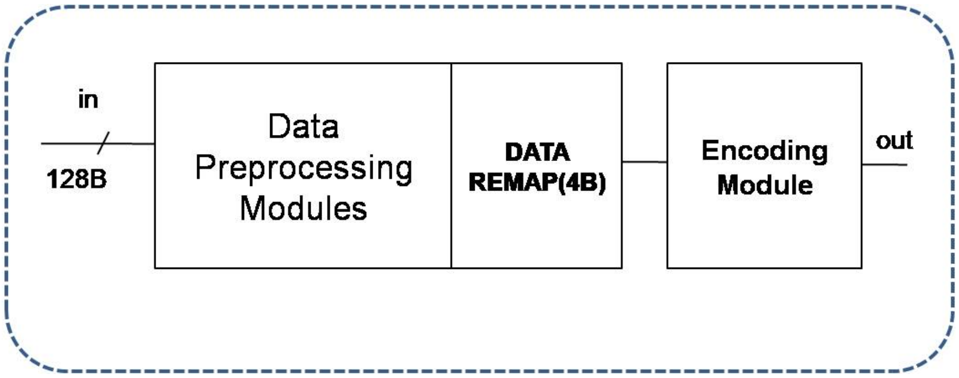

Figure 3.6: Illustration of DPC Pipeline.

DPC exploits data redundancy in a cache block size input. Figure 3.6 shows the DPC pipeline. The lightweight but powerful algorithm accepts by default a cache block (128 Byte size) as inputs. An input is decomposed into 32 segments (4 Byte granularity) and the 32 segments are compressed into a smaller format after preprocessing to reduce possible redundancy in the compression course. The remap function in the DPC reorganizes previously decomposed 32 segments as a list of group *i* that takes a sequence of a bit value at the bit position *i* of all the elements. Then, it creates two compressible patterns, consecutive all zeros or ones. The compressible patterns signifies that if all elements of the 32 segments at the same bit position are either zero or one they are compressible from 4 bytes (32 bits) to 1 bit. Otherwise, the data is situated as the original in the compressed format. Figure 3.7 plots an example of the data remap function and encoding with 32 byte input.

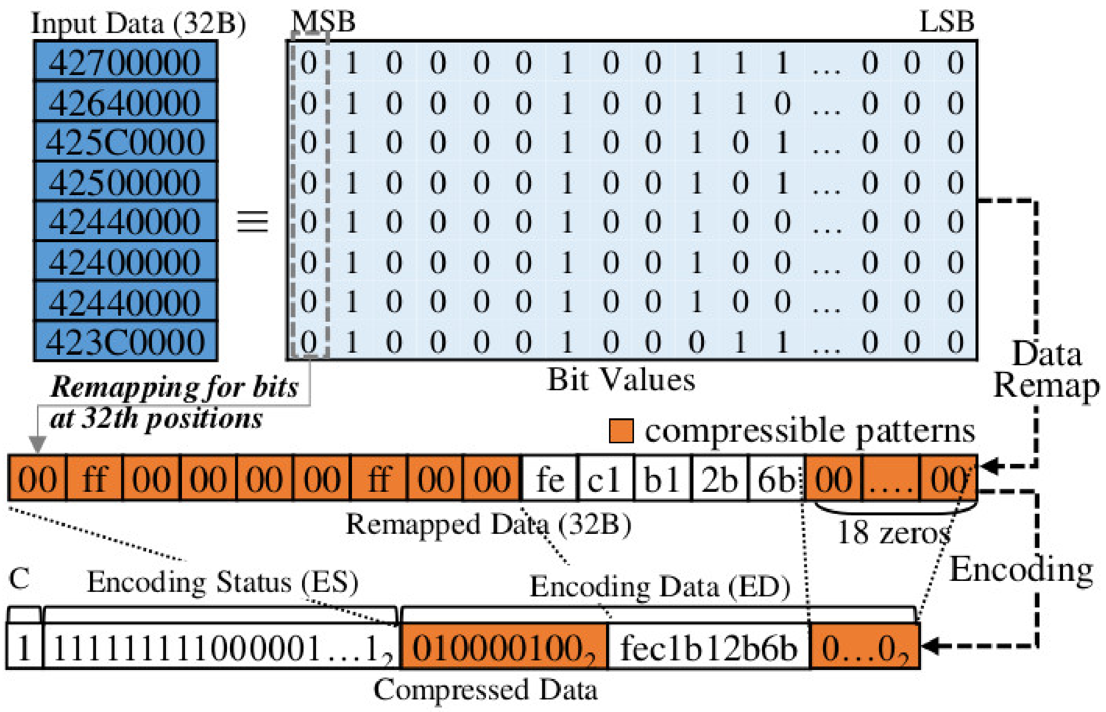

Figure 3.7: Illustration of DPC Remapping and Encoding.

Once a cache block is preprocessed by the remap function, the DPC compressor encodes the remapped data into a smaller form represented as a compression flag (*C*), a sequence of encoding status (*ES*), and a sequence of encoded data (*ED*). The compression flag is for recognizing if the output is encoded (1) or not (0). If the compression bit of input data is set at 0, it restores the remaining data without the bit as an output. Otherwise, it begins to decompress into individual segments by referring to each encoding status bit. If an encoding status bit is one, a segment is recovered as 4 bytes, consecutive zeros, or ones depending on its encoded data. Otherwise, the segments are restored as its original data. All the segments are rebuilt in the same manner.

3.4.1.3 *Applying DPC to Data Filtering Mechanism*

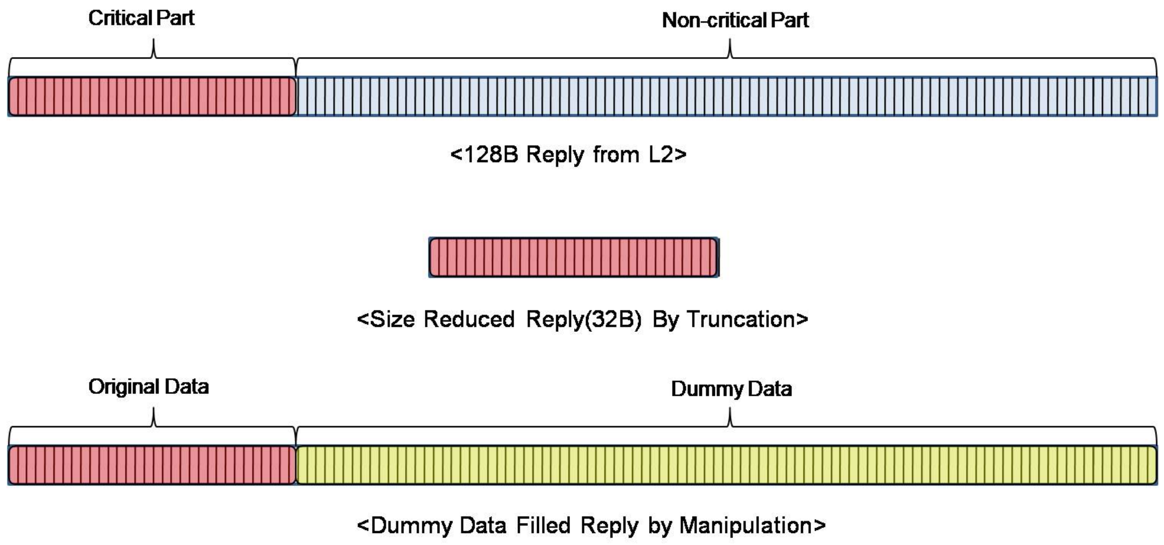
Critical Part
Non-critical Part
<128B Reply from L2>
<Size Reduced Reply(32B) By Truncation>
Original Data
Dummy Data
<Dummy Data Filled Reply by Manipulation>

Figure 3.8: Illustration of DF-TRUC and DF-MAN

The non-critical part of each reply from the L2 cache can be detected by the data filtering mechanism. The table situated between MCs and an interconnect retains the information in which the SM id, memory address, critical sub-block information can be identified by an entry index.

As illustrated in Figure 3.5, we employ DPC in the data filtering mechanism. A reply from the L2 cache is to be an input for DPC. The criticality-awared reply needs to be manipulated by either removing or substituting the non-critical part of the reply. While removing the unnecessary fraction of a reply packet is simple, subsequently producing a smaller sized packet, manipulating the unwanted portion favorable to The DPC entails additional steps. First is to directly remove all unnecessary data where the corresponding packet size naturally decreases for PDRs. For example, if an entry of the table contains *4*-bitmap as "1000" it means the first 32 bytes of the request are the critical data and eliminate the remaining 96 byte unnecessary part. In turn, the reply packet is transformed into a reduced size form (32 bytes) compared to the primitive 128 byte size form. This method is defined as *Data Filtering with Truncation* (DF-TRUC). The second option is to manipulate the non-critical portion by putting dummy data to increase compressibility for the DPC. From the discussed example for DF-TRUNC, the last 96 bytes are alternatively filled with dummy data. It is defined as *Data Filtering with Manipulation* (DF-MAN). Figure 3.8 shows both DF-TRUC and DF-MAN. For simplicity, DF-MAN sets all zeros in the unnecessary parts and critical regions remain as original where the basic 128 size form is kept.

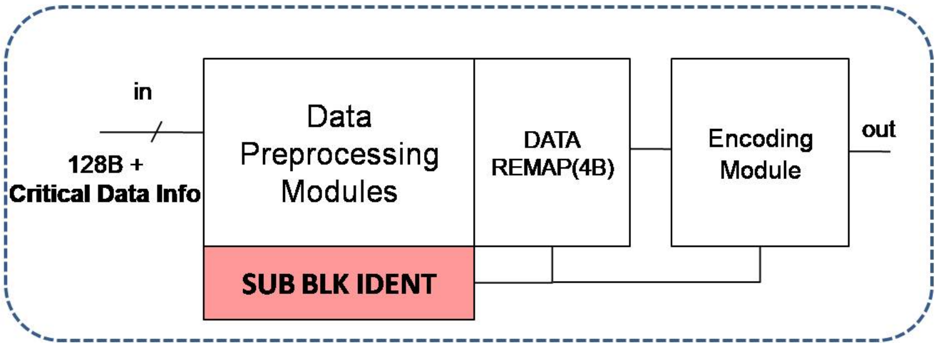

Figure 3.9: Illustration of DPC Extension

3.4.1.3.1 Data Filtering With Truncation The DF-TRUC is implemented by simply truncating a piece of a reply data. To make the truncation effective, DPC needs to take critical information from the data filtering table in addition to the reply data, then the remap function and the encoding module consider only a critical part of the data as illustrated in Figure 3.9. The sub-block identifier extended in DPC provides both the data remap function and the encoding module with critical the data information. If a request is required to fetch a fractional part, the unnecessary part is ignored creating an impact of the truncation. When decompressing a reply in an SM, the DPC decompressor restores data by referring to the critical data information held in the reply. To support the case that multiple requests merged in the data filtering mechanism, the critical information of an outgoing reply from the data reduction module needs to be updated as the corresponding entry information in the table. For instance, if a memory request has "1000" for the critical data and this request is still being processed by an MC, another request in the same address has "0100" and it is just ejected from the interconnect to the MC (the same MC with the first request). These two requests are merged by the data filtering mechanism and the table has "1100" in the corresponding entry. The merged reply should contain "1100" for the *4*-bitmap to successfully restore data from the DPC decompressor.

3.4.1.3.2 Data Filtering With Manipulation. To enhance compressibility for the DPC, the unnecessary part is filled with all 0s considering the DPC takes advantage of bit level redundancy. However, putting all 0s in the unnecessary part cannot alway be beneficial to all reply data because the DPC exploits a dual pattern (0 or 1 value) in the bit-plain (a sequence of bit-value at the vertically same bit position). As a result, the DF-MAN compares the original input data and the manipulated data. The copy that exhibits more bit level redundancy is selected as the input for DPC. Therefore, this technique can use DPC as the original.

3.5 Cache Fill

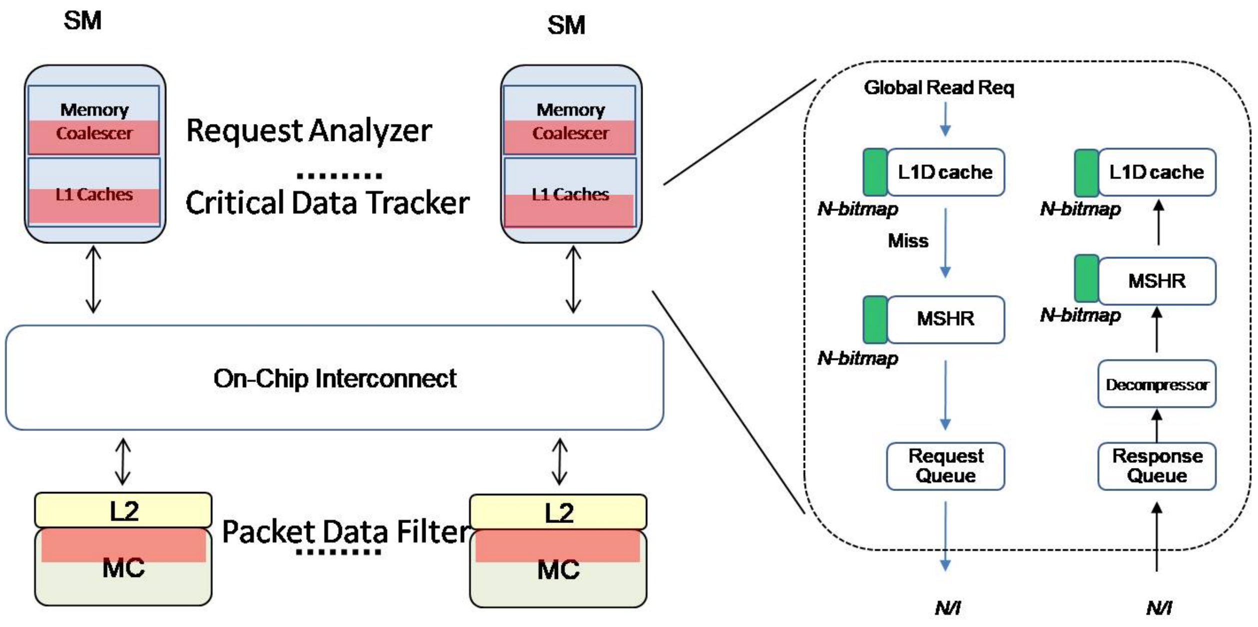

Figure 3.10: Illustration of Data Flow

A Request Analyzer and a Critical Data Tracker are employed in the SMs and a Packet Data Filter is incorporated between an interconnect and an MC to support filtering critical data. The data flow of the request track throughout the data filtering is depicted with blue lines in Figure 3.10. Now, reduced size packets are de-packetized into a smaller number of flits to be injected into the interconnect. Reply packets successfully arrive at the MSHR in an SM traveling over the interconnect and a DPC decompressor. The corresponding bits of a *4*-bitmap of an MSHR is set as 0s by the *4*-bitmap of a reply. For example, if the *4*-bitmap of a reply is "1000" and the *4*-bitmap of an MSHR is "1111", the bitmap in the MSHR turns into "0111." Then the restored data of a reply fills the cache block. However, cache blocks can be valid only when all SMs bring their data. The validation is made by the *4*-bitmap in an MSHR. if a *4*-bitmap is all set at 0s, it tells that the cache block can finally be valid. Therefore, when an entry in an MSHR is deallocated the corresponding cache block becomes valid.

4. REQUEST PREDICTION

We have studied the data filtering mechanism with two options for data reduction. We are aware that both MSHRs and cache blocks have extensions storing critical data information of requests. From the working combination of these, we expect to decrease the traffic volume of replies to be delivered through an interconnect where the overall network bandwidth burden is alleviated. However, naively exploiting the scheme can create a serious problem. Figure 4.1 describes one of the worst scenarios when the data filtering scheme is simply applied.

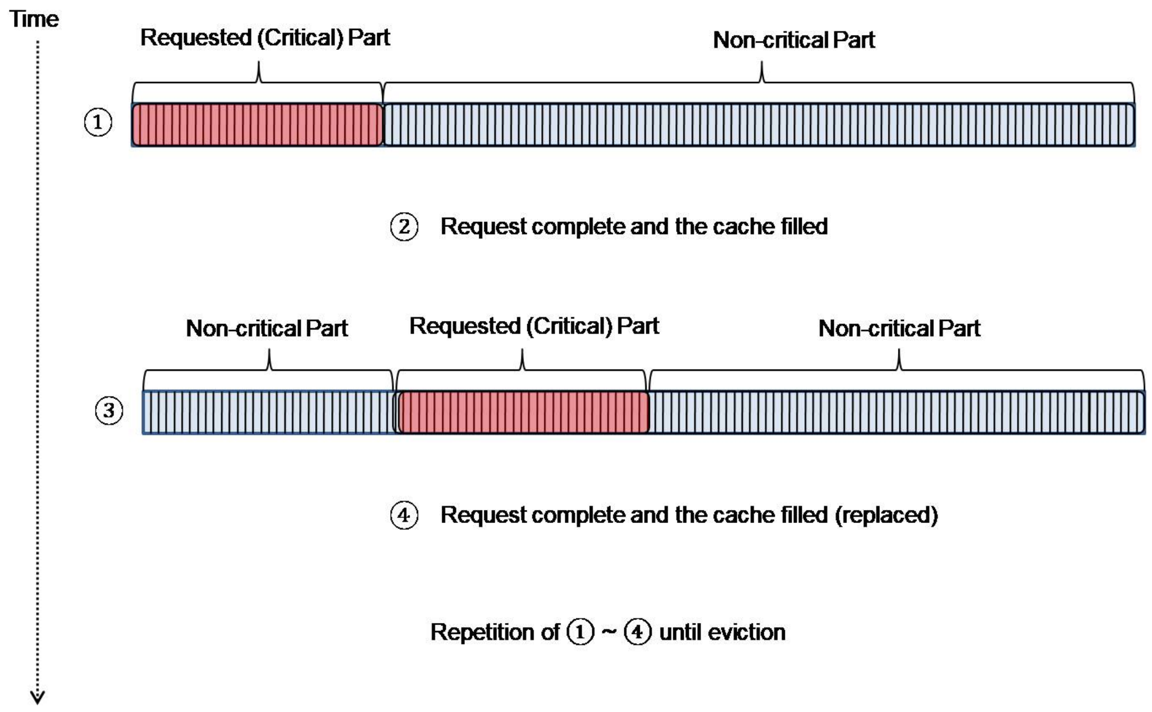
Time
Requested (Critical) Part
Non-critical Part
①
② Request complete and the cache filled
Non-critical Part
Requested (Critical) Part
Non-critical Part
③
④ Request complete and the cache filled (replaced)
Repetition of ① ~ ④ until eviction

Figure 4.1: A Worst Case Example

A warp requests a memory read and it is determined that 32 byte data is needed from the request analyzer (the extended memory coalescing unit). Neither an existing entry in the cache array nor appropriate data in a found cache block makes the read request a cache miss. The cache miss requires getting the data from the L2 cache or DRAM by traversing through the data filtering mechanism. The successfully restored data fills the corresponding block with the critical part. A subsequent request from another warp to the same cache block hits in which necessary data is not resided. As a result, the request causes extra stalls to obtain the data from the lower memory. The reply of the sub-request replaces the previous data with the newer critical one on the block. If another request demands another sub-block or the sub-block requested by the first request rather than the one filled by the second request, these multiple requests on the same cache block significantly harm performance by going back and forth to the interconnect multiple times. Thus, our mechanism requires a gear to smartly manage multiple requests on the same cache block by feasibly controlling the number of requests.

One might think of appending the data brought by the SMs through OR operation with the existing data. Although it could be a possible solution to deal with the shown example above, it has a drawback as well as Figure 4.2 plots.

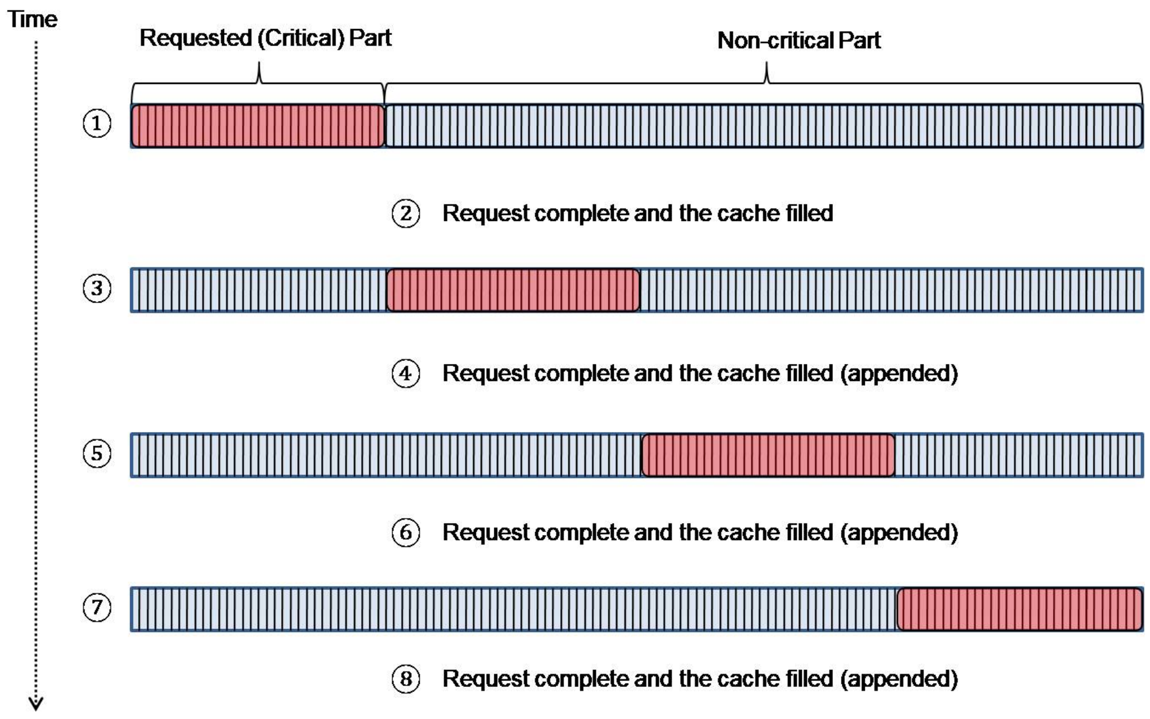
Time
Requested (Critical) Part
Non-critical Part
①
② Request complete and the cache filled
③
④ Request complete and the cache filled (appended)
⑤
⑥ Request complete and the cache filled (appended)
⑦
⑧ Request complete and the cache filled (appended)

Figure 4.2: An Example of Drawback For Appending Cache

Figure 4.2 shows 4 times increase of the additional overheads in case we apply an appending technique for cache blocks as opposed to the baseline that sends a request only one time. In this example it is more efficient to request the entire data than a partial critical portion if multiple requests to a cache block are ICA patterns. In addition, a higher traffic volume of requests from the SMs to the MCs leads to system performance degradation due to the corresponding larger replies. Therefore, it is essential to smartly control the number of requests to alleviate the burden of the request path.

4.1 Request Controller and Windows Manager

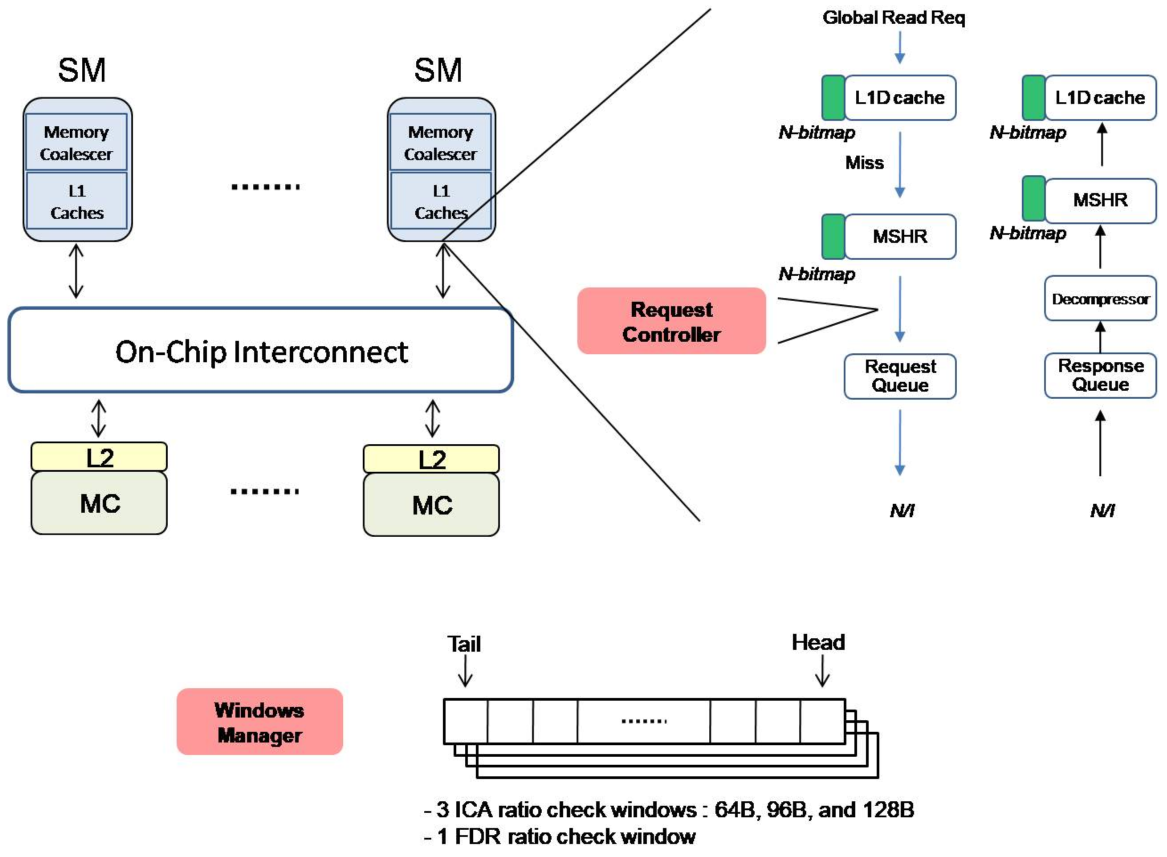

Figure 4.3: Architectural Overview of RC included Data Filtering

Given the observations, we propose a *Request Controller*(RC) as well as a *Windows Manager* (WM) as illustrated in Figure 4.3. The goal of this prediction mechanism is not to improve system performance but to prevent performance degradation of workloads showing ICA patterns. Each SM has an RC and a WM to determine whether a memory request needs to bring the entire block or partial critical data from an MC. The more RC requires the FDRs, it becomes the similar to the DPC because the data filtering mechanism cannot be activated with the FDRs. For the RC's decision, the WM maintains 3 ICA ratio check windows (64B, 96B, and 128B size) and 1 FDR ratio check window. Each window retains the most recent information of memory accesses according to memory access size in an FIFO fashion. The recent information preserves the number of CA and ICA for the ICA ratio check windows and the number of PDRs and FDRs for the FDR check window within an assigned window size.

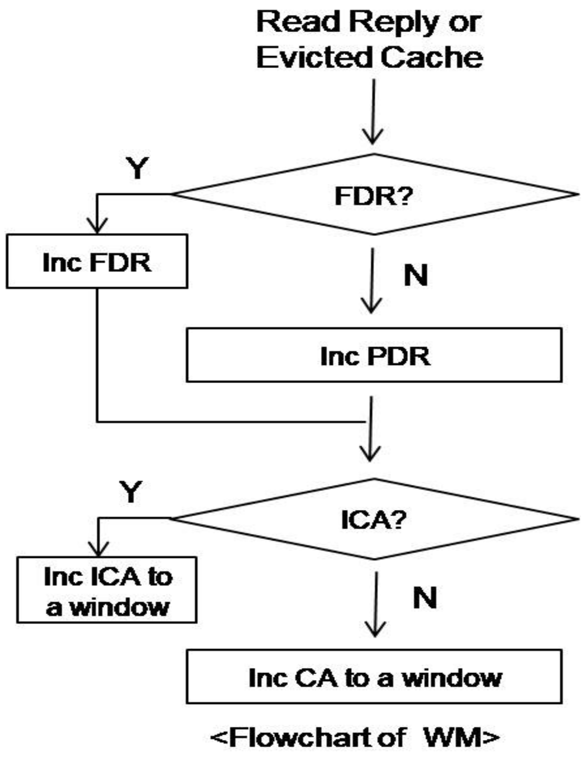

Figure 4.4: Flowchart of WM

Each global read reply from an interconnect and L1D evicted cache block are inspected by the WM in an SM to see whether they are CA or ICA and FDR or PDR as Figure 4.4 elaborates. To support the auditing of consistency, cache blocks and MSHRs have an extra bit containing consistent status of multiple memory accesses. If multiple requests on the same memory address are inconsistently accessed, the consistent status bit, either in a cache block or an MSHR is turned on. Inconsistent multiple memory requests before the foremost request fills the cache block flip the consistent status bit in an MSHR. If the consistent status bit in an MSHR is set on when a reply comes to the MSHR, a bit pointing inconsistent status is pushed back to the corresponding size consistent check window. Otherwise the other bit is pushed back to the window.

To check if a reply or an evicted block is FDR or PDR, the *4*-bitmap in an MSHR or a cache block is exploited. If the bitmap is set to all 1s, it is considered FDR, otherwise PDR. In the case of multiple requests sent, only the last request is considered to push back a value in a FDR ratio check window.

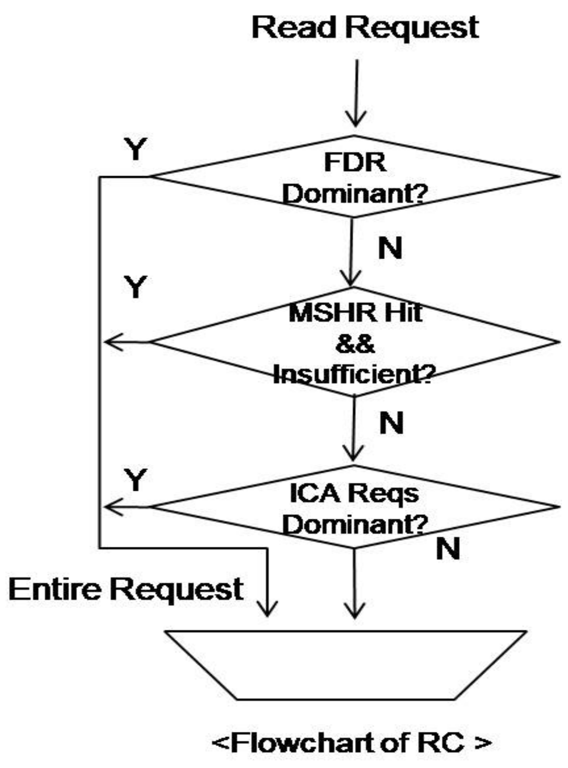

Figure 4.5: Flowchart of RC

Figure 4.5 plots the flow of RC. RC takes an L1D cache read miss as an input and then determines whether to demand the entire data or partial critical data. Since bringing only a critical fractional part, it is risky to create another memory request, even on the already filled cache. RC conservatively and pessimistically makes a decision.

First, the RC checks the FDR ratio check window. If the ratio of the entire data requests exceeds a threshold in a given time, the RC decides to bring all data regardless of its criticality. Second, supposing the ratio of the entire data requests is not over a threshold, if the MSHR is hit and the prior memory request did not call for sufficient data for the current request, we also demand the entire data because it signifies that the request patterns are inconsistent. Finally, also assuming a request does not hit the MSHR, if the ratio of ICA requests in the corresponding size window does not surpass a threshold it is the appropriate time to demand the partial memory request.

5. EVALUATION

5.1 Evaluation Methodology

We implement our data filtering mechanism with a cycle-accurate GPU simulator, GPGPU-Sim 3.2.2 [8]. We modify the GPGPU-sim to model a GPUs architecture supporting this scheme. The detailed configurations are summarized in Table 5.1

System Parameters	56 SMs, 8 MCs
Shader Core L1 Cache L2 Cache	1.4Ghz, Greedy-then-oldest (GTO) Scheduler L1I(2KB), L1D(16KB), L1T(12KB), L1C(8KB) 128KB per MC
Interconnect	8 x 8 Mesh, 1.4Ghz, 2- Cycle Router, Wormhole, Credit-based Flow-control
ICA threshold FDR threshold Windows size	25%, 50%(default), 75% 30%, 60%, 90%(default) 8, 16(default), 32
Table size	32, 256(default), 1024

Table 5.1: System Configuration Parameters

It is configurable to dynamically partition between the L1 cache and shared memory which means the L1 cache can take 16KB, whereas shared memory can take 48KB, and vice versa. We take 16 KB L1 cache size for this evaluation. For the NoC, we use a 2D mesh topology due to its scalability, simplicity, and regularity [19, 20, 21]. To prevent a protocol deadlock, we built a single network with two separate virtual channels (VCs) for the request network from SMs to MCs and the reply network from MCs to SMs. We use heterogeneous 33 benchmarks from Rodinia [9], Graph, Mars [10] benchmark suites. Table 5.2 summarizes their characteristics. We evaluated them either up to one billion instructions implemented or finished before one billion instructions was reached.

We set threshold values for the Windows Manager of the Request Prediction technique and set the size of the table for PDRs. By default, the window size is set at 16 (heuristic approach), each

Benchmark	Acronym	Benchmark	Acronym	Benchmark	Acronym
Backprop	R.BP	BFS	R.BFS	CFD	R.CFD
DWT2D	R.DWT2D	Gaussian	R.GSN	Heartwall	R.HW
Hotspot	R.HS	Hybridsort	R.HYS	Kmeans	R.KM
LavaMD	R.LMD	LUD	R.LUD	Mummergpu	R.MUM
Myocyte	R.MYC	NN	R.NN	NW	R.NW
ParticleFileter	R.PARTF	PathFinder	R.PATHF	Srad	R.SRAD
StreamCluster	R.SC				
APSP	G.APSP	CCL	G.CCL	GCO	G.GCO
GCU	G.GCU				
II	M.II	KM	M.KM	MM	M.MM
PVC	M.PVC	PVR	M.PVR	SM	M.SM
SS	M.SS	WC	M.WC		

Table 5.2: Benchmarks

of the 4 windows has 50% as a threshold and the turning point of requiring entire data regardless of criticality is 90 %. Namely, if the ICAs ratio exceed 50 percent or the FDR ratio is more than 90 percent within the most recent 16 requests of the corresponding window, it demands the entire data rather than a partial critical one. Otherwise, it fetches only the critical part data. For the table, we set 256 entries for PDRs.

5.2 Effect on Performance

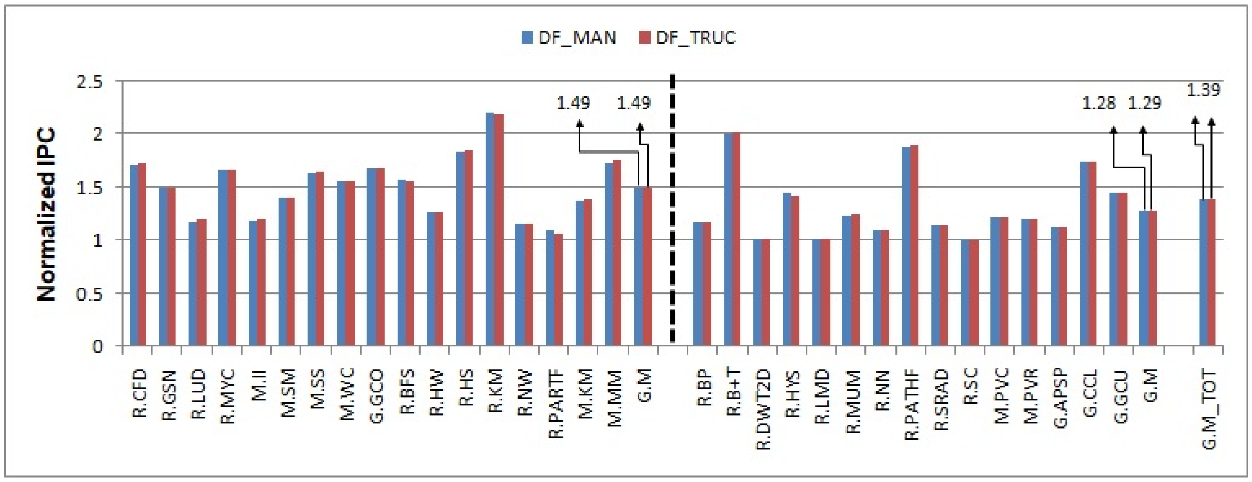

Figure 5.1: Normalized IPC Over Baseline

Figure 5.1 shows the normalized IPC across all benchmarks over the baseline. The blue bar indicates DF-MAN and the red bar presents DF-TRUC. We make two major observations on system performance in this analysis.

First, the data filtering mechanism generally works well without consideration of data reduction methods.On average, the proposed mechanism provides 39% IPC improvement across all benchmarks. The workloads of the left group from the dotted line shows a better performance result than the right because they mostly have partial CA patterns. Although the right group is not does not have sufficient partial CA patterns, performance is not degraded showing at least the same result with the baseline.

Second, one method for data reduction does not outperform the other. In general, the DF-TRUC shows a slightly better performance such as R.CFD, R.LUD, M.II, M.SS, and R.HS, but the overall performance difference is subtle.

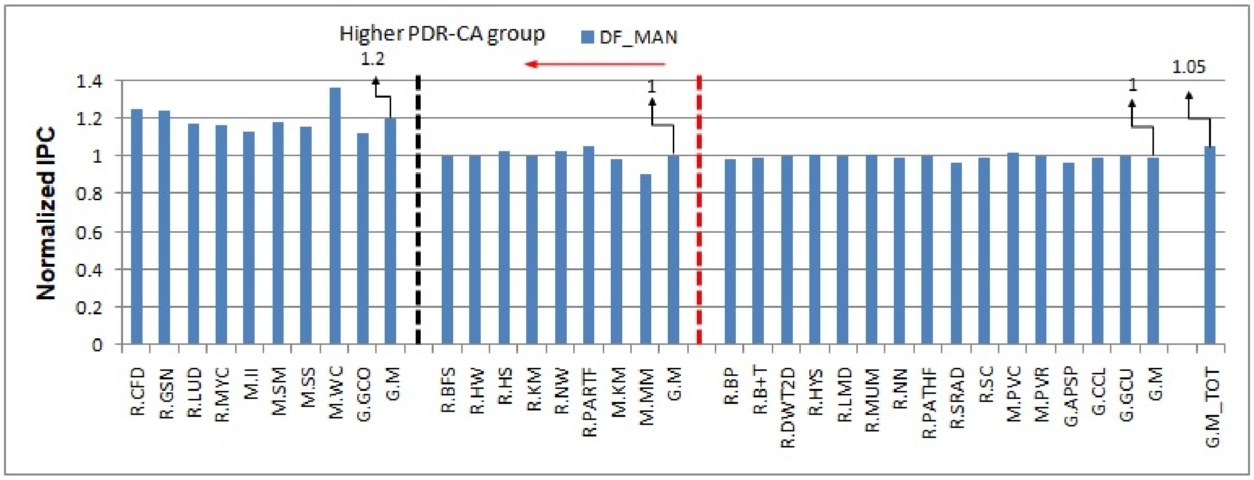

Figure 5.2: Normalized IPC Over DPC

Figure 5.2 denotes the normalized IPC over DPC. DF-MAN is compared with DPC. Normally, the data filtering improves IPC 5% across all benchmarks. Since DF-MAN and DF-TRUC give almost the same performance whichever scheme we choose, they show a similar result. We make two conclusions from this result.

First, although the data filtering mechanism can achieve better IPC improvement over DPC for workloads in favor of partial CA patterns, some benchmarks favorable to partial CA patterns don't increase IPC. The leftward ones from the red dotted line are the workloads preferable to partial CA patterns. They had a promising potential to improve performance showing mostly partial CA patterns, but only those benchmarks after the black dotted line to the left can benefit from the scheme. Benchmarks towards the left after the black dotted line provide 20% IPC improvement. On the other hand, benchmarks between the red and black dotted line show on average the same performance result with DPC. The reason is discussed with the compression ratio in a following section.

Second, several applications such as M.KM, M.MM, R.BP, R.SRAD, and G.APSP exhibit performance degradation compared to DPC. The following sections explain why such workloads bring worse performance results by analyzing in terms of traffic ratio, space savings, and enhanced requests.

5.3 Network Bandwidth Analysis

5.3.1 Traffic Reduction on Reply

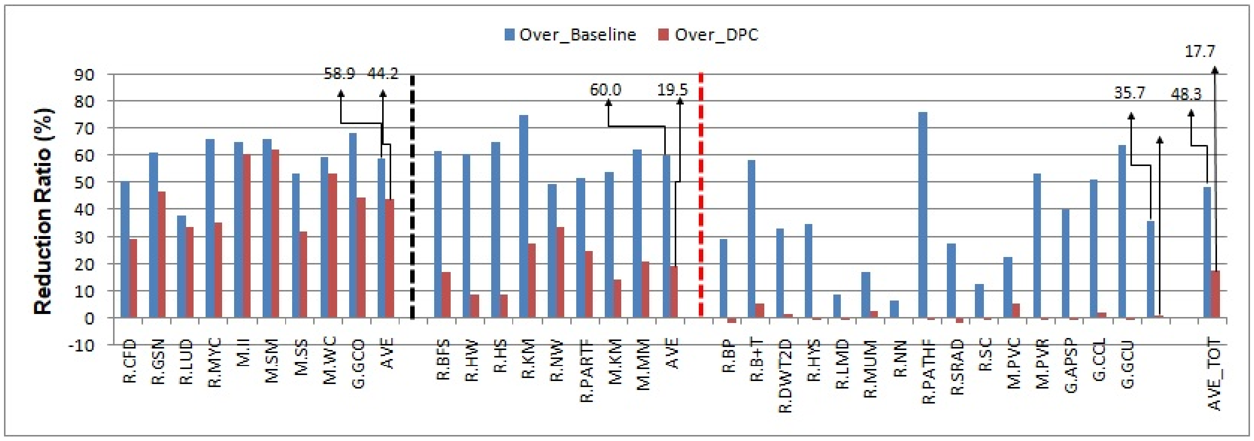

Figure 5.3: Reduced Traffic Ratio From MC to Interconnect

Figure 5.3 plots the reduced traffic volume on reply paths. The reduction ratio is measured with the number of flits returning to the corresponding SM at interconnect injection queues. On average, 48.3% is saved against the baseline and 17.7% is lowered over DPC. Interestingly, while applications in favor of partial CA patterns show higher traffic savings even over DPC, others hardly yield traffic reductions to DPC.

We observe that the group having better IPC than DPC achieves a larger traffic reduction ratio as compared to DPC (on average 44.2%), whereas the applications providing favorable memory access patterns but not performance improvement have only a 19.5% reduced traffic ratio over DPC.

5.3.2 Space Savings

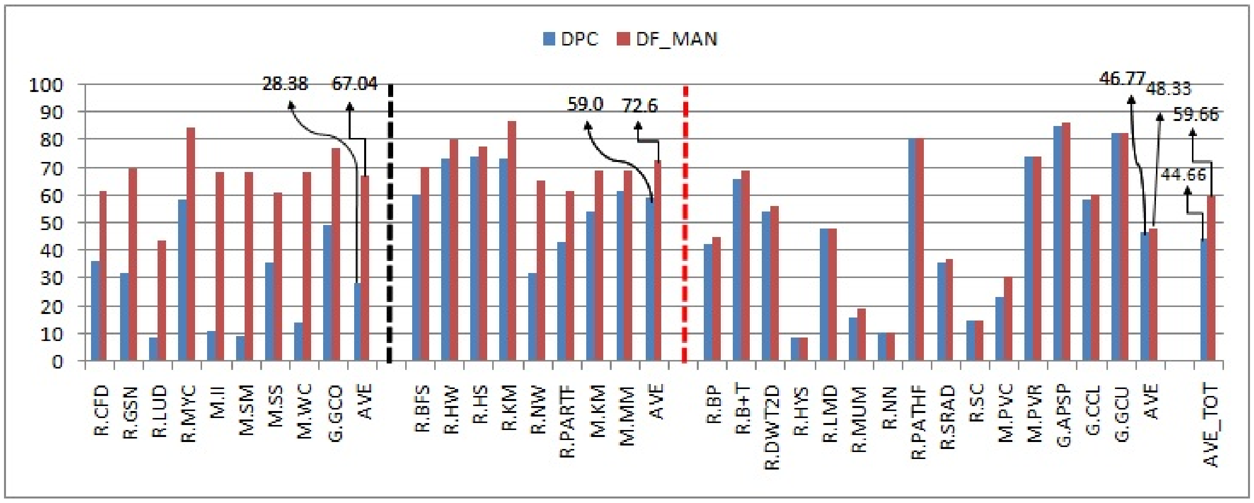

Figure 5.4: Compression Ratio

Figure 5.4 compares the space savings ratios between DPC and DF-MAN across all benchmarks. The first group of applications from the left shows the highest ratio difference between the two (67.04% for DF-MAN and 28.38% for DPC). This is the main reason why the scheme can gain better performance results over DPC. Since DF-MAN saves the network bandwidth more effectively than DPC for the first group, system performance increases for them. However, the middle group or the rightmost group, show only show a little savings ratio gap between the two scheme. It implies DPC itself achieves the performance benefit at most for those workloads.

5.4 Cache Miss Rate

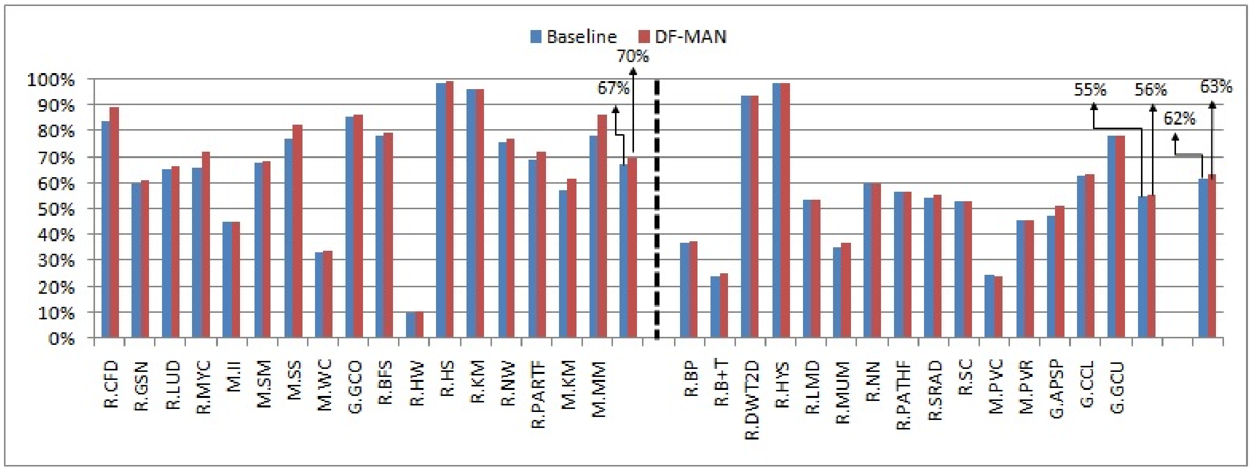

Figure 5.5: Miss Rate in L1D Cache

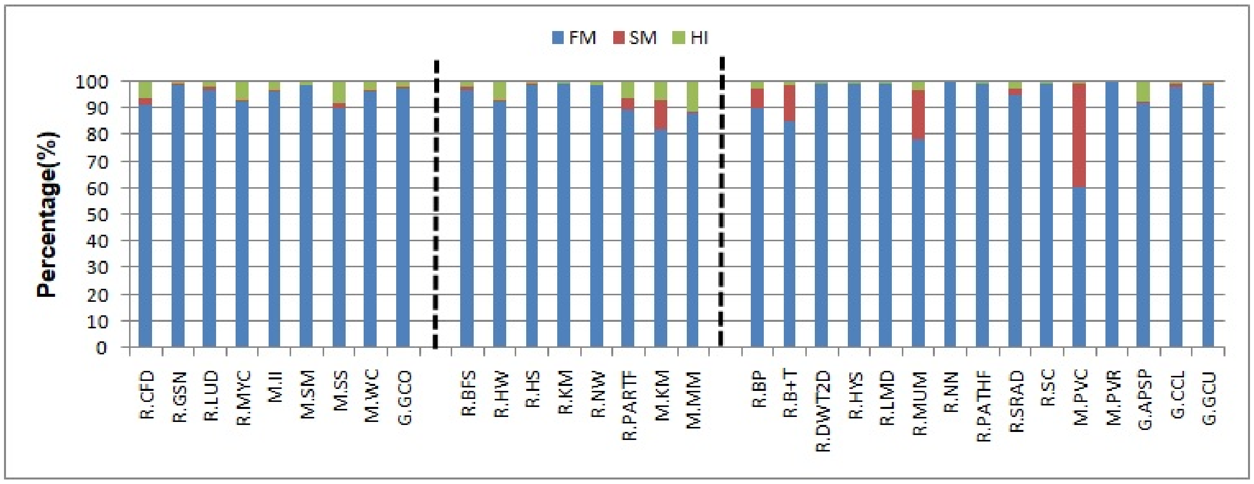

Figure 5.6: Miss Distribution According to The Newly Defined Misses

The L1D cache miss rate is shown in Figure 5.5. Also, the miss distribution of DF-MAN according to 3 different miss types is presented in Figure 5.6. DF-MAN improves system performance by reducing the traffic volume of replies at the cost of increased cache misses.

The group not favorable to the partial CA patterns has very few increased miss penalties. In Figure 5.5 we see only 1% difference compared to the baseline, and in Figure 5.6 we hardly find HI type misses. From the result, we conclude the request prediction mechanism satisfactorily understands memory access patterns at runtime. It prevents from fetching partial critical data if memory access patterns do not show a inclination to partial CAs.

HIs are considered as a traffic overhead creator. While SMs can be merged by the data filtering table at an MC, there is no possibility for multiple requests on the same cache block to merge when a cache block is hit. As a result, HIs brings about more requests overhead.

Although HIs are relatively higher in applications in favor of this scheme, they attain performance improvement. This is mainly because the increased number of requests is reimbursed by the decreased number of flits injected into the interconnect from MCs.

6. SUMMARY AND CONCLUSIONS

In this work, we initially explore the inefficient usage of the L1D cache in various applications. We study memory access patterns according to consistency and categorize them. Essential micro-architectural enhancements are proposed to support filtering critical data. The key idea is to design the mechanism to relieve the reply traffic while being aware of the critical information that the SMs exactly need rather than full byte cache data. At the same time, it does not augment traffic volume of the request side so that reduced traffic on the reply path efficiently improves the performance. Discussions include two data reduction methods and a request prediction mechanism. Our evaluation shows that the critical data filtering mechanism coupled with DPC achieves on average 39% IPC improvement and 23% network bandwidth saving on the reply path across heterogeneous and text processing workloads. Also, comparing our scheme with DPC 5% IPC improvement for overall benchmark suits and 20% increase for workloads favor our mechanism.

REFERENCES

[1] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, “Scaling the bandwidth wall: Challenges in and avenues for cmp scaling,” *SIGARCH Comput. Archit. News*, vol. 37, pp. 371–382, June 2009.

[2] W. A. Wulf and S. A. McKee, “Hitting the memory wall: Implications of the obvious,” *SIGARCH Comput. Archit. News*, vol. 23, pp. 20–24, Mar. 1995.

[3] NVIDIA Corporation, “CUDA C programming guide, 9.2 edition,” 2018. Available at `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[4] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, “GPUs and the future of parallel computing,” *IEEE Micro*, vol. 31, pp. 7–17, Sept 2011.

[5] D. Abts, N. D. Enright Jerger, J. Kim, D. Gibson, and M. H. Lipasti, “Achieving predictable performance through better memory controller placement in many-core cmps,” *SIGARCH Comput. Archit. News*, vol. 37, pp. 451–461, June 2009.

[6] A. Bakhoda, J. Kim, and T. M. Aamodt, “Throughput-effective on-chip networks for many-core accelerators,” in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 421–432, Dec 2010.

[7] B. Li, J. Sun, M. Annavaram, and N. S. Kim, “Elastic-cache: Gpu cache architecture for efficient fine- and coarse-grained cache-line management,” in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 82–91, May 2017.

[8] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, “Analyzing cuda workloads using a detailed gpu simulator,” in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 163–174, April 2009.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54, Oct 2009.

[10] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: A mapreduce framework on graphics processors," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, (New York, NY, USA), pp. 260–269, ACM, 2008.

[11] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," *SIGARCH Comput. Archit. News*, vol. 38, pp. 235–246, June 2010.

[12] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonia, "Managing dram latency divergence in irregular gpgpu applications," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 128–139, Nov 2014.

[13] A. Arelakis and P. Stenstrom, "Sc2: A statistical compression cache scheme," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 145–156, June 2014.

[14] A. Arelakis, F. Dahlgren, and P. Stenstrom, "Hycomp: A hybrid cache compression method for selection of data-type-specific compression methods," in *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, (New York, NY, USA), pp. 38–49, ACM, 2015.

[15] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas, "C-pack: A high-performance microprocessor cache compression algorithm," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, pp. 1196–1208, Aug 2010.

[16] Y. Jin, K. H. Yum, and E. J. Kim, "Adaptive data compression for high-performance low-power on-chip networks," in *2008 41st IEEE/ACM International Symposium on Microarchi-*

tecture, pp. 354–363, Nov 2008.

[17] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, (New York, NY, USA), pp. 377–388, ACM, 2012.

[18] A. R. Alameldeen and D. A. Wood, "Frequent pattern compression: A significance-based compression scheme for l2 caches," *Technical Report 1500, Computer Sciences Dept., UW-Madison*, April. 2004.

[19] K. H. Kim, R. Boyapati, J. Huang, Y. Jin, K. H. Yum, and E. J. Kim, "Packet coalescing exploiting data redundancy in gpgpu architectures," in *Proceedings of the International Conference on Supercomputing*, ICS '17, (New York, NY, USA), pp. 6:1–6:10, ACM, 2017.

[20] A. Bakhoda, J. Kim, and T. M. Aamodt, "Throughput-effective on-chip networks for many-core accelerators," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, (Washington, DC, USA), pp. 421–432, IEEE Computer Society, 2010.

[21] H. Jang, J. Kim, P. Gratz, K. H. Yum, and E. J. Kim, "Bandwidth-efficient on-chip interconnect designs for gpgpus," in *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, (New York, NY, USA), pp. 9:1–9:6, ACM, 2015.