MEMORY OPTIMIZATIONS FOR HIGH-THROUGHPUT COMPUTER SYSTEMS

Zhiyuan Lu
Michigan Technological University, zhlu@mtu.edu

MEMORY OPTIMIZATIONS FOR HIGH-THROUGHPUT COMPUTER SYSTEMS

By

Zhiyuan Lu

A DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

In Computer Science

MICHIGAN TECHNOLOGICAL UNIVERSITY

2023

This dissertation has been approved in partial fulfillment of the requirements for the Degree of DOCTOR OF PHILOSOPHY in Computer Science.

Department of Computer Science

Dissertation Advisor:	*Dr. Jianhui Yue*
Committee Member:	*Dr. Soner Onder*
Committee Member:	*Dr. Zhenlin Wang*
Committee Member:	*Dr. Qinghui Chen*
Department Chair:	*Dr. Zhenlin Wang*

Dedication

To my families, teachers, and friends

who didn't hesitate to encourage me, criticize my work, and give help at every stage - without which I would neither be who I am nor would this work be what it is today.

Contents

List of Figures . . . . **xi**

List of Tables . . . . **xv**

Preface . . . . **xvii**

Acknowledgments . . . . **xix**

List of Abbreviations . . . . **xxi**

Abstract . . . . **xxvii**

1 Introduction **1**

1.1 Memory Wall 1

1.2 System with Non-volatile Memory 2

1.3 Neural Network Accelerator 4

2 Background **7**

2.1 Logging in Non-volatile Memory 8

2.1.1 Crash Consistency in NVM systems 8

2.1.2 Conventional Log Organization 10

2.1.3 Asynchronous DRAM Refresh (ADR) 11

2.1.4 Logless Atomic Durability (LAD) 12

2.2 Neural Network and Neural Network Accelerator 13

2.2.1 Neural Networks 13

2.2.2 Computations in Neural Network 15

2.2.3 Computational Graph 19

2.2.4 Neural Network Accelerators 22

2.2.5 *im2col* methods: NCHW vs. NHWC 25

2.2.6 Dataflow in Neural Network Accelerators 28

2.2.7 Off-chip Memory Mapping in NNAs 33

3 Improving the Performance of NVM Crash Consistency under Multicore 35

3.1 Overview 35

3.2 Motivation 36

3.2.1 Persist requirements of a transaction 36

3.2.2 Unnecessary Logs 37

3.3 Two-Stage Transaction Execution (TSTE) 39

3.4 Virtualization of the ADR Buffer (VADR) 44

3.5 Redo Logging for ADR Depletion 47

3.6 Evaluation 48

3.6.1 Experiment Setup 48

3.6.2 Removal of Unnecessary Log Operations 52

3.6.3 Transaction Throughput Improvements 53

3.6.4 Log Operations Reduction 55

3.6.5 Sensitivity of ADR Buffer Capacity 56

3.7 Summary 58

4 Accelerate Hardware Logging for Efficient Crash Consistency in NVM 61

4.1 Overview 61

4.2 Motivation 62

4.3 Log Entry Allocation Scheme (LALEA) 63

4.4 Log Metadata Buffering Scheme (BLOM) 68

4.5 Evaluation 71

4.5.1 Experiment Setup 71

4.5.2 Transaction Throughput 72

4.5.3 Log Entry Persistence Latency 74

4.5.4 Transaction Commit Latency 75

4.5.5 Throughput under the Alternative Address Mapping Scheme 76

4.6 Summary 77

5 Improving Neural Network Accelerator Performance by Optimizing Memory Accesses 79

5.1 Overview 79

5.2 Motivation 80

5.2.1 Wide Layers 80

5.2.2 Performance Impact of Wide Layers. 84

5.3 Alleviate Repeated Access to IFMAP (ARAI) 86

5.4 Load-Aware Placement of Data Tiles (LAP) 88

5.4.1 Microarchitecture of LAP 93

5.5 Evaluation 108

5.5.1 Evaluation Setup 108

5.5.2 Improvement in System Performance 111

5.5.3 Reduction in DRAM Read Traffic 112

5.5.4 Increased Bandwidth of Off-chip Memory 113

5.5.5 Sensitivity Analysis 116

5.5.6 Discussion 117

5.6 Summary 117

6 Conclusion 119

References 121

A ACM Copyright Permission 137

B IEEE Copyright Permission 139

List of Figures

2.1 Transaction execution under (a) UNDO log and (b) REDO log . . . 8

2.2 (a) Ordering constraints of transaction execution in redo-log. (b) A log record. 10

2.3 LAD execution 12

2.4 Brief graph of Neural Network 14

2.5 **Convolution computation demonstration**: H, W, and C indicate the height, width, and channels of IFMAP. R and S indicate the height and width of the filters. K is the number of filters, equal to the number of channels in OFMAP. P and Q are the height and width of the matrix on each channel of OFMAP. 16

2.6 Activation function demonstration: Binary Step 17

2.7 Example of pooling: 3x3 filter over 4x4 feature map using stride=1. 19

2.8 Computational Graph of a two-layer convolutional neural network . 20

2.9 **Mapping type analysis** in DNNFusion [1] 21

2.10 Architecture of Neural Network Accelerator 23

2.11 Example of NCHW *im2col* 26

2.12 Example of NHWC *im2col* 26

2.13 **LoopNest presentations.** (a) Weight Stationary. (b) Output Stationary. (C) Input Stationary. 28

2.14 **Example of convolution computation on NNA with WS. Stride=1.** (a) IFMAP and Filters. (b) Convolution computation on NNA with Weight Stationary. 30

3.1 Transaction update demand distribution 37

3.2 Transaction types distribution. 38

3.3 TSTE architecture 40

3.4 Examples of Two-Stage Transaction Execution 42

3.5 Virtual ADR buffer (VADR) in the memory controller 45

3.6 **Undo logging and Redo logging**. (a)Undo log operation, (b) Redo log operation. 47

3.7 Transaction execution type distribution 52

3.8 Transaction throughput improvement over LAD 53

3.9 Log operations reduction 55

3.10 Throughput improvements over LAD with various ADR buffer sizes 56

4.1 Compare LALEA with conventional logging 63

4.2 LALEA log record organization 65

4.3 LALEA controller 67

4.4 Compare the redo logging and our proposed log metadata buffering 70

4.5 Improving transaction throughput 73

4.6 Reducing log entry persistence latency 74

4.7 Reducing latency to commit transaction 75

4.8 Improving transaction throughput with cache line level interleaving 76

5.1 **Example of repeated off-chip memory access to IFMAP in a Wide Layer.** LoopNest(a) and LoopNest(b) present the convolution computation with Weight Stationary before and after mapping filters on the Systolic Array (SA), respectively. The sequence number of Tiles indicates their execution order. IFMAP is read from off-chip memory to on-chip memory. Arrows with different colors correspond to different tiles of IFMAP. 81

5.2 Execution time distribution by layer types 84

5.3 Memory-stall time distribution by layer types 85

5.4 Illustration of ARAI 87

5.5 **Off-chip memory mapping examples.** Off-chip memory has 2 channels, 2 banks per channel, and 4 rows per bank. There are three concurrent data tiles. The numbers inside blocks indicate their order during memory allocation. (a) is based on ROMANet. (b) is based on LAP. 89

5.6 Architecture overview of LAP 94

5.7 **Key components in DTile Manager Unit (DMU):** Memory Location Table (MLT), DTile Mapping Table (DMT), Bank Status Table (BST), Off-chip DTile Tracking Table (GT), Reg0: The head of free MLTEntries, Reg1: The tail of free MLTEntries, Reg2: SelfMinBkRkCh in Algorithm 2, Reg3: MemMinBkRkCh in Algorithm 2, Reg4: NextBkRkChToCheck in Algorithm 3 95

5.8 Main steps of DRAM allocation for DTile in LAP 101

5.9 Main steps of reading DRAM location for off-chip DTile 103

5.10 Reduction in *Inference Time*. Baseline is DWS_ROMA. 111

5.11 Reduction in *DRAM Read Traffic*. 113

5.12 Increased *DRAM Bandwidth* by LAP. Baseline is ROMA 113

5.13 Increased Channel-Level Parallelism of DRAM. Baseline is ROMA. 114

5.14 Increased Bank-Level Parallelism of DRAM. Baseline is ROMA. . . 114

5.15 Reductions in *Inference Time* with different sizes of Systolic Array . 116

List of Tables

3.1 **Muclticore workloads** 39

3.2 **System parameters** 49

3.3 **Benchmarks used for evaluation** 49

3.4 **Multicore workloads** 51

4.1 **System parameters** 71

5.1 **Space requirement of key components in DMU** 105

5.2 **Average number of rowTasks per DTile under different operations**. Generation indicates *Generating DTile*. Read indicates *Reading DTile*. DWS and ARAI indicate that the different dataflow adopted. DWS and ARAI are explained in the evaluation 5.5.1. 106

5.3 **System parameters of NNA evaluation** 109

5.4 **Summarization of NN workloads.** Conv: the standard convolution layer. ConvDW: the depthwise convolution layer. Res: the residual block containing Conv and skip connection. FC: the fully connected layer. 109

Preface

This dissertation is submitted for the degree of Doctor of Philosophy at Michigan Technological University. It contains published, completed papers, some important preparations, and achievements for future publications completed by the author. The research is to the best of my knowledge and original, except where references are made to previous work. Part of this work contains previously published material. Zhiyuan Lu did all of the work under the supervision of Dr. Jianhui Yue from the Department of Computer Science at Michigan Technological University.

The important parts of the research work during this Ph.D. program are presented in 3 Chapters. Chapter 3 is the combination of two papers published by Zhiyuan Lu, etc. on ICCD2020 and NAS2021. Chapter 4 is based on the paper published by Zhiyuan Lu, etc. on DATE2022. These two chapters are focused on optimizing the logging operations in the system with non-volatile memory. Chapter 5 aims to mitigate the off-chip memory overhead in the neural network accelerators. Zhiyuan Lu and Dr.Jianhui Yue designed all related research. Zhiyuan Lu modified and developed the simulators to evaluate all related designs. Zhiyuan performed the statistical analyses following discussions with Dr. Jianhui Yue. The manuscript is coedited by Zhiyuan Lu and Dr.Jianhui Yue.

Acknowledgments

At Michigan Technological University, I have obtained valuable experience in learning and researching. I would like to express my sincerest gratitude to the Department of Computer Science for giving me the opportunity to pursue the Degree of DOCTOR OF PHILOSOPHY in Computer Science. I would like to express my deepest gratitude to all the people who have helped me, including but not limited to my academic advisor, advisory committee members, course instructors, and office assistants.

I would like to first thank my academic advisor, Dr. Jianhui Yue, who gave me a lot of instructions during my research work and the chance to have research practice. Without his guidance, I cannot make the work presented in this dissertation. I would like to thank the NSF for supporting my Ph.D. program. I also would like to thank all collaborators during my research work.

I would like to express my sincerest acknowledgment to my academic advisory committee members, Dr.Soner Onder, Dr.Zhenlin Wang, and Dr.Qinghui Chen. Without their support and help, I cannot keep pursuing and finally complete my Ph.D. program.

I would like to show my deepest thanks and love to my families. They support and encourage me unconditionally when I encounter difficulties and challenges.

I would like to thank the fish at Michigan State. Their selfless sacrifice provided me with a lot of fun and tasty food in my life after research work. I also would like to thank Jay Zhou. Thanks to his singing for accompanying me night after night of hard work. Without them, it is hard for me to keep a healthy mental status to solve the pressure and problems during my Ph.D. program.

List of Abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
ACM	Association for Computing Machinery
ADR	Asynchronous DRAM Refresh
AIT	Address-Indirection-Translation
ATOM	Atomic Durability in Non-volatile Memory through Hardware Logging
AccNum	The number of memory blocks to be accessed
Addr	Address
AllocNum	The number of memory blocks allocated
BLOM	Buffer log metadata in ADR buffer
BST	Bank Status Table
B_Tree	Balanced Tree
BkID	Index of Memory Bank
CNN	Convolutional neural network
CONV	Convolution
CPU	Central processing unit
ChID	Index of Memory Channel
Conv	The standard convolution layer
ConvDW	The depthwise convolution layer

D Cache	Data Cache
DDR	Double Data Rate
DIMM	Dual In-line Memory Module
DMT	DTile Mapping Table
DMU	DTile Manager Unit
DNN	Deep Neural Network
DRAM	Dynamic random-access memory
DTile	Tile of Data
DWS	Weight Stationary optimized by DNNFusion
FC	The fully connected layer
FCFS	First-Come-First-Serve
FE	Free entries collector
FH	Free header
FIFO	First-In-First-Out
FLE	Free log entry
GEMM	General Matrix Multiply
GID	The global unique index of DTile
GT	Table indexed by GID
im2col	The transformation of Convolution Computation into GEMM
I/D Cache	Instruction/Data Cache
I/O	Input and Output

IEEE	Institute of Electrical and Electronics Engineers
IFMAP	Input feature maps
IS	Input Stationary
LAD	Logless Atomic Durability
LAD-RdLog	LAD with redo log
LALEA	Log entry allocation scheme
LAP	Load-Aware Placement scheme for feature map tiles in off-chip memory
LEC	Log entry collation
LR	Log record registers
LSQ	Load-Store Queue
MAC	Multiply-Accumulate
MC	The memory controller
MLT	Memory Location Table
MLTEID	Index of MLTEntry
MLTEntry	Eentry of MLT
MLTHead	The index of the first MLTEntry in the list of MLTEntries
MSHR	Miss Status Holding Registers
MUDF	Minimal-Update-Demand-First
NCHW	Batch-Channel-Height-Width
NHWC	Batch-Height-Width-Channel
NN	Neural Network

NNA	Neural Network Accelerator
NVDLA	NVIDIA Deep Learning Accelerator
NVM	Non-volatile memory
NextRowID	Index of the next DRAM row to be allocated for a rowTask
NoC	Network On Chip
OFMAP	Output feature maps
OLTP	On-line transaction processing
OS	Output Stationary
OoO	Out-of-order execution
PE	Processing Element
Psum	Partial sum
rMLTEID	read MLTEID
robabgrachco	row-bank-bankgroup-rank-channel-column
RAI	Reduce the Repeated Access to IFMAP
RB_Tree	Red-black tree
REDU	Redo-based logging with Direct-Update
RNN	Recurrent neural network
ROB	Re-order buffer
ROMA	DRAM mapping policy proposed by ROMANet
Res	The residual block containing Conv and skip connection
RkID	Index of Memory Rank

RoID	Index of Memory Row
SA	Systolic Array
SPM	Scratchpad Memory
SRAM	Static random-access memory
STQ	Store Queue
SchNum	The number of rowTasks schedule
TATP	Telecommunication Application Transaction Processing Benchmark
TLB	Translation lookaside buffer
TPCC	Transaction Processing Performance Council Benchmark C
TPU	Tensor Processing Unit
TSTE-RdLog	TSTE with redo log
VADR	Virtual ADR buffer
VADR-RdLog	Virtual ADR buffers with redo log and DRAM cache
WPQ	Write pending queue
WS	Weight Stationary
wMLTEID	Write MLTEID
#Bank	The number of banks inside an off-chip memory rank
#Channel	The number of channels inside an off-chip memory
#Column	The number of columns inside an off-chip memory row
#Rank	The number of ranks inside an off-chip memory channel
#Row	The number of rows inside an off-chip memory bank

Abstract

The emergence of new non-volatile memory (NVM) technology and deep neural network (DNN) inferences bring challenges related to off-chip memory access. Ensuring crash consistency leads to additional memory operations and exposes memory update operations on the critical execution path. DNN inference execution on some accelerators suffers from intensive off-chip memory access. The focus of this dissertation is to tackle the issues related to off-chip memory in these high-performance computing systems.

The logging operations, required by the crash consistency, impose a significant performance overhead due to the extra memory access. To mitigate the persistence time of log requests, we introduce a load-aware log entry allocation scheme that allocates log requests to the address whose bank has the lightest workload. To address the problem of intra-record ordering, we propose to buffer log metadata in a non-volatile ADR buffer until the corresponding log can be removed. Moreover, the recently proposed LAD introduced unnecessary logging operations on multicore CPU. To reduce these unnecessary operations, we have devised two-stage transaction execution and virtual ADR buffers.

To tackle the challenge of low response time and high computational intensity associated with DNN inferences, these computations are often executed on customized accelerators. However, data loading from off-chip memory typically takes longer than computing, thereby reducing performance in some scenarios, especially on edge devices. To address this issue, we propose an optimization of the widely adopted Weight Stationary dataflow to remove redundant accesses to IFMAP in off-chip memory by reordering the loops in the standard convolution operation. Furthermore, to enhance the off-chip memory throughput, we introduce the load-aware placement for data tiles on off-chip memory that reduces intra/inter contentions caused by concurrent accesses from multiple tiles and improves the off-chip memory device parallelism during access.

Chapter 1

Introduction

1.1 Memory Wall

In modern computer systems, the memory wall represents a critical challenge that arises due to the growing performance disparity between the processors and the memory systems.

The processor's performance driven by improvements in microarchitecture and parallel processing, has increased rapidly. However, the speed and bandwidth of memory technologies have not kept pace with these advancements. This imbalance in performance leads to situations where the processor spends a significant portion of its time waiting for data to be fetched from memory, leading to underutilization of the processor and reduced overall system efficiency.

The memory wall is particularly pronounced in data-intensive workloads, artificial intelligence applications, and big data analytics, where rapid access to large datasets is essential. These types of workloads further exacerbate the challenges associated with the memory wall.

To address the memory wall problem and the off-chip memory bottleneck, computer architects and system designers have explored various solutions, including the use of multi-level cache hierarchies, non-volatile memory technologies, and improved memory access techniques. Additionally, software optimizations such as data locality and memory management strategies play a crucial role in mitigating this bottleneck.

In this dissertation, several designs are proposed to tackle the memory wall problem in the system with non-volatile memory and the neural network accelerator.

1.2 System with Non-volatile Memory

Non-volatile memory (NVM) aims to minimize the gap between memory and storage due to its large storage capacity, fast speed, non-volatility, and byte-addressability. However, one important challenge of the successful adoption of NVM in computer systems is to ensure the atomicity of transaction updates in the event of a system crash or power loss.

Crash consistency requires that all updates within a transaction are always committed to NVM in a nothing-or-all manner, even upon a system crash. Traditional systems adopt undo-log, redo-log, or a combination of both to guarantee crash consistency. With logging, transaction update is applied to in-place data after the log of modifications is stored in NVM.

The logging method suffers from inferior performance due to ordering constraints between logging and in-place update, which places the logging execution in the I/O critical path. With caches being transparent, software applications cannot predict when a dirty cache line is written to memory. Therefore, a memory update cannot be performed until its log has been written to NVM. Nevertheless, flushing cache lines and memory barriers introduced by software logging significantly degrade the overall system performance. To reduce the logging overhead, hardware/hardware-assisted logging methods [2, 3, 4, 5, 6] have been proposed to move the logging out of the critical path. However, no logging operations are eliminated in either software or hardware approaches, which is the root cause of the inefficiency.

In 2019, LAD [7] proposed a novel atomic in-place update without logging for those transactions whose write set is smaller than the ADR (Asynchronous DRAM Refresh) buffer [8]. Its main idea is that memory write requests of a transaction are accumulated in the ADR buffer. These dirty data are flushed to NVM when all update requests from this transaction have arrived at the ADR buffer. In case of a crash, the ADR technology utilizes the energy provided by super-capacitors to flush all buffered update requests to NVM, enabling the system to survive crashes. In this way, LAD can avoid writing logs for these buffered transactions, thus boosting the performance. However, when a multi-core system executes multiple transactions concurrently, these transactions compete for the limited ADR buffer. Once the ADR buffer runs out of space, LAD must resort to logging to ensure crash consistency, degrading system performance.

1.3 Neural Network Accelerator

Specialized neural network accelerators (NNAs) have been developed to enhance the performance of deep neural networks (DNNs) in diverse artificial intelligence fields, such as computer vision, natural language processing, recommendation systems, graph analytics, and robotics. These NNAs employ systolic arrays (SAs) to expedite computationally intensive convolution (CONV) operations, which are the most prevalent and time-consuming tasks in deep neural networks. To optimize data access for SA-based NNAs, a purpose-built on-chip memory is employed for the storage of input feature maps (IFMAP), output feature maps (OFMAP), as well as filters. This design minimizes the necessity for frequent access to slower off-chip memory, such as DDR3/4. Nevertheless, the limited storage capacity of on-chip memory still necessitates substantial off-chip memory accesses, as documented in the previous works [9, 10, 11, 12, 13, 14]. This, in turn, introduces additional latency in DNN inference.

To address the overhead incurred by accessing off-chip memory during DNN inference, researchers have proposed two primary approaches:

† Enhancing on-chip data reuse: This approach focuses on maximizing the reuse of on-chip data to minimize off-chip memory access. One strategy involves designing the data movement within NNAs, referred to as dataflow. Data of a particular type is stationed on NNAs with minimal movement, while other data is streamed through NNAs to perform neural network computations with the stationed data. These dataflows can be categorized into Weight Stationary (WS) [15, 16, 17, 18, 19, 20, 21, 22], Output Stationary (OS) [23], or Input Stationary (IS) [24]. WS, the most commonly adopted approach among current NN chip vendors like NVIDIA and Google [18, 20, 21], is utilized to minimize data movement. Another strategy involves operation fusion [1, 25, 26], which eliminates the need for off-chip memory access for intermediate data between fused operators. An example of this is the fusion of Convolution with Batch-Normalization, Activation, and Pooling, a technique implemented by current NN execution frameworks like Tensorflow [25] and TVM [26]. Given the diversity of neural networks, different models may require unique operator fusions,

necessitating further profiling based on model-specific characteristics.

† Optimizing off-chip data placement: This approach is dedicated to optimizing the allocation of data for tiles within the off-chip memory [14, 27, 28], with the aim of improving off-chip memory access speed.

Despite these optimizations, the speed of off-chip memory access can still act as a bottleneck in some NNAs [11, 14, 23, 27, 28, 29, 30, 31, 32]. Rather than customizing dataflow for specific NNAs [10, 33], the Weight Stationary is a widely adopted dataflow for NNAs utilizing systolic arrays [17, 18, 20, 21, 22], which consist of a matrix of processing elements. In this scenario, the unrolled filter must be tiled to perform general matrix multiplication (GEMM) since the matrix size of the systolic array is constrained. These layers are referred to as wide layers.

To complete convolution computations for wide layers, IFMAP is partitioned into multiple segments, each processed with a filter tile. This partitioning can lead to repeated accesses to the same portions of IFMAP. Furthermore, the STOA tile placement optimization described in ROMANet [14] doesn't fully account for memory contention arising from simultaneous off-chip memory access for different feature map segments during inference, leaving room for further enhancements in off-chip memory access efficiency.

Chapter 2

Background

The emphasis of this dissertation lies in memory optimizations for non-volatile memory systems and neural network accelerators. In this chapter, two distinct sections offer detailed introductions to the relevant background and related research. Section 2.1 delves into topics such as crash consistency, logging operations, log organizations, ADR, and LAD. Meanwhile, Section 2.2 presents comprehensive information on neural networks, computational graph, accelerator architecture, and developments related to neural network accelerators.

2.1 Logging in Non-volatile Memory

2.1.1 Crash Consistency in NVM systems

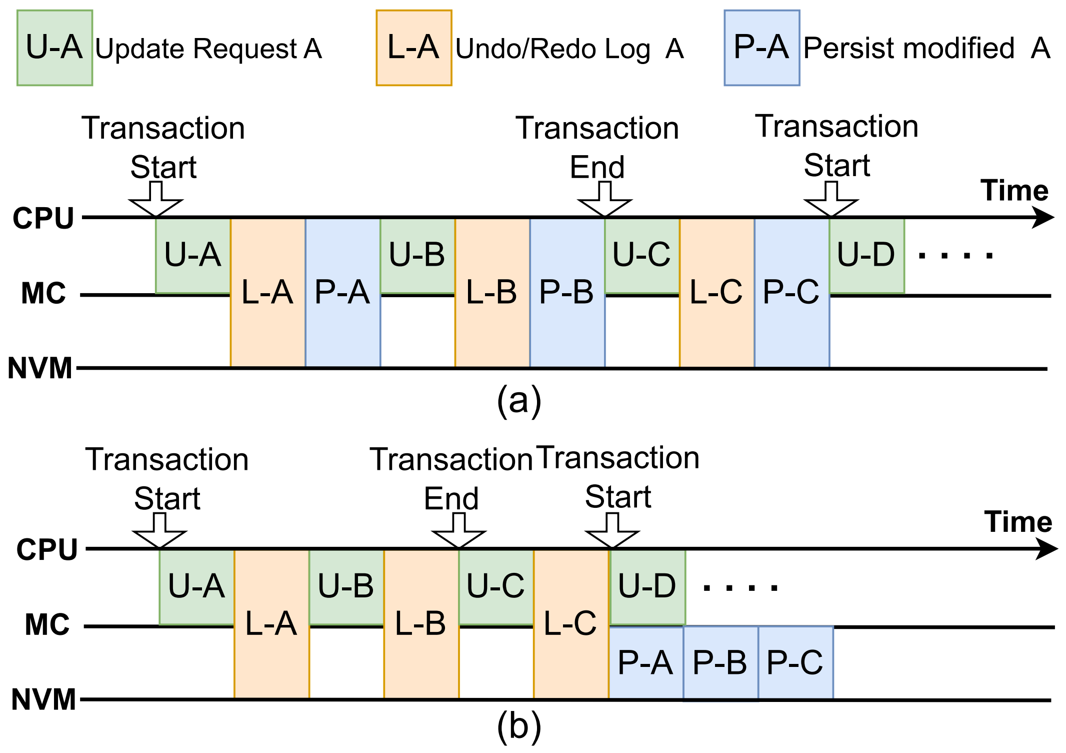


Figure 2.1: Transaction execution under (a) UNDO log and (b) REDO log

Crash consistency ensures that data on NVM are recoverable upon a system crash or power loss. Prior studies [2, 3, 4, 6, 34] have proposed software logging to achieve crash consistency. Based on log content, these logging schemes can be classified into two categories: (1) *undo-log*: data are copied to the log before they are modified. During recovery, logs are used to undo all changes made. (2) *redo-log*: new data are written to the log before in-place updates. During recovery, logs are used to redo all modifications.

To enforce the order constraint of write-log and write-data, software needs to run cache line flush instructions and store fence instructions in the write-log stage and write-data stage. Therefore, in these software schemes, the logging process is on the critical path, leading to inferior performance. Figure 2.1 shows the execution overhead introduced by logging with an example whose first transaction has 3 unique updates, A, B, and C. Under *undo-log*, log entries with unmodified data are persisted before their in-place updates are performed. Both persisting log entries and updates are on the critical path. Under *redo-log*, while log entries with modified data are persisted, their in-place updates are executed in the background after the transaction ends, with the fast transaction commit speed. *redo-log* directs read requests to the log entries to provide the latest values, incurring performance overhead.

Hardware-assisted logging methods are proposed to improve crash consistency performance and reduce the burden on programmers. For example, ATOM [2] proposes a hardware undo log, which initiates the write-log request for a store in the L1 cache and delays the update to L1 until its write-log is done. The modified cache lines are flushed to NVM later. The state-of-the-art hardware redo-log, REDU [6] buffers the latest updates in the DRAM cache after persisting log entries to NVM and hence avoids reading log entries from slow NVM when performing in-place updates. However, these state-of-the-art designs still suffer from (1) handling complicated constraints on ordering write-log and write-data, and (2) generating significant log write traffic and reducing NVM's lifetime.

2.1.2 Conventional Log Organization

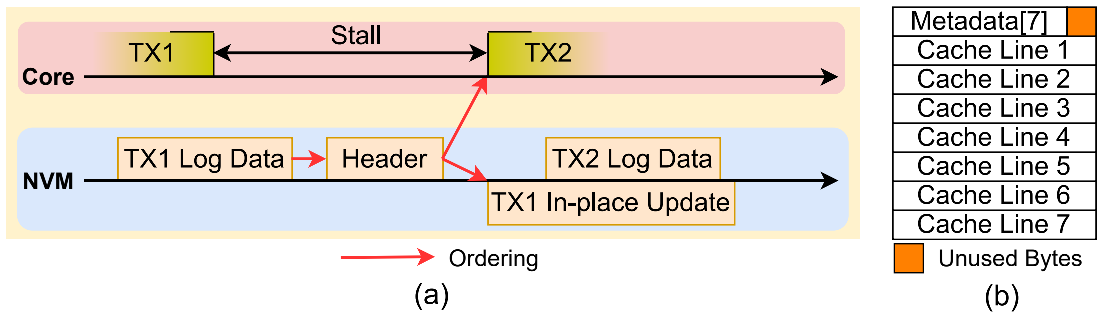


Figure 2.2: (a) Ordering constraints of transaction execution in redo-log. (b) A log record.

A transaction log includes 64-byte data content and an 8-byte home address for each write request that occurred in this transaction. The address information in the log is referred to as log metadata. To reduce the log metadata write traffic, ATOM [2] proposes log entry collation (LEC) that co-locates seven log blocks' metadata in a 64-byte block referred to as a header. These seven log entries and the header constitute a log record, shown in Figure 2.2(b). A log record header can be persisted to NVM only after its log data blocks are written to NVM. This is the ordering constraint for persisting a log record, and it is referred to as intra-record ordering. The intra-record ordering is on the critical execution path for both the redo and undo logging scheme.

Figure 2.2(a) shows the ordering under redo logging.

However, LEC is sub-optimal in terms of write traffic reduction when the number of write requests in a transaction is smaller than seven. For example, if a transaction writes one cache line, its 64B log record header stores an 8B address, wasting NVM write bandwidth. Whisper [35] shows most transactions have less than two write requests in the NVM workloads. In addition, the log metadata traffic accounts for 12.5% of log traffic for transactions with a large number of write requests, degrading performance, and NVM endurance.

2.1.3 Asynchronous DRAM Refresh (ADR)

Intel introduced a new technology ADR [8], which leverages energy held in capacitors to ensure that pending write requests received by the memory controller (MC) will be persisted to NVM even upon a power failure. The write pending queue (WPQ), which holds write requests, is called the ADR buffer. The memory controller sends an acknowledgment to the CPU immediately after the ADR buffer receives a *clwb* instruction. This technology provides an exciting opportunity to efficiently achieve crash consistency.

The energy stored in capacitors can also flush the memory controller buffer and AIT cache to NVM upon crash [36]. Therefore, ADR makes the memory controller become part of the persistence domain. However, it is challenging to provision the large ADR buffer due to the memory controller's power consumption and the capacitor size limited by DIMM [37]. Typically, the ADR buffer can accommodate 24 or more cachelines [2, 38, 39].

2.1.4 Logless Atomic Durability (LAD)

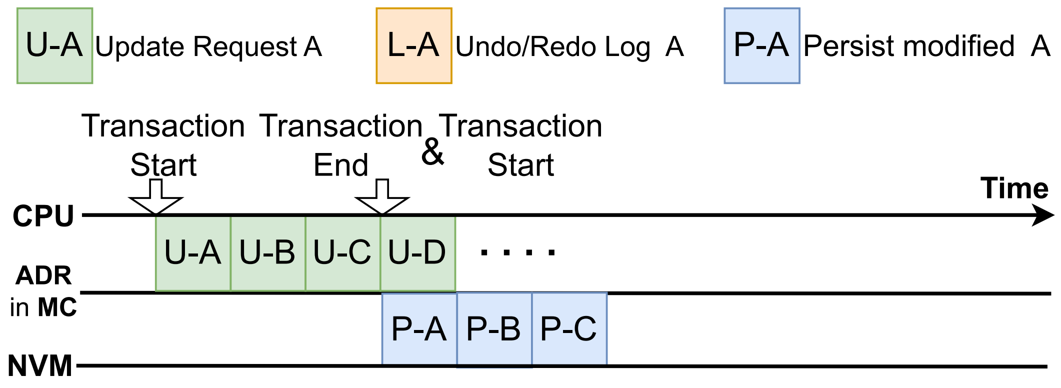


Figure 2.3: LAD execution

To reduce log overhead, LAD [7] proposes the hardware mechanism that ensures a transaction's updates atomically persisted to NVM without logging, by exploiting ADR. LAD stores updates of a transaction in the ADR buffer as it is running and flushes these updates to NVM when the transaction ends. If a power failure happens before an in-flight transaction ends, the partial log entries are discarded, without affecting corresponding data stored in NVM. If a power failure happens after an in-flight transaction ends, ADR can ensure that the buffered updates are safely persisted to NVM, since the ADR buffer is part of the persistence domain. Figure 2.3 shows how LAD works with a simple example. Assuming a transaction has 3 updates A, B, and C, they are accumulated in the ADR buffer which is in MC as it is running and LAD persists them after the transaction ends, without introducing log operations. However, when the ADR buffer overflows, LAD falls back to hardware logging, by writing log entries for speculative updates stored in the ADR buffer. After log entries are persisted, their corresponding in-place updates are performed, losing the benefits of LAD.

2.2 Neural Network and Neural Network Accelerator

2.2.1 Neural Networks

Neural network (NN) is a type of artificial intelligence model that has gained significant attention in recent years due to its ability to perform complex tasks such as image recognition, natural language processing, and decision-making. It is modeled after the structure and function of the human brain, capable of learning and adapting to new information through a process known as training. The basic building block of a neural network is the artificial neuron, also known as a node. These neurons are connected in layers, with each neuron in one layer connecting to one or more neurons

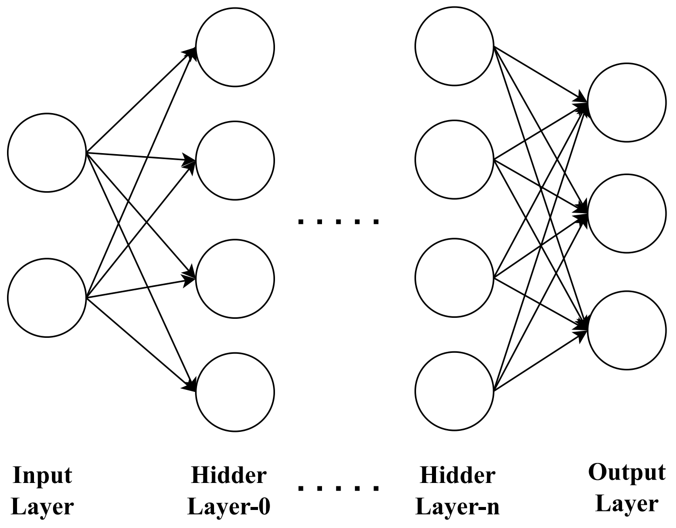


Figure 2.4: Brief graph of Neural Network

in the next layer. As shown in Figure 2.4, the input layer receives input data, such as an image or a sentence, and the output layer produces a prediction or decision based on the input. The intermediate layers are known as hidden layers and are responsible for learning and processing information.

There are several types of neural networks, including convolutional neural networks (CNN), recurrent neural networks (RNN), and deep neural networks (DNN). CNN is commonly used for image and video recognition, while RNN is used for sequential data such as speech and text. DNN has many hidden layers and has been successful in a wide range of applications, including speech recognition, natural language processing, and image and video analysis.

Neural networks have proven to be very effective in many applications and have demonstrated state-of-the-art performance in tasks such as image recognition and natural language processing. They have also been used in a wide range of industries, including healthcare, finance, and marketing. However, neural networks can be computationally expensive and require a large amount of data. As such, research continues to focus on improving the efficiency and performance of neural networks, as well as developing new architectures to tackle specific problems.

2.2.2 Computations in Neural Network

The standard convolution computation, referred to as convolution computation in this work, constitutes a fundamental operation in various neural network architectures, particularly in DNN models employed for image and video recognition tasks. Convolution computation entails the application of a set of filters to the input feature map (IFMAP), wherein these filters are systematically moved across the input space. At each spatial location, element-wise multiplication and summation operations are executed. During the convolution process, every filter is convolved with every pixel within the IFMAP, resulting in the generation of pixels on a channel of the output feature map (OFMAP). This procedure is iterated for each filter, ultimately yielding the OFMAP with multiple channels, each of which encodes distinct characteristics or aspects of the input data.

Figure 2.5 provides the demonstration of convolution computation. The input data

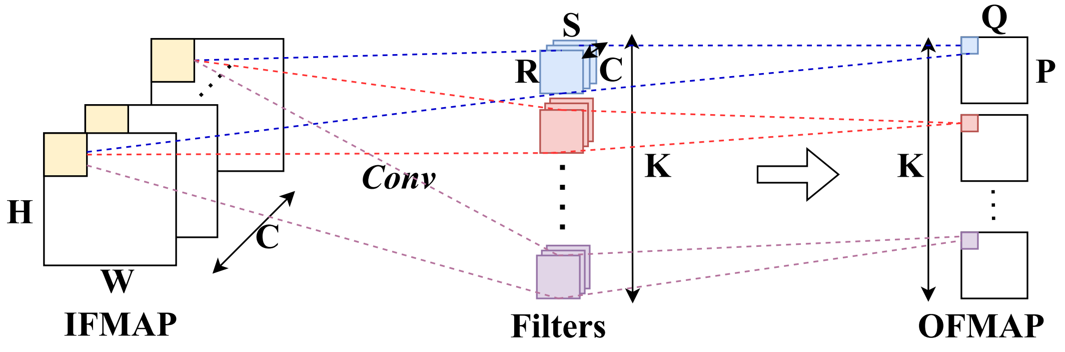


Figure 2.5: Convolution computation demonstration: H, W, and C indicate the height, width, and channels of IFMAP. R and S indicate the height and width of the filters. K is the number of filters, equal to the number of channels in OFMAP. P and Q are the height and width of the matrix on each channel of OFMAP.

denoted as IFMAP, consists of **C** channels and is represented as matrices of dimensions $\mathbf{H} \times \mathbf{W}$ with consistent shapes across channels. The filters, which serve as weights, are **K** matrices, each with dimensions $\mathbf{R} \times \mathbf{S} \times \mathbf{C}$. OFMAP represents the output, with the number of channels in OFMAP equal to **K**, corresponding to the number of filters. The dimensions **P** and **Q** denote the height and width of the OFMAP. Equation 2.1 and Equation 2.2 describe the relationships between the dimensions of IFMAP, filters, and OFMAP within the convolution computation process, assuming no padding. The term **Stride** refers to the number of pixels by which the filter shifts over IFMAP between steps, in the height or width directions. Equation 2.3 defines the mathematical calculation for each step, resulting in an output pixel. Each output pixel O(p, q) on output channel k is derived by summing the products across three dimensions: **C**, **R**, and **S**. As depicted in Figure 2.5, each matrix in OFMAP is the outcome of convolving a filter of dimensions $\mathbf{R} \times \mathbf{S} \times \mathbf{C}$ with all matrices in

IFMAP whose dimensions is $\mathbf{H} \times \mathbf{W} \times \mathbf{C}$.

$$P = (H - R + Stride)/Stride \tag{2.1}$$

$$Q = (W - S + Stride)/Stride \tag{2.2}$$

$$O(k,p,q) = \sum_{c=0}^{C-1}\sum_{r=0}^{R-1}\sum_{s=0}^{S-1} I(c, p+r, q+s) * W(k,c,r,s) \tag{2.3}$$

Besides the standard convolution computation, there are other types of convolution operations, like Depthwise Convolution computation [40]. However, non-standard convolution operations are not as commonly adopted in current NN algorithm designs as standard convolution computation.

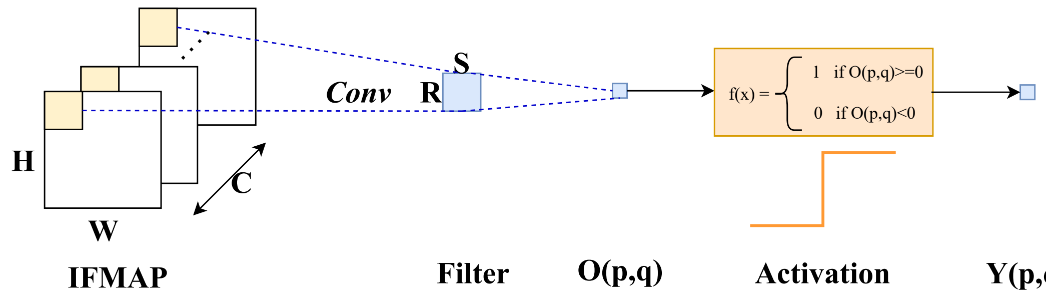


Figure 2.6: Activation function demonstration: Binary Step

Activation functions are another essential component of artificial neural networks.

They are used to introduce nonlinearity into the output of a neuron, enabling neural networks to model complex, nonlinear relationships between input data and output predictions. As illustrated in Figure 2.6, the output pixel of a convolution operation, O(p,q), is passed through an activation function, Binary Step, to produce a new output pixel, Y(p,q). The most commonly used activation functions include Binary Step 2.4, Sigmoid 2.5, Tanh 2.6, and ReLU 2.7. Choosing the right activation function is crucial for achieving optimal model performance and training stability.

$$f(x) = \begin{cases} 1 & x >= 0 \\ 0 & x < 0 \end{cases} \tag{2.4}$$

$$f(x) = 1/(1 + e^{-x}) \tag{2.5}$$

$$f(x) = 2/(1 + e^{-2x}) - 1 \tag{2.6}$$

$$f(x) = \begin{cases} max(0, x) & x >= 0 \\ 0 & x < 0 \end{cases} \tag{2.7}$$

Pooling is a technique commonly used in neural networks to reduce the matrix shape of OFMAP while preserving critical features. It involves dividing the feature map into non-overlapping or overlapping sub-regions and then applying a mathematical operation to each sub-region. The resulting output has a decreased spatial resolution, while the original number of channels is maintained. Figure 2.7 provides an example of pooling, where the size of the feature map is reduced from 4×4 to 2×2. During this process, the 4×4 feature map is partitioned into multiple 3×3 overlapping sub-regions. Max pooling and average pooling are two common types of pooling, where max pooling returns the maximum value in each sub-region, while average pooling returns the average value.

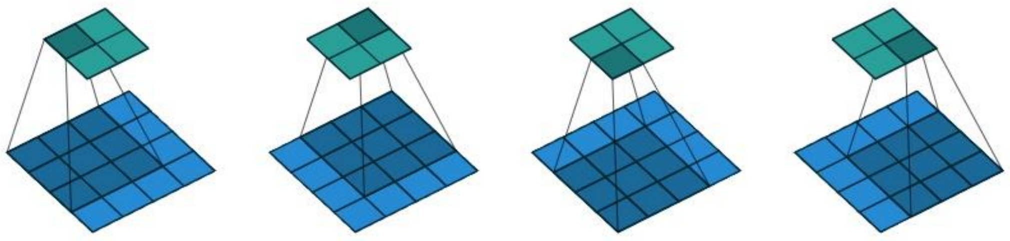


Figure 2.7: Example of pooling: 3x3 filter over 4x4 feature map using stride=1.

2.2.3 Computational Graph

A computational graph is a fundamental concept in the field of neural networks and machine learning, serving as a graphical representation of mathematical operations and dependencies within a model. It provides a way to visualize and understand the flow of data through a network.

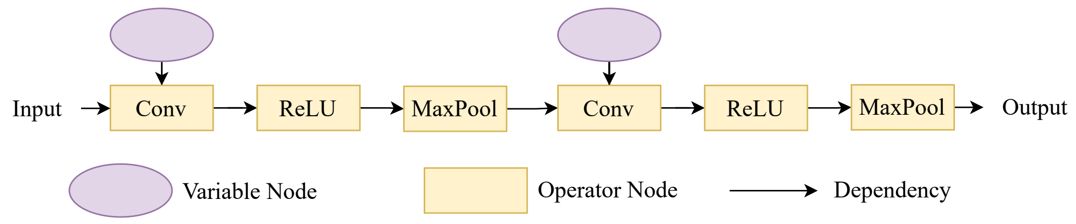


Figure 2.8: Computational Graph of a two-layer convolutional neural network

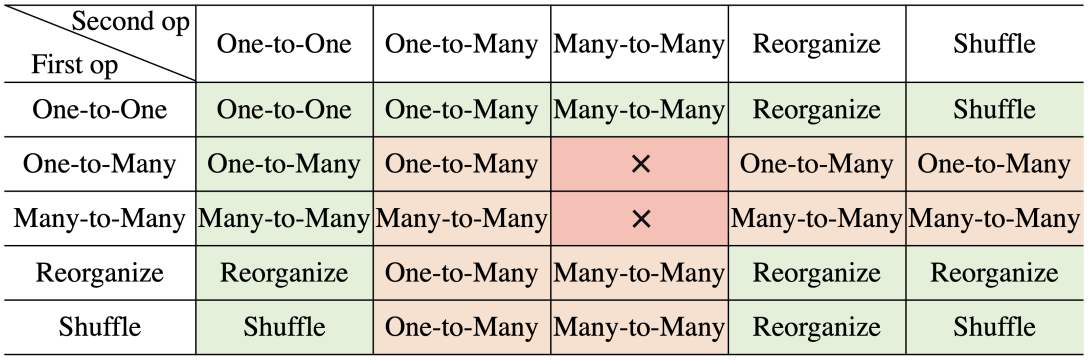
As the example shown in Figure 2.8, a computational graph consists of nodes and edges, where nodes represent mathematical operations or transformations, and edges represent the data dependency between these operations. The graph is directed, indicating the order in which computations are performed. There are two main types of nodes in a computational graph:

† Operator Nodes: These nodes represent mathematical operations, such as addition, multiplication, or more complex operations like convolution or activation functions (e.g., ReLU). Each operation takes input data, performs a specific computation, and produces output data.

† Variable Nodes: These nodes represent the learnable parameters of the model, such as weights and biases in a neural network.

To improve the efficiency of DNN inference, operator fusion is a key optimization for computational graph in DNN execution frameworks, such as TensorFlow [25] and

TVM [26]. To further explore fusion opportunities, DNNFusion [1] developed a classification of both individual operator and their combinations. The fundamental concept involves categorizing operators into distinct types and establishing rules for different combinations of the types. Based on the correspondence between input and output, operators are categorized into five types: one-to-one, reorganize, shuffle, one-to-many, and many-to-many. In one-to-one operations, input pixels are directly mapped to output pixels, exemplified by BatchNormalization. Reorganize and shuffle can be considered variations of one-to-one with special mappings. Reorganize changes the input dimensions without data movement, as seen in operators like Reshape. Shuffle permutates the input dimensions with data movement, exemplified by operators like Transpose. A representative operator featuring one-to-many mapping is Expand, which broadcasts input tensors based on the given shape and rule. Many-to-many mapping operators involve multiple input elements serving as operands to generate one or more output elements, such as the standard convolution.



Second op / First op	One-to-One	One-to-Many	Many-to-Many	Reorganize	Shuffle
One-to-One	One-to-One	One-to-Many	Many-to-Many	Reorganize	Shuffle
One-to-Many	One-to-Many	One-to-Many	×	One-to-Many	One-to-Many
Many-to-Many	Many-to-Many	Many-to-Many	×	Many-to-Many	Many-to-Many
Reorganize	Reorganize	One-to-Many	Many-to-Many	Reorganize	Reorganize
Shuffle	Shuffle	One-to-Many	Many-to-Many	Reorganize	Shuffle

Figure 2.9: Mapping type analysis in DNNFusion [1]

DNNFusion conducted a fusion analysis based on the mapping type of operators, as depicted in Figure 2.9. The first column and the first row, depicted without any color, indicate the mapping types of the first and second operators slated for fusion. The colored cells denote the resulting mapping type of the fused operator. This analysis further categorizes the fusion of these mapping type combinations into three groups, represented by green, orange, and red, respectively. Green denotes fusions that are deemed legal and advantageous. Red signifies fusions known to be either illegitimate or clearly unprofitable. Orange indicates that while these fusions are legitimate, their profitability necessitates further examination.

With such exploration on operator fusion, the optimized computational graph not only eliminates unnecessary computations but also reduces memory accesses for intermediate results, leading to enhanced efficiency in DNN inferences.

2.2.4 Neural Network Accelerators

Neural network accelerators (NNAs) are specialized hardware designed to improve the throughput of neural network training/inference tasks. The intensive computations in NN can be accelerated with targeted optimization for operations that are commonly used in NNs, such as matrix-matrix computations and vector-matrix computations. For NNAs tailored for DNN inferences, it is expected to have expeditious accesses to data and the swift completion of computations, arising from the necessity to obtain inference results with low latency.

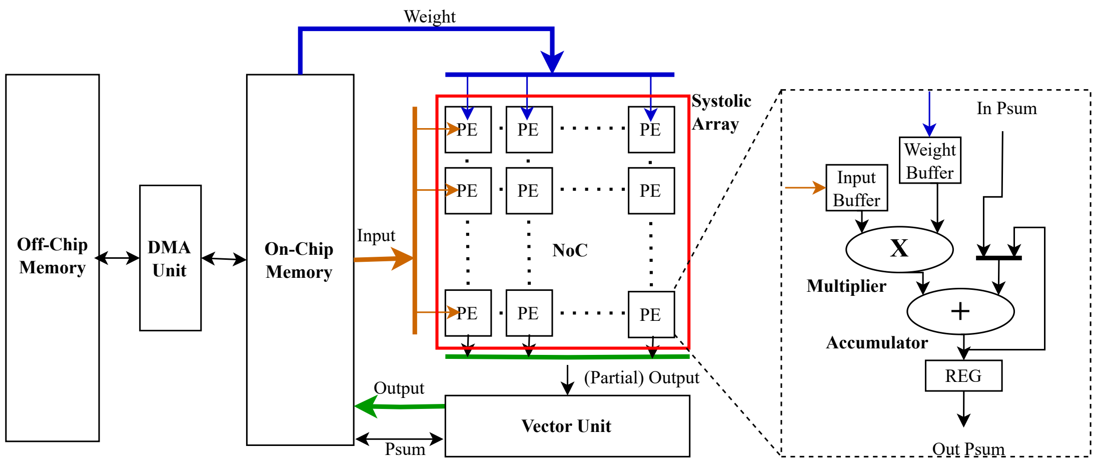


Figure 2.10: Architecture of Neural Network Accelerator

Figure 2.10 depicts a simplified NNA architecture, comprising a Systolic array (SA) that includes a homogeneous network of tightly coupled Processing Elements, a Vector Unit, on-chip memory, and off-chip memory.

Within the Systolic array, each Processing Element (PE) exhibits the capability to independently and simultaneously conduct Multiply-Accumulate (MAC) operations in parallel with other PEs. Figure 2.10 also provides a detailed focus on the primary components of the PE. The multiplier and accumulator within the PE are responsible for executing multiplication and addition operations, which are pivotal for efficient matrix multiplication. In this illustrated example with Weight Stationary, buffers are employed to store both the weights and inputs of a neural network. Simultaneously, registers are utilized for temporarily storing intermediate results during computation. These PEs are typically homogeneous, signifying that they share an identical architecture and functionality. Network on Chip (NoC) is a communication infrastructure that is designed to connect PEs in a Systolic array. The NoC serves as a high-bandwidth communication network that enables the PEs to exchange data and coordinate their operations efficiently. As the example illustrated in Figure 2.14, NoC is employed to transfer weights from the top of the PEs matrix to the bottom, prefilling each PE with the required data. During the convolution computation, input pixels are streamed from the left to the right through the NoC. Additionally, the intermediate result, partial sum (Psum), within each PE is drained from the top to the bottom using the NoC. By harnessing the parallelism inherent in the processing elements, the Systolic Array can achieve significantly higher performance levels in comparison to traditional single-processor architectures.

The Vector Unit serves as an adjunct coprocessor, tightly integrated with SA. It specializes in executing vector operations, including accumulation, activation, normalization, and pooling [17, 18, 41]. Its inputs are derived from the data drained from the SA and the data buffered in on-chip memory, while its outputs are directed back to the on-chip memory.

In NNAs, compiler-controlled on-chip memory offers the advantages of both higher speed and increased energy efficiency when compared to off-chip memory. Among the types of on-chip memory used in NNAs, Scratchpad Memory (SPM) is a common choice [9, 14, 23, 29, 42, 43]. SPM is typically organized as a small, multi-bank SRAM, and each bank can be independently accessed, enabling concurrent access to multiple data elements. Another prevalent form of on-chip memory in NNAs is Vector Memory, which is specifically designed for processing and manipulating extensive arrays of data in a vectorized fashion, as highlighted in [17, 18, 41]. In some designs, operations, like accumulation, are merged into the vector memory. The utilization of vector memory can notably enhance the performance of vector operations. On-chip memory not only allows for simultaneous on-chip access by the Systolic Array to multiple tensor elements but also facilitates the reading and writing of data to and from off-chip memory during on-chip computations. This functionality effectively reduces the latency associated with off-chip memory access, which is essential for storing the initial input and intermediate data generated during NN inference due to the limited size of on-chip memory.

2.2.5 *im2col* methods: NCHW vs. NHWC

$$Offset\_nchw(n, c, h, w) = n \times CHW + c \times HW + h \times W + w \tag{2.8}$$

$$Offset\_nhwc(n, c, h, w) = n \times HWC + h \times WC + w \times C + c \tag{2.9}$$

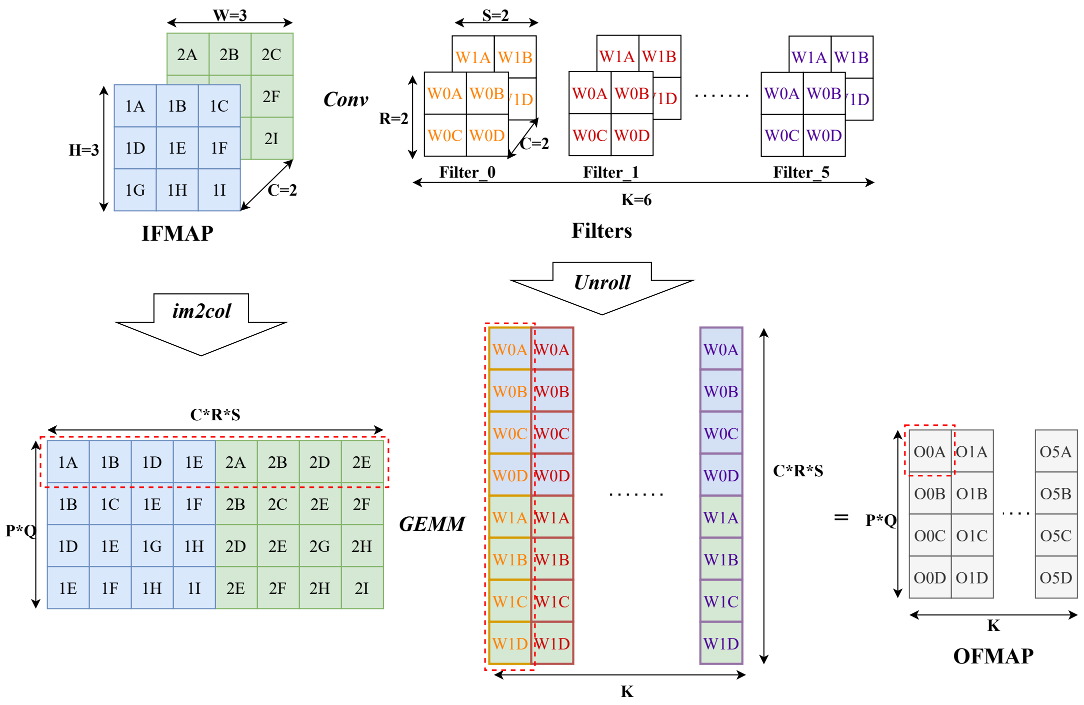


Figure 2.11: Example of NCHW *im2col*

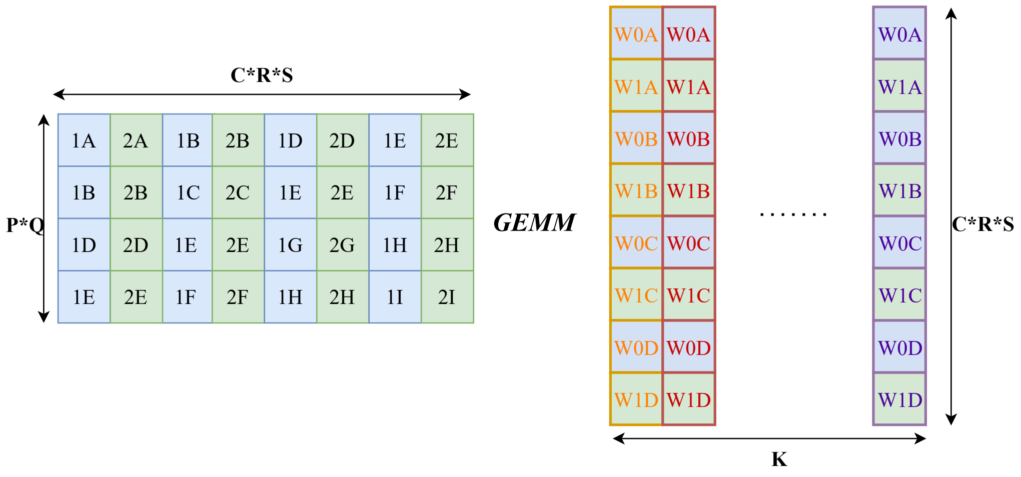


Figure 2.12: Example of NHWC *im2col*

$$O0A = 1A \times W0A + 1B \times W0B + ... + 2D \times W1C + 2E \times W1D \tag{2.10}$$

Processing Elements in NNAs enable the execution of General Matrix Multiply (GEMM) operations by performing parallel MAC operations. This necessitates the transformation of Convolution Computation into GEMM using a technique known as *im2col* transformation. An example of *im2col* is provided in Figure 2.11, where both filters and IFMAP are unrolled into 2D matrices. Two commonly used implicit *im2col* algorithms are NCHW [42, 44] and NHWC [43]. In the NCHW format, pixels from the same channel of IFMAP are continuously placed, as shown in Figure 2.11. As a result, filters are also unrolled in accordance with the NCHW format. In this example, each column of unrolled weights corresponds to each filter, positioned above from left to right. The blue and green squares signify the associated input channel, indicating that weights within the same channel are sequentially placed in the NCHW *im2col* format. However, due to hardware scalability limitations in stride Convolution Computation when using NCHW, the NHWC format was introduced in 2021 by Zhou [43]. This format has since been embraced as the default format in contemporary machine learning frameworks [25] and recommended by various vendors [45]. As alternating blue and green squares shown in Figure 2.12, elements from different channels are prioritized to be placed together in NHWC format.

Equation 2.8 and Equation 2.9 elucidate the pixel's offset within the tensor layout for NCHW and NHWC, respectively. With the *im2col* transformation, the OFMAP is derived using GEMM. Equation 2.10 provides an example to obtain O0A, illustrating that each element within the OFMAP represents the result of a MAC operation

Shape of IFMAP = [C, H, W],   Shape of Filters = [K, C, R, S],   Shape of OFMAP = [K, P, Q]

```
for k : [0, K)
 for r : [0, R)
  for s : [0, S)
   for c : [0, C)
    for h : [0, H)
     for w : [0, W)
      ofmap[k,h-r,w-s] +=
      ifmap[h,w,c]*filter[k,r,s,c]
```

(a)

```
for k : [0, K)
 for p : [0, P)
  for q : [0, Q)
   for c : [0, C)
    for r : [0, R)
     for s : [0, S)
      ofmap[k,p,q] +=
      ifmap[p+r,q+s,c]*filter[k,r,s,c]
```

(b)

```
for c : [0, C)
 for h : [0, H)
  for w : [0, W)
   for k : [0, K)
    for r : [0, R)
     for s : [0, S)
      ofmap[k,h-r,w-s] +=
      ifmap[h,w,c]*filter[k,r,s,c]
```

(c)

Figure 2.13: LoopNest presentations. (a) Weight Stationary. (b) Output Stationary. (C) Input Stationary.

performed on a row and column from two unrolled matrices.

2.2.6 Dataflow in Neural Network Accelerators

Since hardware resources are limited, it is necessary in practice to partition big data sets or layers into smaller chunks, which is called tiling [15]. Tiling divides a large data set into smaller sub-regions, called tiles. Since tiling reduces the memory requirements of the computation, a tile can be stationed on an SA, improving data reuse.

Dataflow refers to the way data is processed and propagated within and through NN layers. To optimize the execution of neural network algorithms, various dataflow algorithms have been proposed. Based on the data type stationed, here are listed three dataflows:

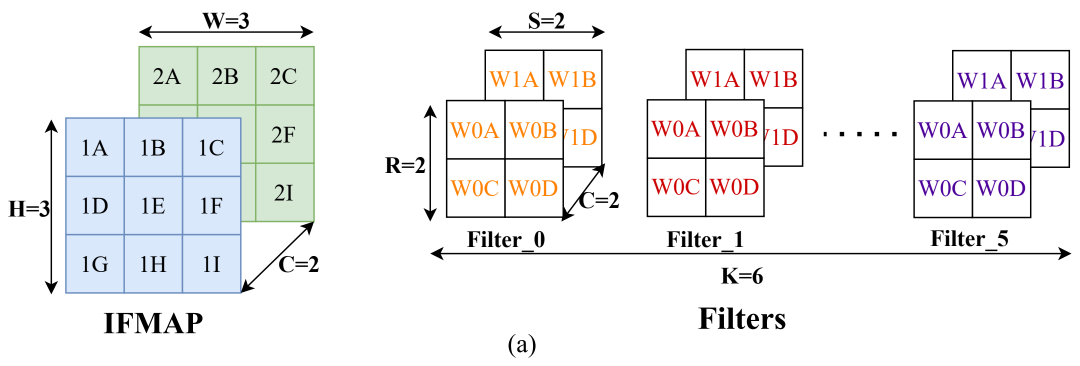
† Weight Stationary (WS) [15, 16, 17, 18, 19, 20, 21, 22]: WS stations pixel of

filters on SA. Once a tile of filters is mapped onto SA, they are kept in the buffers of PEs until all the computations involving the given set of filters are finished. Every cycle, the pixels of IFMAP required to be multiplied with the loaded pixels of filters are streamed into SA, generating partial sums. Figure 2.13(a) illustrates the loop nest presentation of WS, where the iteration of filters is in the outermost loop, enabling the highest degree of reuse.

† Output-Stationary (OS) [23]: Registers of PEs are reserved for the pixels of OFMAP. Both the input and weight are streamed through SA to generate the partial sum. Partial sums are kept and accumulated inside each PE to get the final output. The LoopNest of OS is depicted in Figure 2.13(b).

† Input-Stationary (IS) [24]: Instead of stationing pixels of filters, a tile of IFMAP is kept on SA, while the weights are streamed through to complete the computation. Figure 2.13(c) indicates LoopNest of IS, in which the input is highest reused.

Among various NNA architectures, different Weight Stationary, Output Stationary, and Input Stationary dataflows each could come with their respective advantages and drawbacks. In addition, there are customized dataflow approaches tailored for specific hardware, such as ISOSceles [33], which have demonstrated superior performance. However, it is common for hardware vendors to adopt Weight Stationary dataflow, as seen in TPUs [17, 18] and NVDLAs [19, 20, 21], for their NNAs [15].



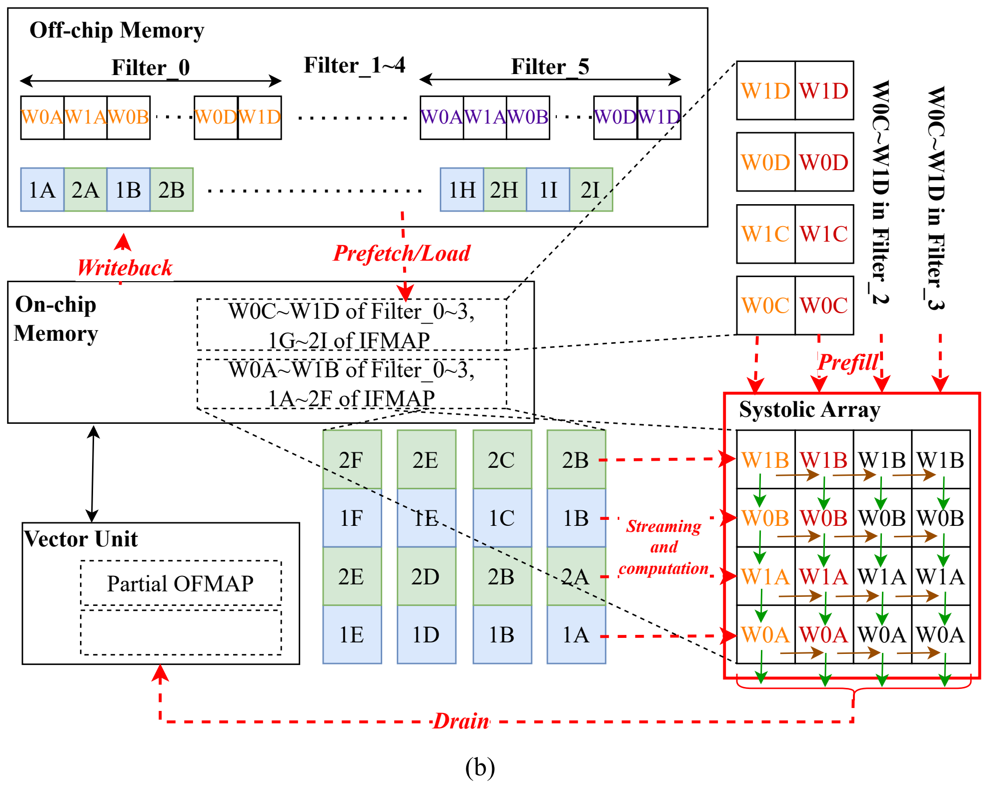


Figure 2.14: Example of convolution computation on NNA with WS. Stride=1. (a) IFMAP and Filters. (b) Convolution computation on NNA with Weight Stationary.

Figure 2.14 illustrates the key steps of convolution computation in an NNA, employing a Weight Stationary dataflow. In Figure 2.14(a), the IFMAP is shown as having dimensions of 3×3 with 2 channels (H=3, W=3, C=2), and there are 6 2×2 filters

(R=2, S=2, K=6) for each input channel, denoted as $Filter\_0$ to $Filter\_5$. The Systolic Array is composed of a 4×4 PEs matrix, and all the data is unrolled and stored in off-chip memory in NHWC format. Due to the limited size of the SA, the convolution computation of a layer necessitates multiple tiles. In this specific example, there are 4 tiles involved. Figure 2.14(b) outlines these steps. The on-chip memory employs double-buffering, which enables data related to the next tile to be prefetched from off-chip memory. Apart from the computation on SA, there are 5 main steps involved in the data flow on the NNA. The example shown in Figure 2.14(b) assumes that the NNA is at the beginning of computation for weights $W0A \sim W1B$ of $Filter\_0 \sim Filter\_3$.

1. Prefetch/Load: The first step involves transferring data from off-chip memory to on-chip memory. In this instance, $W0C \sim W1D$ of $Filter\_0 \sim Filter\_3$ and $1G \sim 2I$ of IFMAP will be prefetched from off-chip memory once both $W0A \sim W1B$ of $Filter\_0 \sim Filter\_3$ and $1A \sim 2F$ have been loaded into on-chip memory.

2. Prefill: The second step entails loading data from on-chip memory into registers of PEs. This data will remain stationed on the PEs throughout the computation of the current tile. In this particular illustration, the SA has been preloaded with $W0A \sim W1B$ of $Filter\_0 \sim Filter\_3$, and these filter pixels will remain within the SA until the completion of GEMM operation for this set of weights.

The subsequent data to be loaded into the SA is $W0C \sim W1D$ of $Filter\_0 \sim Filter\_3$.

3. Streaming and Computation: The third step encompasses the continuous streaming of data from on-chip memory to PEs for their computation within SA. In the illustrative example presented in Figure 2.14, the data range $1A \sim 2F$ is streamed to the PEs for conducting MAC with the stationary filter pixels. Following a MAC operation on a PE, the input pixel is passed on to the adjacent PE to the right. The Psum produced is then forwarded to the next PE below, where it undergoes accumulation and is subsequently stored in on-chip memory.

4. Drain: Following the computation, the remaining partial sums residing in the registers of PEs are transferred to the Vector Unit, where they undergo final left vector operations. It's noteworthy that during the computation and draining phases, vector operations can be executed simultaneously within the Vector Unit, hiding the latency caused by them.

5. Writeback: Due to the limited capacity of on-chip memory, the intermediate results produced during inference need to be written back to off-chip memory. In this instance, it is assumed that the on-chip memory can buffer intermediate results within a layer but lacks the capability to store the entire OFMAP. Consequently, Partial OFMAP, $OFMAP\_0 \sim OFMAP\_3$, will be written back to off-chip memory once all elements have attained their final values subsequent

to accumulation in the vector unit.

2.2.7 Off-chip Memory Mapping in NNAs

Algorithm 1: Pseudo-code of DRAM Mapping proposed by ROMANet

```
Data:
$\#Channel$: The number of channels in DRAM.
$\#Rank$: The number of ranks per DRAM Channel.
$\#Bank$: The number of banks per DRAM Rank
$\#Row$: The number of rows per DRAM Bank.
$\#Column$: The number of columns per DRAM Row.
1 for $ch \leftarrow 0$ to $(\#Channel - 1)$ do
2     for $rk \leftarrow 0$ to $(\#Rank - 1$ do
3         for $ro \leftarrow 0$ to $(\#Row - 1)$ do
4             for $bk \leftarrow 0$ to $(\#Bank - 1)$ do
5                 for $co \leftarrow 0$ to $(\#Column - 1)$ do
6                     //map a tile of data to
7                     DRAM$[ch, rk, bk, ro, co]$
```

ROMANet [14] proposed an approach for mapping data of Neural Networks in of-chip memory based on the tiling, resulting in improved DRAM access speed through enhanced row buffer hit rates and the utilization of multi-bank burst feature in DRAM. In comparison to Caffeine [27], SmartShuttle [11], and BWA [28], ROMANet's DRAM Mapping technique achieves higher throughput while minimizing access energy and latency.

Algorithm 1 provides a pseudo-code representation of the mapping policy for a data tile, referred to as DTile in this research. At line 5, ROMANet gives priority to mapping feature map elements to different columns within the same DRAM row to maximize row buffer hits. This process can be carried out in parallel across different chips if applicable, to leverage chip-level parallelism. If all columns within the same row are fully occupied, any remaining data are mapped to different banks within the same DRAM chip to exploit bank-level parallelism (line 4). Mapping to different DRAM banks can also be executed in parallel across different chips if applicable. For each bank, data are mapped to different columns within the same row, mirroring the procedure described in line 5. Here, if all columns within the same row are filled, any remaining data are mapped to a different row (line 3). These steps (lines 3∼5) are iterated until all data are mapped within a DRAM rank. If any data remain, they can be mapped to different DRAM ranks (line 2) and channels (line 1), respectively, if applicable, using the same process outlined in lines 3∼5.

In ROMANet, off-chip memory access requests are sequentially directed to different DRAM banks. Specifically, if bank0 was accessed in the previous cycle, the subsequent request will be directed to bank1, and so forth. Furthermore, ROMANet employs the First-Come-First-Serve (FCFS) scheduling policy. FCFS ensures that memory requests are serviced in the order they are received, without any prioritization or reordering based on various policies. This approach simplifies the design of the memory controller and reduces its complexity.

Chapter 3

Improving the Performance of NVM Crash Consistency under Multicore

3.1 Overview

In this work, it is observed that in our studied 4-core workloads, up to 59.5% transactions' log operations would be avoided if these transactions ran alone, without any ADR buffer contention. We propose two algorithms to reduce unnecessary log operations for concurrent transactions, two-stage transaction execution (TSTE) and virtual

ADR buffer (VADR). TSTE allows write requests of a transaction to be in both the ADR buffer and an introduced staging SRAM buffer. Log operations and in-place updates are performed in different buffers at different stages to reduce the number of log operations. VADR decouples the ADR buffer's buffering from the number of its reliable draining requests. With VADR, requests in the ADR buffer only issue in-place updates. Additionally, VADR can parallelize ADR-assisted in-place updates with log write operations, increasing the parallelism in NVM writes. Compared with LAD, the evaluation shows that combined with the redo log, TSTE improves the throughput by 28.7% on average. VADR achieves an average of 36.2% improvement in transaction throughput.

3.2 Motivation

3.2.1 Persist requirements of a transaction

A persist requirement is defined as the number of modified cache lines in a transaction. Figure 3.1 shows the distribution of the persist requirement for the benchmarks used in this work. A sub-bar represents the percentage of the range of the number of modified cache lines in a transaction. The persistent requirement for most transactions in all workloads studied in this work is less than 64. For example, the persistent

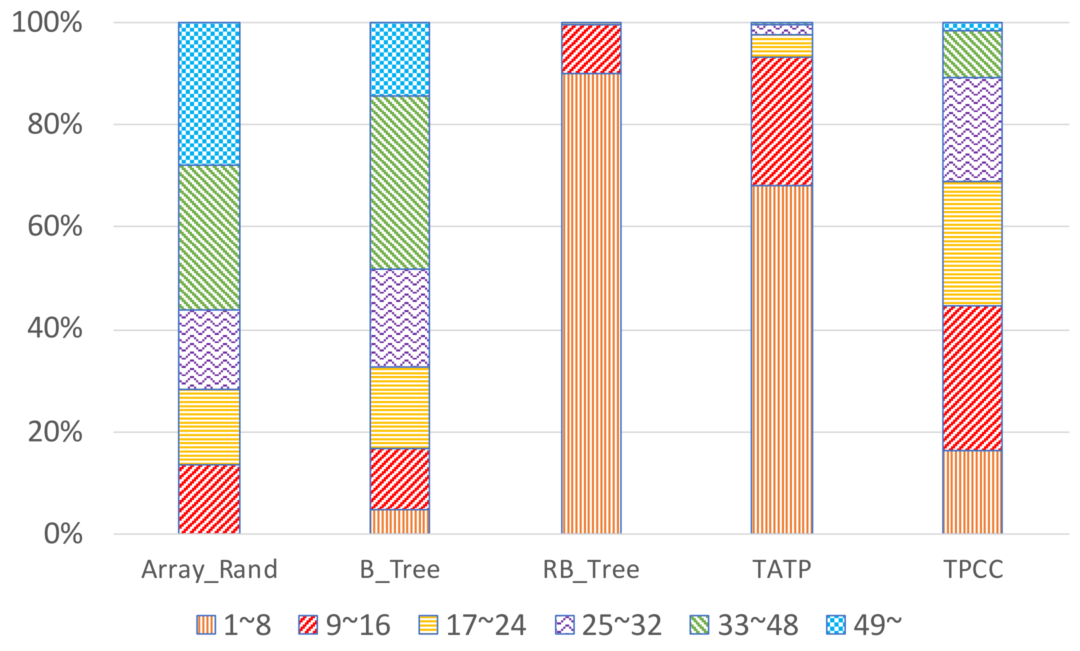


Figure 3.1: Transaction update demand distribution

requirements are less than 32 for the benchmark RB_Tree and TATP. This is consistent with observations made by the recent study [35]. Innovative techniques, such as deduplication [46, 47] and data structure optimization for persistence [48], have been proposed to further reduce the persistent requirement of a transaction. These observations motivate us to leverage the recently introduced ADR to achieve write atomicity without logging.

3.2.2 Unnecessary Logs

Unfortunately, concurrent transactions compete for ADR buffer and make ADR buffer overflows frequently, under a multi-core system. It is often that the update demand

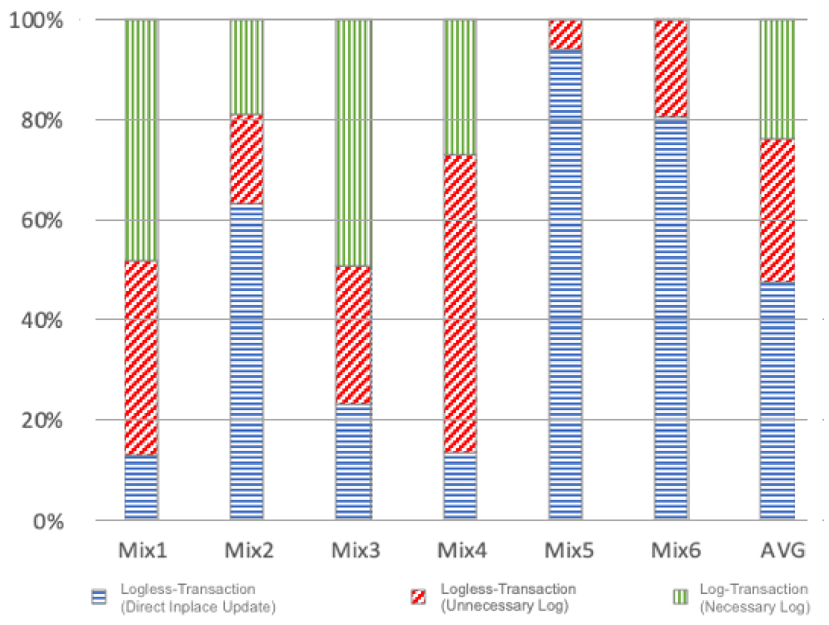


Figure 3.2: Transaction types distribution.

of each individual transaction is smaller than the ADR buffer but the total update demand of all in-flight transactions is larger than the capacity of the ADR buffer. Log operations caused by the overflow of the ADR buffer are unnecessary if these transactions do not issue update requests simultaneously. These log operations, referred to as *unnecessary logs*, consume significant I/O bandwidth and reduce transaction throughput, leading to inferior system performance. Figure 3.2 shows that LAD can cause up to 59.5% unnecessary log transactions in our 4-core workloads listed in Table 3.1. This important observation motivates us to improve efficiency by eliminating unnecessary logging.

We classify each transaction into a log-transaction or a logless-transaction depending on whether this transaction involves log operation or not when it runs individually.

In terms of a transaction's update demand and the capacity of the ADR buffer, a transaction is a logless-transaction if its update demand is less than or equal to the capacity of the ADR buffer, otherwise, a transaction is a log-transaction.

Workload	Benchmarks
Mix1	Array_Random + B_Tree + Array_Random + B_Tree
Mix2	Array_Random + RB_Tree + Array_Random + RB_Tree
Mix3	Array_Random x 4
Mix4	B_Tree + TPCC + B_Tree + TPCC
Mix5	RB_Tree x 4
Mix6	TATP x 4

Table 3.1
Mucltiсore workloads

3.3 Two-Stage Transaction Execution (TSTE)

LAD switches to logging when in-flight transactions take more than 80% of the ADR buffer. However, LAD often generates log operations that are in fact unnecessary. To reduce unnecessary log operations, we propose the Two-Stage Transaction Execution. It is motivated by the observation that write requests to be logged do not have to be stored in the ADR buffer. Therefore, TSTE stores logged requests in the introduced SRAM staging buffer and ADR accommodates more write requests without logging. As a transaction is running, some of its write requests could be stored in the staging buffer and its remaining write requests could be in the ADR buffer. TSTE performs log operations for a transaction's write requests stored in the staging buffer at the first stage and then executes in-place updates for its write requests in the ADR buffer

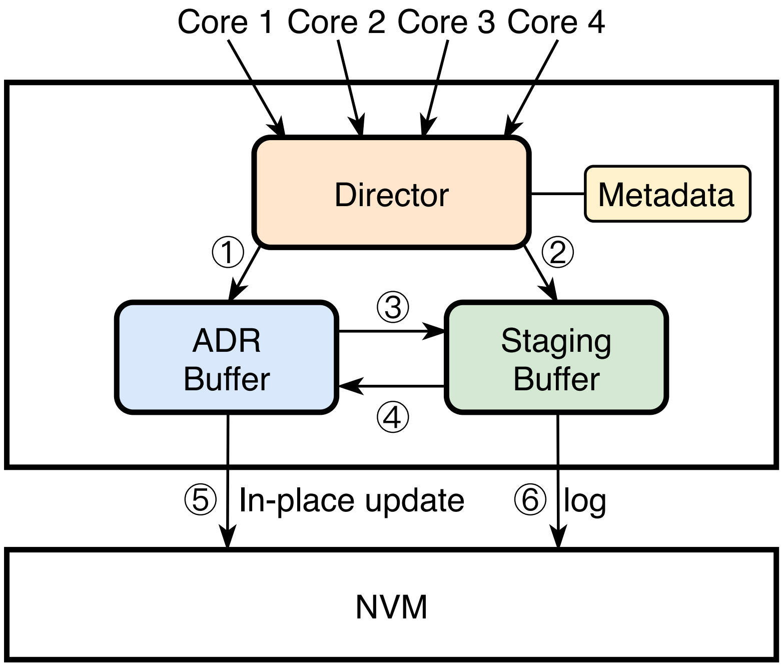


Figure 3.3: TSTE architecture

in the second stage. We will discuss how to direct a transaction's write requests to the ADR buffer and the staging buffer later. After issuing all update requests, the transaction is stalled and becomes an end-transaction, waiting for its commit completion. Being an end-transaction may start in-place update operations for its write requests in the ADR buffer. TSTE commits a transaction immediately after its second stage begins since its remaining write requests are in the ADR buffer.

The proposed hybrid scheme ensures the atomicity of a transaction's update. If a crash takes place during a transaction's first stage, the recovery system reverts all affected transactions to their original states by applying persisted log entries. The recovery system can apply a partial undo log to recover the affected data to their original value, while the recovery system discards partial redo log entries and removes corresponding remapping entries. If a crash occurs during the second stage of a transaction, ADR guarantees that all remaining in-place update requests of this transaction are executed, achieving the update atomicity.

Figure 3.3 shows the TSTE architecture. After receiving a write request from a core, the director sends this request to either the ADR buffer(step ①) or the staging buffer(step ②). On the ADR buffer overflow, TSTE keeps the priority transaction's write requests in the ADR buffer and migrates non-priority transactions' requests to the staging buffer(step ③). When the ADR buffer has sufficient free entries, TSTE promotes an end-transaction to be a secondary one and moves its pending write requests from the staging buffer to the ADR buffer(step ④). While TSTE performs in-place update operations for write requests in the ADR buffer(step ⑤), it executes log operations for write requests in the staging buffer(step ⑥). When a priority transaction depletes the ADR buffer, TSTE directs this transaction's remaining write requests to the staging buffer(step 3).

Each transaction has its metadata to keep tracking the locations of its in-flight write requests in the ADR/staging buffer with the ADR bitmap and the staging bitmap where each entry has 2 bits to represent a request's state and its validity. Assuming the ADR buffer and the staging buffer are 32 entries and 64 entries respectively, one transaction ADR bitmap and the staging bitmap have 32 bits and 64 bits respectively.

The bitmaps overhead is $4 \times (64 \times 2 + 32)$ bits (160 bytes) for a 4-core system since each core only runs one in-flight transaction at a time. We set the staging buffer size to 4KB SRAM. The Cacti [49] estimates its area overhead to be 0.0143 mm^2 with the 22 nm technology.

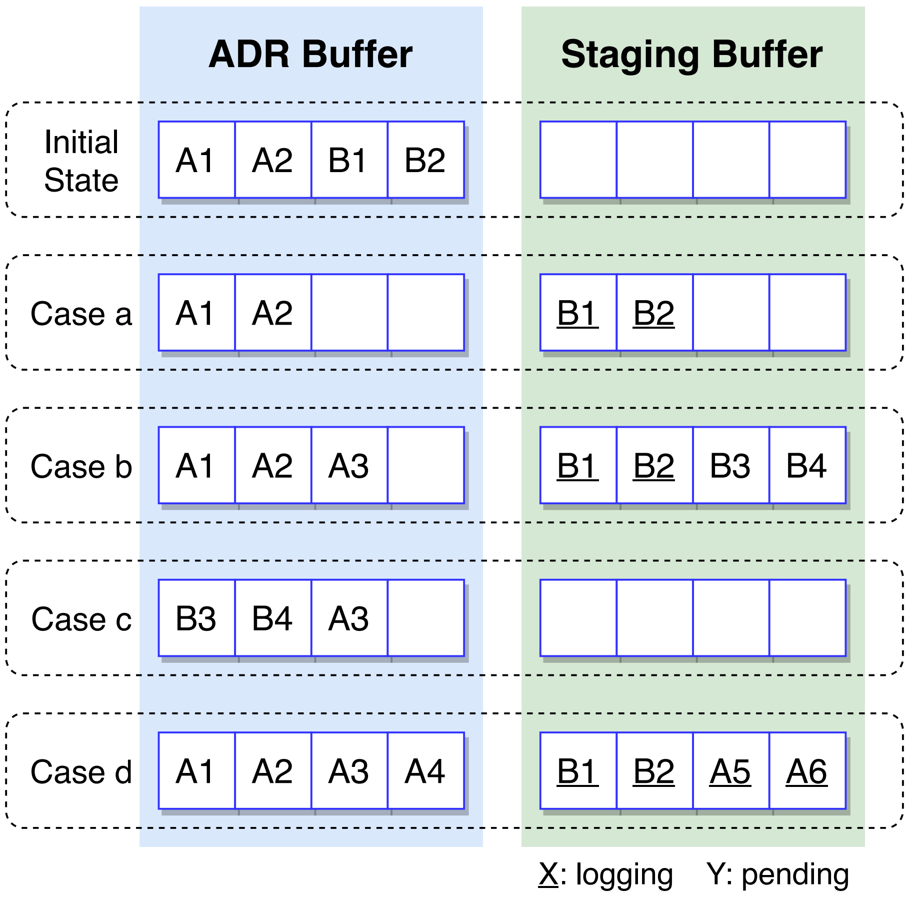


Figure 3.4: Examples of Two-Stage Transaction Execution

Figure 3.4 uses examples to illustrate how TSTE directs write requests for concurrently running transactions to either the ADR buffer or the staging buffer. Assume both the ADR buffer and the staging buffer can hold 4 entries, and transactions A and B cause the ADR buffer overflow (see the initial state in Figure 3.4).

† Case a. If transaction A is selected as a priority one, write requests of transaction A are kept in the ADR buffer. Write requests of all non-priority transactions are moved to the staging buffer, and then log operations are issued for them.

† Case b. Since transaction A and B are priority transaction and non-priority one respectively; the request $A3$ is directed to the ADR buffer, and the requests $B3$ and $B4$ are stored in the staging buffer. TSTE limits the maximal degree of logging parallelism for each non-priority transaction. For instance, Figure 3.4 case b shows logging operations of $B1$ and $B2$ are running, but $B3$ and $B4$ are pending.

† Case c. After a non-priority transaction eventually becomes an end-transaction, it will be promoted to be a secondary one if its pending write requests can be accommodated to the ADR buffer. For example, Figure 3.4's case c shows the non-priority transaction B becomes a secondary one and its pending requests, including $B3$ and $B4$, are migrated to the ADR buffer. The completion of its issued log operations leads to committing this transaction; simultaneously their in-placed updates are initiated in parallel.

† Case d. If the priority transaction, tx, makes the ADR buffer overflow, its remaining update requests are directed to the staging buffer. After that, the TSTE initiates log operations for tx's update requests in the staging buffer, which is in the first stage of execution. After tx's first stage is done, TSTE commits it, and then starts in-place updates for tx's write requests in the ADR

buffer, avoiding writing logging for them. Case d in Figure 3.4 shows that the priority transaction A causes the ADR buffer overflow, and its $A5$ and $A6$ perform logging in the staging buffer. After $A5$ and $A6$ are done, TSTE commits transaction A, flushing A's requests in the ADR buffer.

When a transaction $TX1$ issues its first write request, we need to decide its location: either the ADR buffer or the staging buffer. If ADR is empty, $TX1$'s first request is directed to ADR, which is a priority transaction. If ADR buffers write requests of a priority transaction or a secondary transaction, the transaction $TX1$'s write requests are directed to the staging buffer, as a non-priority transaction. On completion of a write request in the ADR buffer, TSTE attempts to promote a non-priority end-transaction, tx, to be a secondary if the ADR buffer has sufficient free space to accommodate tx's pending requests in the staging buffer.

3.4 Virtualization of the ADR Buffer (VADR)

In Section 3.3, we propose to allow a priority transaction to use ADR exclusively when the ADR buffer overflows. This reduces unnecessary log operations for logless-transactions. However, a logless priority transaction's write request is held in ADR until its all update requests arrive, leading to inferior performance. The worse is that a log priority transaction's dirty cachelines can be written after its first stage is done.

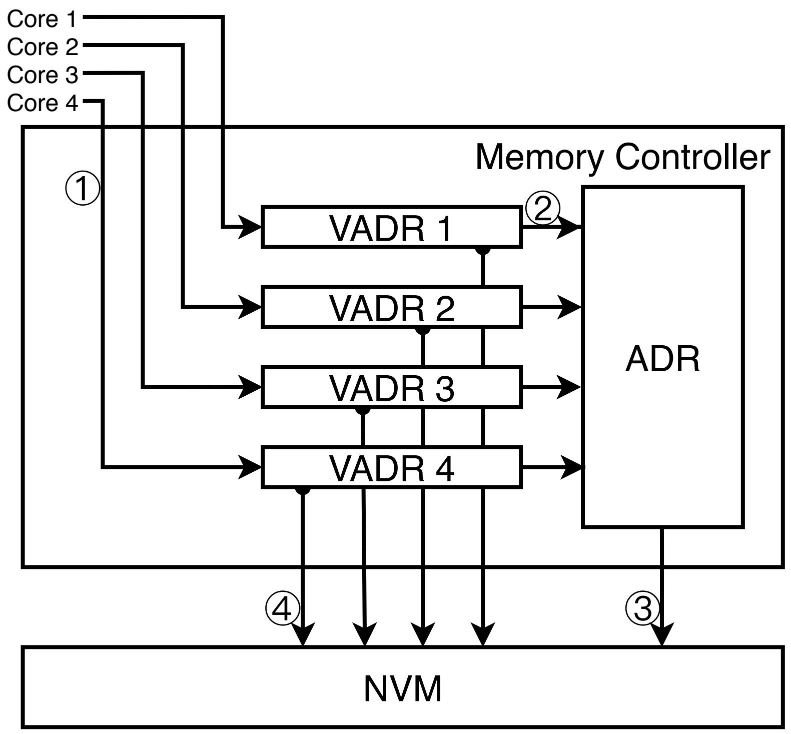


Figure 3.5: Virtual ADR buffer (VADR) in the memory controller

To address this issue, we propose to decouple the ADR's buffering function from its durable writing function. We only send the write requests of one transaction to the ADR buffer after all of its requests arrive, and then these write requests can be immediately issued. To achieve this goal, we propose to add a private SRAM buffer (called Virtual ADR buffer, *VADR*) in the memory controller for each core, as shown in Figure 3.5. Each VADR has the same capacity as the ADR buffer and it predicts whether an in-flight transaction is logless. Each core sends write requests to its private VADR (step ①). Since VADR has the same size as the ADR buffer, VADR can accept all write requests of a logless-transaction without overflow. After VADR collects all dirty cache lines of a logless-transaction, the memory controller moves them to the ADR buffer (step ②) and then writes them to NVM (step ③). When VADR overflows, the memory controller marks the corresponding in-flight transaction as a log-transaction, and starts to directly log this transaction to NVM (step ④). The hardware overhead is negligible. Assuming the ADR buffer has 32 entries, VADRs for a 4-core processor consume only 8KB SRAM in the memory controller. The Cacti [49] estimates its area overhead to be 0.0147 mm^2 with the 22 nm technology.

Each VADR collects all updated write requests of an in-flight logless-transaction, without stalling the execution of its corresponding core. However, this transaction cannot be committed until all these write requests have been moved to ADR. This is because an uncommitted transaction in ADR can survive a crash and flushing these write requests to NVM violates the update atomicity of the transaction. For the same reason, ADR initiates in-place updates to NVM only after ADR has collected all dirty write requests from a transaction. To facilitate this process, an ADR buffer entry is marked as flushable after ADR collects all write requests of the corresponding transaction, and only flushable write requests can be drained to NVM.

Multiple logless-transactions can consume ADR simultaneously. When there is sufficient free space in ADR, the memory controller admits all requests of a logless transaction in VADR to ADR. In this design, we deploy a simple admission policy: First-Come-First-Serve (FCFS). We have also evaluated a complicated policy, called Minimal-Update-Demand-First (MUDF). However, the performance gain is minimal.

Thus, FCFS is chosen due to its simplicity and fairness.

VADR can guarantee crash consistency for transactions. The logging mechanism ensures crash consistency for log-transactions. A logless-transaction has no update requests persisted to NVM if a crash occurs before all write requests are admitted to the ADR buffer. If a logless transaction has been admitted to ADR, the update-atomicity of this transaction can survive a crash.

3.5 Redo Logging for ADR Depletion

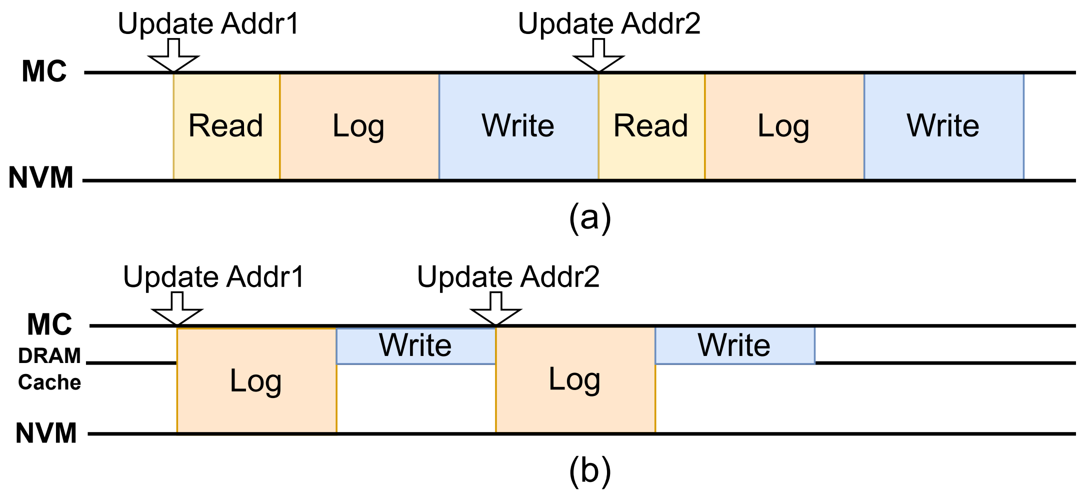


Figure 3.6: Undo logging and Redo logging. (a)Undo log operation, (b) Redo log operation.

Upon ADR buffer overflow, LAD resorts to undo-log. However, the speed to commit an undo-logged transaction is slower than that of a redo-logged. Figure 3.6(a) shows undo logging needs to read the old value of the updated cache lines from NVM, then write log entries to NVM, and finally write the dirty cache lines to NVM before this transaction commits. On the other hand, Figure 3.6(b) shows that redo logging can commit a transaction by only writing the dirty cache lines to the log in NVM. However, redo logging needs to consult the remapping table for reading. To reduce remapping overhead, we use a DRAM cache to hold the latest cache lines, which log entries in the NVM log region, in a way similar to REDU [6]. An NVM read request looks up the DRAM cache, which avoids scanning the logs. Therefore, both TSTE and VADR use redo logging to increase the transaction commit speed. This design has the same hardware overhead as REDU [6].

3.6 Evaluation

3.6.1 Experiment Setup

The proposed designs are implemented and evaluated via ChampSim [50] and DRAMSim2 [51]. ChampSim is an Intel PIN [52] based simulator that models out-of-order micro-architecture at the cycle level with detailed memory access behaviors, such as LSQ's memory dependence, Miss Status Holding Registers (MSHR) for TLB and caches. These are not modeled in McSimA+ [53], which is used in previous NVM crash consistency researches [3, 54, 55, 56]. To accurately model NVM access, the

Cores	OoO core @2GHz, 192 ROB entries, 48 store queue entries
L1 Inst. Cache	32KB, 64B lines, 8-way, 1 cycle access latency, 8 MSHR entries
L1 Data Cache	32KB, 64B lines, 8-way 3 cycle access latency, 8 MSHR entries
L2 Data Cache	256KB, 64B lines, 8-way 12 cycles access latency, 16 MSHR entries
LLC Cache	2MB per core, 64B lines, 16-way 30 cycles access latency, 32 MSHR entries
DRAM cache	access latency 100 cycles, 32MB
Memory Controller	1 controller, 32 ADR buffer entries
NVM Access Latency	300 ns Write latency, 48 ns Read latency

Table 3.2
System parameters

Array_Random	Randomly insert/delete Elements in an array
B_Tree	Insert/delete nodes in a B_tree
RB Tree	Insert/delete nodes in a Red_Black_tree
TATP [57]	Update locations
TPCC [58]	Add new orders

Table 3.3
Benchmarks used for evaluation

cycle-level DRAMSim2 is incorporated into ChampSim. We have extended ChampSim to support Intel persistent memory instructions *clwb* and *sfence*. The default ADR buffer capacity is 32, which is similar to the ADR configurations used in previous studies [2, 56]. The configuration and parameters of the processor and memory system used in our experiments are listed in Table 3.2.

Table 3.3 lists benchmarks used for evaluation in this study. Array_Random, B_tree, and RB_tree are micro-benchmarks. TPCC [58] is an online transaction processing workload requiring ACID guarantees. TATP is an OLTP workload that simulates a caller location system [57]. Figure 3.1 shows the transaction update demand distribution for each benchmark. As we have mentioned in section 3.2.1, the persistent requirement for most transactions in all workloads studied in this work is less than 64. Since this work targets multicore workloads, we construct 4-core workloads shown in Table 3.4. In the multicore simulation, a core re-executes its workload if it finishes earlier than other cores. The simulation is done until the slowest core completes the execution of its workload.

We evaluated the following designs.

- † LAD: It implements LAD according to Ref. [7]. A logless-transaction can directly perform in-place updates after all updated cache lines of this transaction are collected by the ADR buffer. Otherwise, this transaction is required to write undo log entries to NVM before performing in-place updates to NVM. Due to simulating a single memory controller, we do not implement the distributed commit protocol for multiple distributed memory controllers. LAD serves as the baseline for comparisons in this study.

- † LAD with redo log (LAD-RdLog): LAD is enhanced by redo log with the DRAM cache [6]. It also models the latency for DRAM cache accesses for each NVM read request.

- † TSTE with redo log (TSTE-RdLog): We evaluate TSTE with redo log and the

Workload	Benchmarks
Mix1	Array_Random + B_Tree + Array_Random + B_Tree
Mix2	Array_Random + RB_Tree + Array_Random + RB_Tree
Mix3	Array_Random x 4
Mix4	B_Tree + TPCC + B_Tree + TPCC
Mix5	RB_Tree x 4
Mix6	TATP x 4

Table 3.4
Multicore workloads

DRAM cache. Its staging buffer has 64 entries, and we limit the maximal log parallelism to 8 for each transaction.

† Virtual ADR buffers with redo log and DRAM cache(VADR-RdLog): We evaluate virtual ADR buffers with redo log, and the DRAM cache latency is simulated for each NVM read request.

To evaluate different designs, we use the following metrics.

† Transaction throughput: It is defined as the number of committed transactions divided by the execution time. It is the transaction system performance indicator. The higher the throughput, the better the system performance is.

† Average Log traffic: It is defined as the number of cache lines read and written for logging divided by the total number of simulated transactions. A large value of log traffic indicates large NVM bandwidth consumption.

3.6.2 Removal of Unnecessary Log Operations

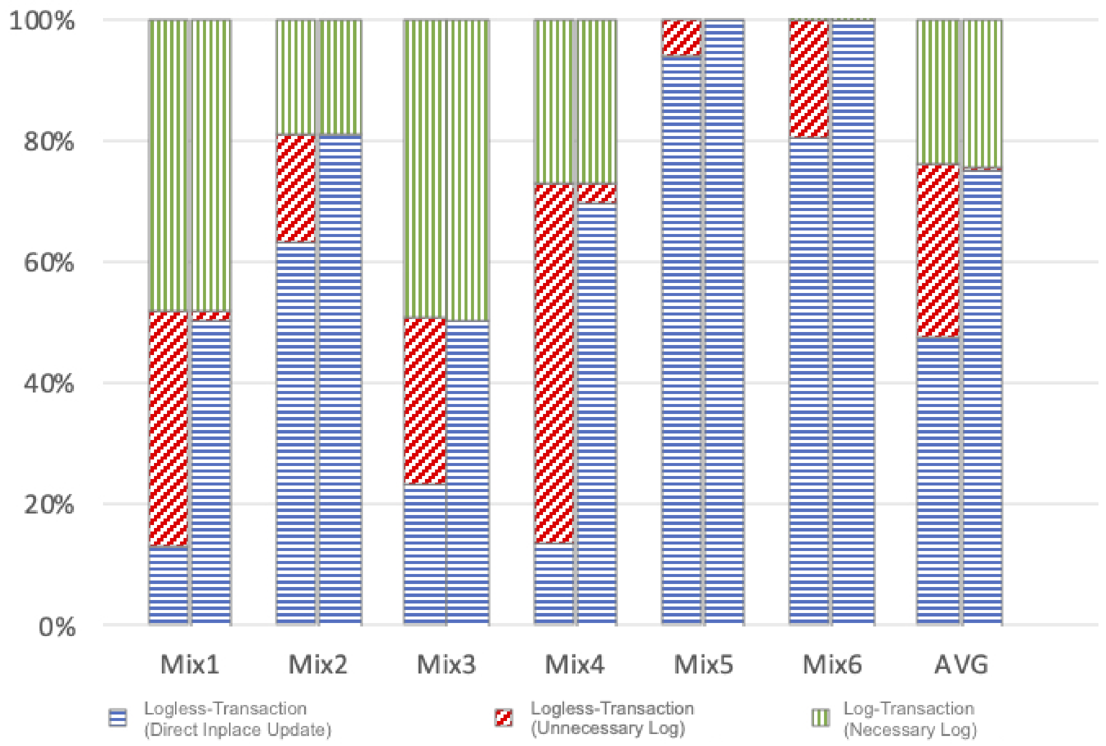


Figure 3.7: Transaction execution type distribution

A transaction execution can be classified into three categories: direct in-place update, unnecessary-log, and necessary-log. A direct in-place update transaction can fit in the ADR buffer, and perform direct in-place updates to NVM without logging. While an unnecessary-log transaction's write set size is smaller than the ADR buffer size but is involved with logging, a necessary-log transaction's write set size is larger than the ADR buffer size. Figure 3.7 shows the distribution of transaction execution types for LAD-RdLog and VADR-RdLog, which are indicated by the left bar and right bar respectively for each workload. First, LAD-RdLog causes a significant percentage of unnecessary-log transactions in some workloads. For example, the workload Mix4's unnecessary-log transactions reaches 59.5%. On average, these workloads' unnecessary-log transactions account for on average 28.3%. Second, VADR-RdLog can successfully remove unnecessary-log transactions and convert them to direct in-place update transactions. VADR can allow logless-transaction to exclusively access the ADR buffer even when multiple transactions compete for the ADR buffer. Therefore, VADR can efficiently remove unnecessary-log transactions.

3.6.3 Transaction Throughput Improvements

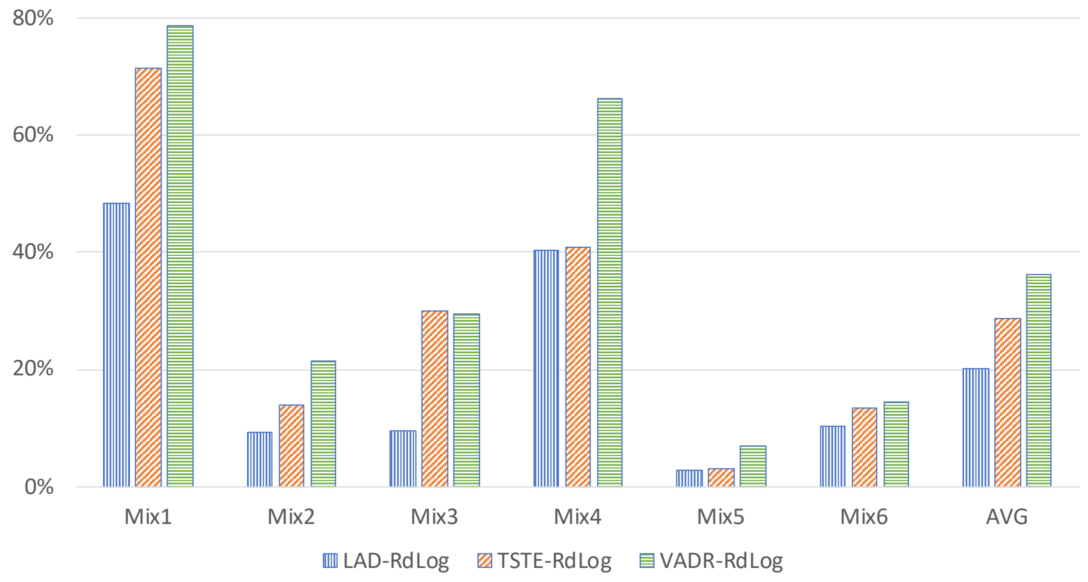


Figure 3.8: Transaction throughput improvement over LAD

Figure 3.8 shows the LAD-RdLog, TSTE-RdLog, and VADR-RdLog transaction throughput improvement over LAD. First, VADR-RdLog achieves the highest throughput improvements. For example, the workload Mix1 boosts the throughput to 78.6% compared with LAD. On average, VADR-RdLog increases the throughput by 36.2%. The VADR buffer can prevent log-transaction from consuming the ADR buffer and only allows logless transactions to access the ADR buffer, almost avoiding log operations caused by the ADR resource contention. Second, TSTE-RdLog improves the throughput by 28.7% on average. TSTE-RdLog not only removes log operations for write requests of priority transactions in the ADR buffer but also avoids log operations for eligible pending write requests in the staging buffer. This log operation reduction leads to throughput improvement. Third, LAD-RdLog boosts the throughput by 20.1% on average. This higher transaction throughput comes from the faster transaction commit speed of the redo logging. Fourth, the efficiencies of TSTE-RdLog and VADR-RdLog depend on the transaction execution type distribution of workloads. For example, in some workloads, such as Mix1 and Mix4, which have significant percentages of unnecessary-log transactions (see Figure 3.7), our design can achieve significant performance improvement (see Figure 3.8). If the workloads, such as Mix5 and Mix6, lack unnecessary-log transactions, our designs have fewer opportunities to reduce log operations, leading to less performance gain.

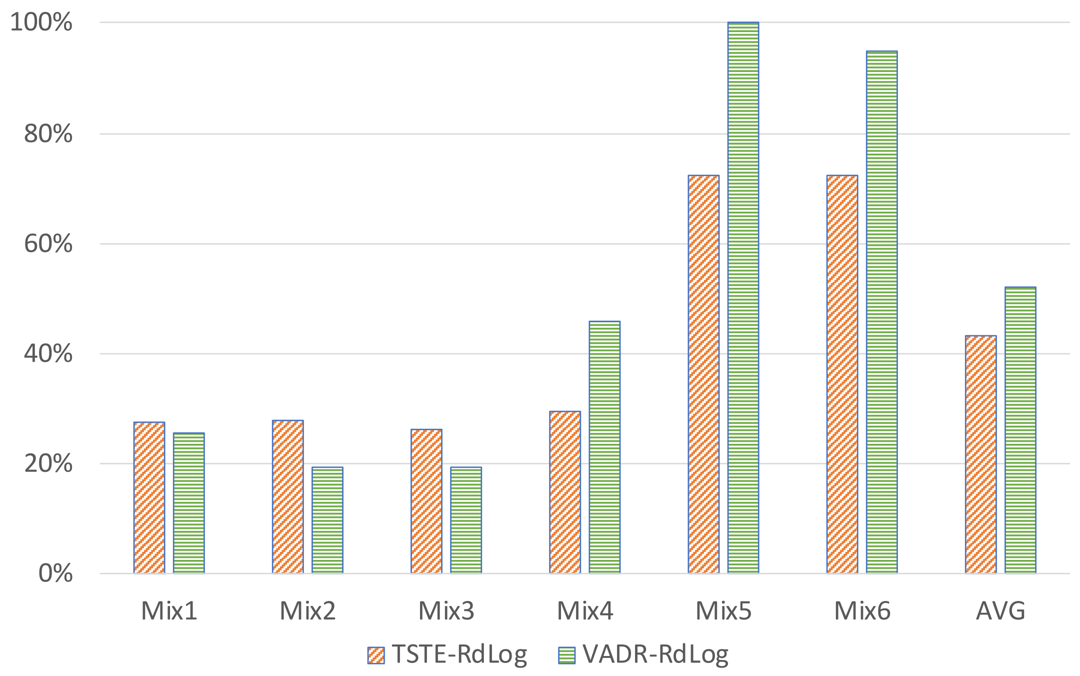


Figure 3.9: Log operations reduction

3.6.4 Log Operations Reduction

Figure 3.9 indicates that TSTE-RdLog and VADR-RdLog can effectively reduce log operations, compared with LAD-RdLog. First, VADR-RdLog is more efficient than TSTE-RdLog to reduce log operations under our workloads. For example, on average, VADR-RdLog generates 9.7% fewer log operations than TSTE-RdLog. However, TSTE-RdLog performs fewer log operations than VADR-RdLog under workloads Mix1, Mix2, and Mix3. This can be explained by the transaction execution type distribution shown in Figure 3.7. Mix1 and Mix3 have more log-transactions than the other workloads. VADR-RdLog fails to reduce log operations for those log-transactions since each write request of such transactions needs a log operation. However, TSTE can avoid log operations for log-transactions' eligible pending requests, by migrating these write requests to the ADR buffer. On the other hand, VADR-RdLog works more efficiently than TSTE-RdLog for workloads dominated by logless-transactions, such as Mix5 and Mix6. This also can explain why Mix5 and Mix6 reduce significant portions of log operations but boost less throughput, as shown in Figure 3.8.

3.6.5 Sensitivity of ADR Buffer Capacity

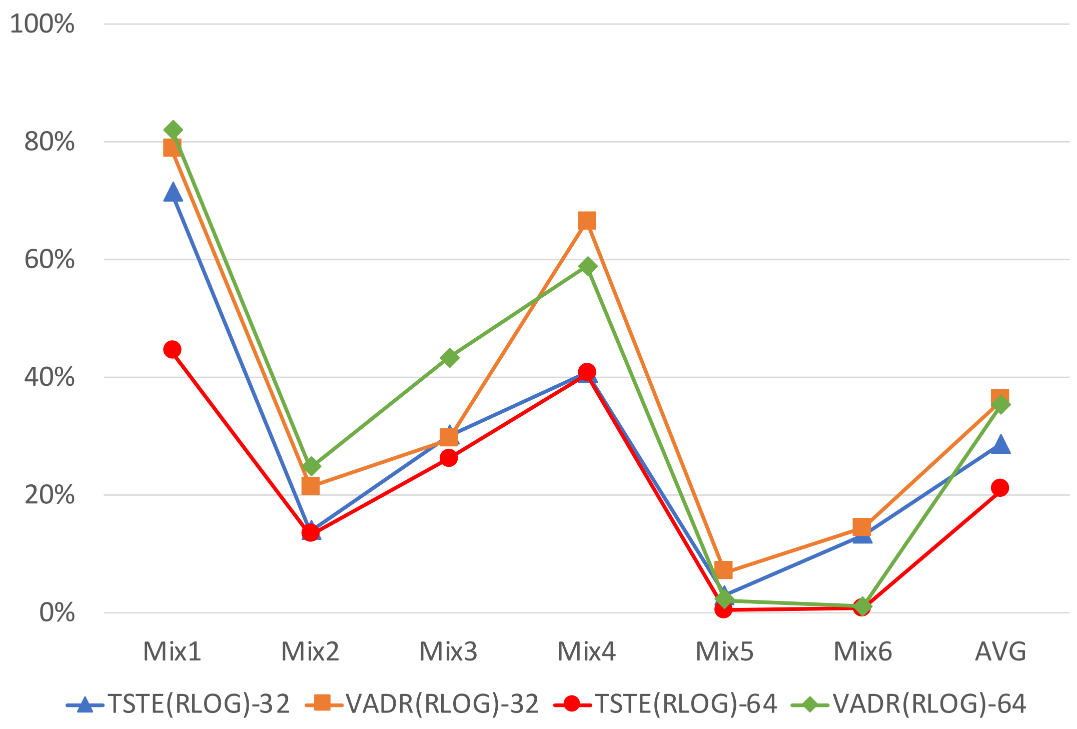


Figure 3.10: Throughput improvements over LAD with various ADR buffer sizes

In this subsection, we study the transaction throughput sensitivity to different ADR buffer sizes. Figure 3.10 shows the throughput improvement of TSTE-RdLog and VADR-RdLog over LAD when the ADR buffer has 32 and 64 entries. On average, TSTE-RdLog improves the throughput by 28.7% and 21.0% with 32 and 64 entries, respectively. In VADR-RdLog, the average throughput improvement is 36.2% and 35.4%, respectively for 32 and 64 entries. Both TSTE-RdLog and VADR-RdLog show the same trend that the performance improvement diminishes as the size of the ADR buffer increases. This is because a larger ADR buffer helps mitigate the ADR resource contention, and reduces the number of logless-transactions and log-transactions under the same workload. However, VADR-RdLog works better for Mix1 and Mix3 as the ADR buffer size increases from 32 to 64. This is because, with a buffer size of 32, more log-transactions are generated under these two workloads, and some log-transactions become logless-transaction if the size of the ADR buffer is increased to 64, resulting in more reduction of log operations. As the number of cores in a processor keeps increasing and it is hard to scale up the ADR buffer size, we expect that efficient ADR buffer management schemes are still needed to mitigate the ADR buffer contention.

3.7 Summary

The NVM systems require log schemes to ensure crash consistency, introducing severe performance overhead. Recently, LAD was proposed to remove log operations for those transactions whose total amount of simultaneously updated cache lines is smaller than the ADR buffer, without sacrificing the crash consistency of such transactions. When concurrently executing transactions compete for the ADR buffer, they tend to deplete the ADR buffer quickly, and hence log operations have to be performed. We observe that there are a significant number of log operations that could be eliminated if a transaction runs alone. To remove these unnecessary log operations, this work presents a new transaction execution scheme that places write requests of a transaction in both the ADR buffer and the SRAM staging buffer. Our scheme, called two-stage transaction execution (TSTE), performs log operations only for write requests of a transaction in the SRAM staging buffer and executes in-place update operations for this transaction's write requests in the ADR buffer. To further improve ADR resource utilization, this work also proposes virtual ADR buffers to decouple the buffering from the ADR's reliable writing data. The proposed virtual ADR buffers only allow logless operations to access ADR resources. Additionally, this work proposes to adopt a redo log with a DRAM cache to speed up the transaction commit speed. Our evaluation results demonstrate that our proposed schemes can efficiently reduce log operations by up to 94.9% and improve the transaction throughput by up to 78.6%.

Chapter 4

Accelerate Hardware Logging for Efficient Crash Consistency in NVM

4.1 Overview

This work reveals two common factors that contribute to the inefficiency of logging: (1) load imbalance among memory banks, and (2) constraints of intra-record ordering. Overloaded memory banks may significantly prolong the waiting time of log requests targeting these banks. To address this issue, we propose a novel log entry allocation scheme (LALEA) that reshapes the traffic distribution over NVM banks. In addition, the intra-record ordering between a header and its log entries decreases the degree of parallelism in log operations. We design a log metadata buffering scheme (BLOM) that eliminates the intra-record ordering constraints. These two proposed log optimizations are general and can be applied to many existing designs. Our evaluation shows that LALEA can achieve 54.04% higher transaction throughput on average, compared to the state-of-the-art design REDU. BLOM can achieve 17.16% throughput improvement over REDU on average. LALEA and BLOM together can boot throughput by 56.62%.

4.2 Motivation

This work is motivated by the observation that the inefficiency of logging is mainly caused by two factors: (1) imbalanced workload over underlying NVM banks, and (2) intra-log record persistence ordering. The imbalanced workload over banks leads to the high latency of log request persistence if a log request is served by the bank with a large number of pending memory requests. However, it is challenging to solve the bank workload imbalance. Reference locality tends to make requests cluster around a few banks, leading to workload imbalance over banks. Conventional log entry allocation schemes blindly allocate a physical address to a log request, further deteriorating this issue. The intra-log record persistence ordering requires a log record's metadata can be persisted only after all log entries of this record have been persisted to NVM. The persisting serialization between a log record metadata and its entries aggravates transaction throughput.

4.3 Log Entry Allocation Scheme (LALEA)

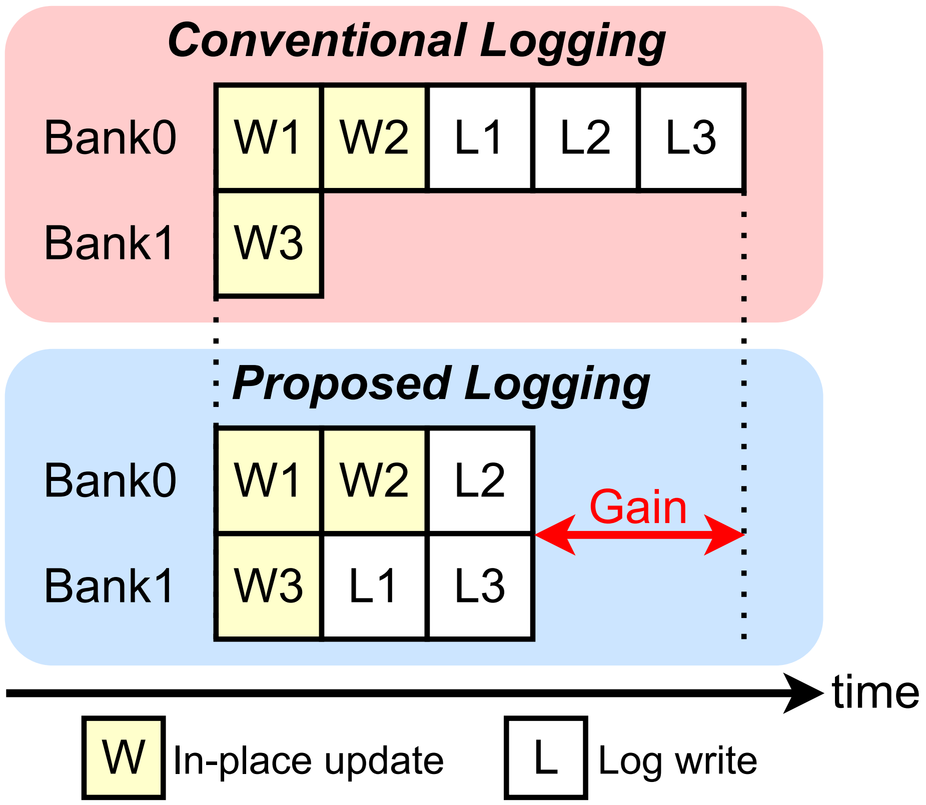


Figure 4.1: Compare LALEA with conventional logging

The log operation execution of a transaction could be on the critical path. For example, redo logging can commit a pending transaction only after all log requests are persisted. The slow execution of these log operations dictates the transaction commit latency. Log requests in a bank with a larger number of pending requests suffer from a higher persistence latency. Therefore, the slow log operation execution could be caused by the imbalanced workloads over memory banks. The uneven workload distribution over banks could be caused by the inherent locality of workloads [59, 60], which makes in-place updates occur in one or a few banks. Furthermore, all prior designs prefer to allocate a well-known contiguous NVM space for log entries, to facilitate log management and recovery. This workload-agnostic log allocation scheme deteriorates the issue of bank workload imbalance.

As discussed above, a balanced workload among banks can lead to low latency for log requests. To balance workload over banks, we propose a novel strategy that performs load-aware log entry allocation (LALEA) and effectively balances log write requests over NVM banks. The main idea of LALEA is to allocate a log entry address according to the underlying banks' current pending operations. Specifically, LALEA allocates an incoming log write operation a free block in the log region whose bank has the smallest number of pending requests. Such adaptive log entry allocation can mitigate the workload imbalance of memory banks caused by transactions' in-place update operations. The balanced workload brings two performance benefits: (1) reduced log request persistence latency and (2) reduced memory latency for non-log requests. The evenly distributed bank traffic can decrease memory latency for both memory read requests and memory write requests issued in the transaction execution stage.

Figure 4.1 compares our LALEA with conventional log management by using a simple example. For simplicity, assume that the system has only two banks, $Bank0$ and $Bank1$, and they have two pending in-place update requests and one in-place update request, respectively. Conventional log management schemes sequentially allocate free memory blocks in $Bank0$ to log requests L1, L2, and L3, ignoring the current workload on these banks. This leads to a longer service time for the log request L3. However, our proposed method can allocate a log entry to the bank with a minimal number of pending operations, and these three log entries are allocated to two banks. LALEA can balance memory requests over banks and reduce the completion time of log requests, especially for L3, decreasing the transaction commit latency.

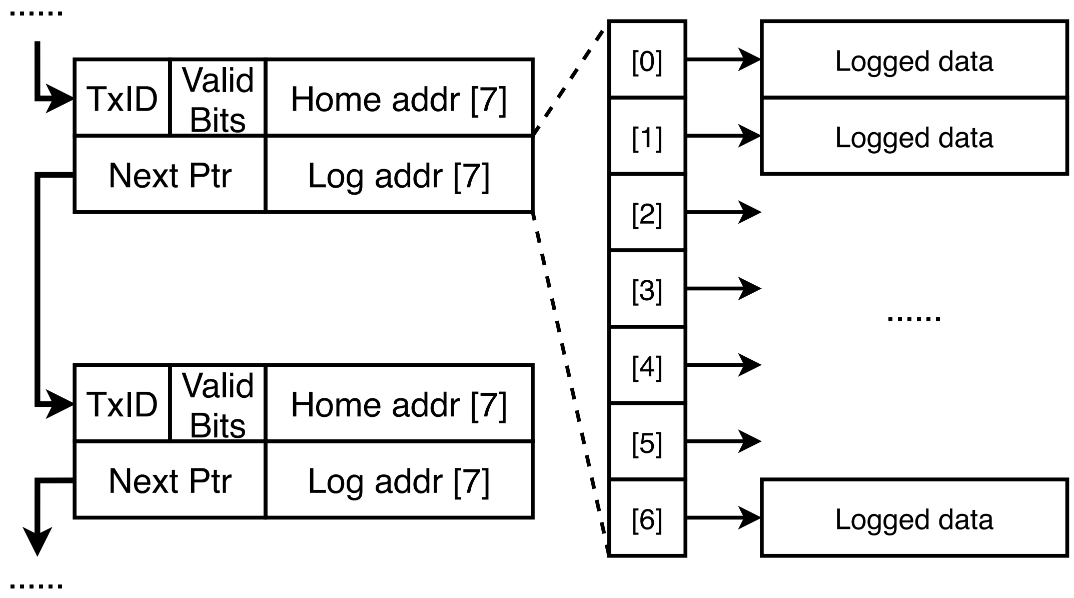


Figure 4.2: LALEA log record organization

The log organization scheme, such as LEC, requires that all log entries of a log record are sequentially stored in the log region so that they can be accessed without storing their addresses. However, the lack of the addresses of log entries makes it impossible for LELEA to access non-sequentially stored log entries of a log record. To address this issue, we enhance a log recorder header that includes an extra 64B metadata block to store addresses for seven log entries stored in different banks, as shown in Figure 4.2. The field $Next\_Ptr$ in the second metadata block points to the log header of the next log record belonging to a transaction. The $Next\_Ptr$ is NULL if its log record is the last one in a transaction. The $Valid\_Bits$ in the header indicates the count of valid log entries in its log record. The two-64B header is allocated to the same bank that has a minimal number of pending requests. Although LALEA writes extra log metadata to NVM, the introduced performance overhead is shadowed by the performance improvement brought by LALEA, which is confirmed by our experimental results.

LALEA requires that each bank has a managed log region to store log entries. Two contiguous 64B data blocks of a LALEA header must be stored in the same bank. However, two contiguous free log entries cannot be guaranteed to be found in the free block list quickly. To solve this issue, we divide a bank's log region into the log entry region and the header region, and their allocation entries are 64B and 128B, respectively. The free log entry (FLE) FIFO and the free header (FH) FIFO indicate the free blocks to be allocated. When an entry is released, its address is appended to its related FLE FIFO. These two FIFOs only maintain addresses of free blocks.

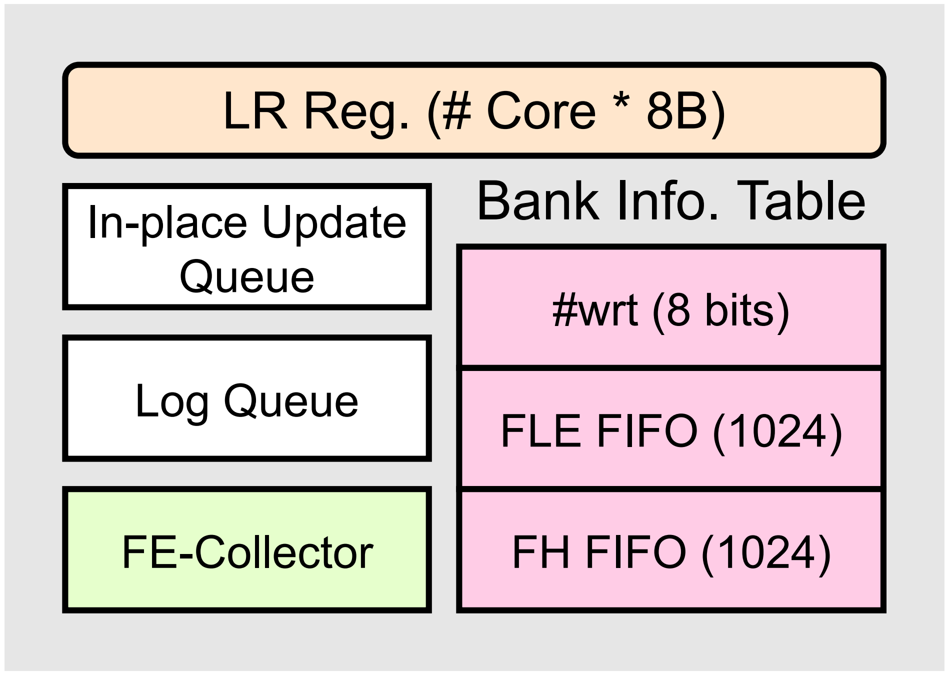


Figure 4.3: LALEA controller

They are not required to be persistent in that they can be rebuilt during the crash recovery process.

Figure 4.3 shows the LALEA controller. It contains a bank information table, log record registers (LR), a free entries collector (FE), a log queue, and an in-place update queue. Each entry of the bank information table includes a write operation counter, an FLE FIFO, and an FH FIFO. An 8-bit operation counter indicates the number of pending memory operations in the bank. Upon a log entry or a header allocation request, the controller first identifies the bank with the minimal operation counter and then grabs a free block address from the corresponding FIFO in the bank. An FLE FIFO and an FH FIFO contain 1024 and 256 free block addresses, respectively. A free block address only includes a row address and a partial column address. Assuming a row address and a column address have 16 and 10 bits respectively, a log entry block address and a header address have 20 bits and 19 bits. Therefore, an FLE FIFO and an FH FIFO require 2.5 KB and 2.375 KB, respectively. A CPU core has an 8B LR register that points to the header of the first log record whose transaction's log records have been persisted to NVM. The LR register is non-volatile in that persisted log entries of a CPU core are accessed through its LR register.

In case of a crash, the recovery module can revert the system to a consistent state, by accessing log entries persisted to NVM. The non-volatile LR register maintains the address to the chain of headers for these persisted log data blocks. The *Next_Ptr* in a log record header links log records stored in NVM, forming the chain. Following this chain, we can walk through these persisted log record headers and apply each logged data to the home region, to make the system state back to a consistent state.

4.4 Log Metadata Buffering Scheme (BLOM)

Conventional log organization is inefficient in terms of log writing throughput. The intra-record ordering constrains the degree of parallelism in log writes. Additionally, the low utilization of the recorder header wastes precious NVM write bandwidth when the corresponding transaction has a few write requests, which is common in NVM workloads [35].

To address these limitations of conventional log organization, we propose to buffer log metadata in ADR buffer, which is referred to as BLOM. The main idea is that log metadata blocks are buffered in the ADR buffer until they are not needed. Log metadata blocks can be discarded when all in-place updates of their corresponding transaction are completed. If there is no crash, the buffering log metadata blocks are not written to the NVM, improving the system performance and enhancing the NVM's lifetime. Upon a crash, ADR persists the buffered log metadata blocks to NVM. When the system reboots, the recovery module can recover the system to a consistent state, following the conventional recovery procedure. When the ADR buffer is used up, the memory controller writes the buffered metadata block of a selected transaction to NVM.

Figure 4.4 compares our proposed metadata buffering and the conventional redo logging. The conventional redo logging initiates the persisting of transaction $TX1$'s header at the time point $t2$ when the log entries are persisted. The header write operation is finished at time $t3$. The intra-recorder ordering requires the serialization of the persisting log data blocks and the header, and the stall of the CPU core execution until time $t3$. Our proposed log metadata buffering eliminates the ordering constraint by buffering the metadata block in the ADR buffer. Furthermore, this

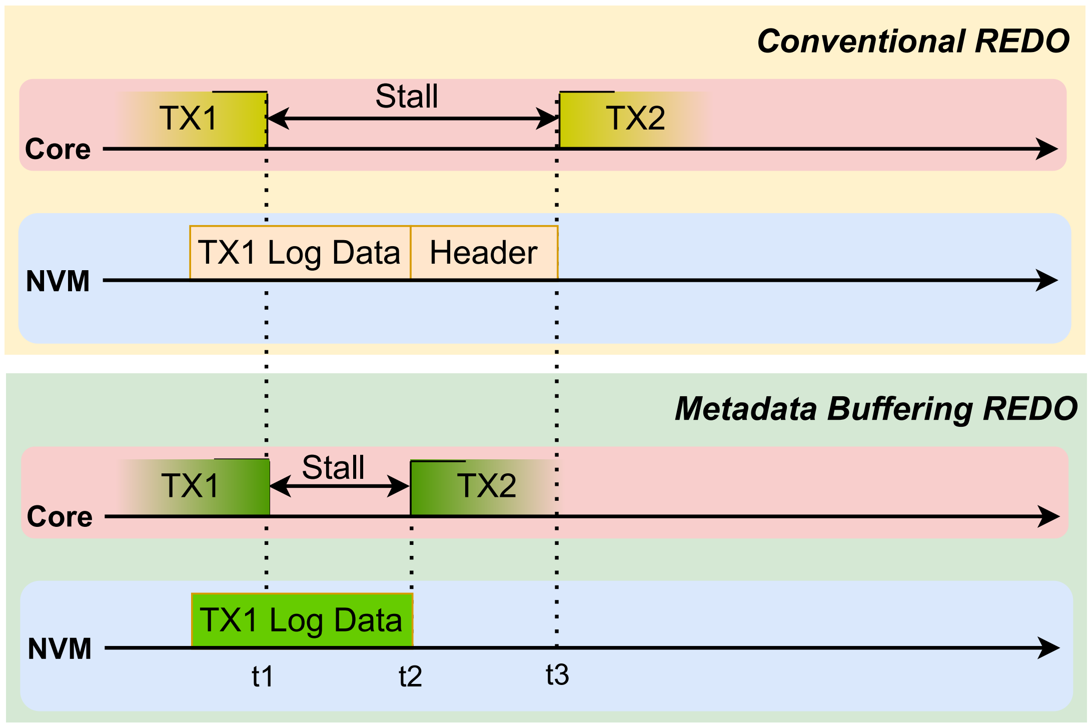


Figure 4.4: Compare the redo logging and our proposed log metadata buffering

metadata buffering avoids writing the metadata block to NVM. Therefore, the metadata buffering allows transaction $TX2$ to execute at time $t2$, reducing the CPU core stalling time.

Our proposed metadata buffering is inspired by LAD [7]. However, our design differs from LAD in the following aspects. First, while LAD buffers all update requests of a transaction, our design only buffers log metadata blocks. LAD degrades to the conventional logging scheme when the ADR buffer overflows. This is a common case when multiple transactions run concurrently and compete for the limited ADR buffer. ADR buffer overflow happens infrequently in our design as it only buffers metadata blocks. In addition, our design writes a smaller number of blocks than LAD upon an ADR buffer overflow. An overflow forces LAD to write log blocks for all update requests in a transaction, while our design only writes the minimal number of buffered metadata blocks.

4.5 Evaluation

4.5.1 Experiment Setup

Cores	4 OoO core @2GHz,192 ROB entries, 48 STQ entries
TLB	L1: 6 sets, 4 ways; L2: 128 sets, 12 ways
L1 I/D Cache	private, 32KB, 2 cycles, 8 ways 8 log buffer entries
L2D Cache	private, 256KB, 8 cycles, 8 ways
LLC	8MB, 25 cycles, 16 ways
Memory Controller	1 channel, 1 rank, 8 banks, 8GB NVM 16 ADR Buffer entries [37], 32 Log Queue entries
NVM Access Latency	300(48) ns write(read) [46, 61]

Table 4.1
System parameters

The proposed designs are implemented and evaluated by using ChampSim [50] with DRAMSim2 [51]. ChampSim is an Intel PIN [52] based simulator that models out-of-order micro-architecture at cycle level with detailed memory access behaviors, including TLB, LSQ memory dependence, and MSHR. To accurately model NVM accesses, the cycle-level memory simulator DRAMSim2 is integrated with ChampSim. We enhance ChampSim to support *tx_begin*, and *tx_end*. The configurations of the processor and memory system used in our experiments are listed in Table 4.1. The default memory address mapping is page-level interleaving. We use both micro-benchmarks and real workloads in our experiments. The micro-benchmarks include B tree (B-Tree), chain queue (ChainQueue), a hash table (HashTable), and red-block tree (RB-Tree), the real workloads include TPCC [58] and TATP [57]. Workloads are simulated under a 4-core processor and each core executes the same workload.

Our proposed LALEA and BLOM are generic logging optimization methods that can be applied to prior designs. As examples demonstrate their capabilities, we apply them to two start-of-the-art designs: REDU [6] and LAD [7]. LALEA and BLOM represent that they are applied to REDU, while LALEA-LAD denotes that LALEA optimization is applied to LAD. Since LALEA and BLOM are orthogonal, they can work together. LALEA+BLOM represents their combinations applied to REDU.

4.5.2 Transaction Throughput

Figure 4.5 shows the transaction throughput improvement over REDU. LALEA, BLOM, and LALEA+BLOM improve the throughput of REDU by 54.04%, 17.16%, and 56.62% on average, respectively. LALEA balances the banks' workloads and reduces log operation latency, leading to throughput improvement. BLOM removes the intra-recording ordering constraints and reduces log traffic, thus also improving

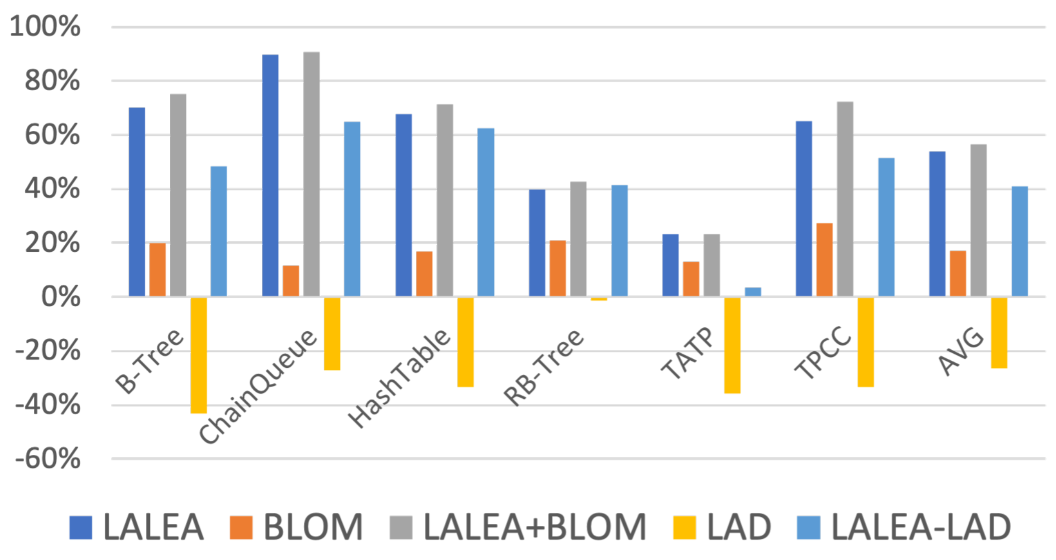


Figure 4.5: Improving transaction throughput

the throughput. LALEA is more effective than BLOM. This is because LALEA accelerates the writing of all log entries while BLOM only speeds up serving log record headers. As expected, LALEA+BLOM achieves an extra 2.58% throughput gain over LALEA.

Our evaluation shows that LAD is inferior to REDU in throughput. There are several reasons for LAD's performance degradation. First, concurrent transactions under multi-core CPU compete for the limited ADR buffer and the depletion of the ADR buffer makes LAD fall back to log operations. Second, ADR buffer capacity is limited in the commodity CPUs that support NVM. For example, the ADR buffer in the memory controller for Intel Optane memory only can buffer 16 cache lines [37]. Last but not least, LAD issues log requests when the LAD buffer overflows, while REDU can immediately initiate log requests as soon as they arrive. The delayed issue of log requests increases LAD transaction commit latency. In addition, REDU's logging scheme is more efficient than the logging scheme used by LAD. After being applied to LAD, LALEA can efficiently execute log requests and outperform LAD by 67.45% in terms of throughput.

4.5.3 Log Entry Persistence Latency

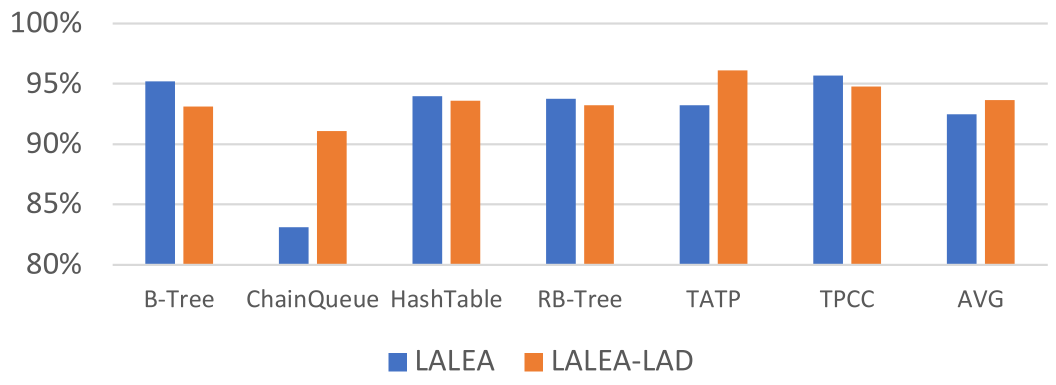


Figure 4.6: Reducing log entry persistence latency

Figure 4.6 shows incorporating LALEA into REDU and LAD can reduce the latency of log entry persistence by 92.51% and 93.66% on average over these two prior designs, respectively. LALEA effectively reduces log request queuing time by workloads over balancing NVM banks. The short queuing time can be translated into decreased persistence latency. The reduced latency minimizes the memory barrier synchronization time required by logging. A transaction commit time is determined by the execution time of transaction instructions and the synchronization time of memory barriers. The reduced log entry latency decreases the memory barrier synchronization time, improving the transaction throughput. This also explains why log entry latency improvement is more significant than the throughput gain.

4.5.4 Transaction Commit Latency

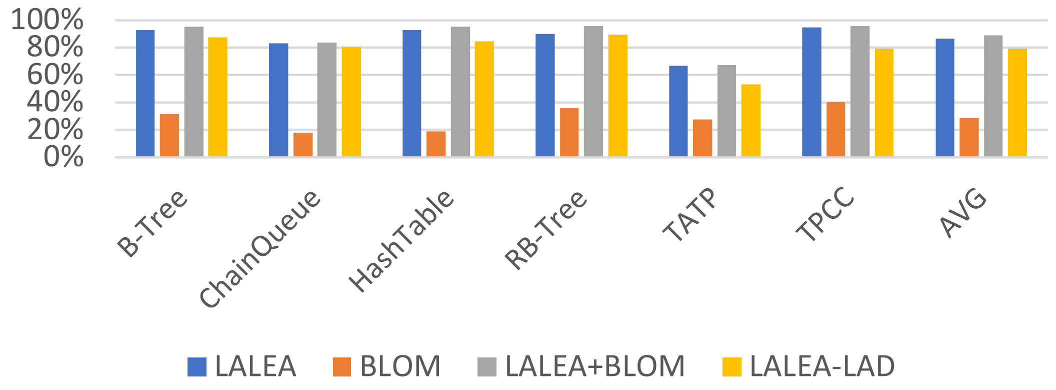


Figure 4.7: Reducing latency to commit transaction

Figure 4.7 shows transaction commit latency reduction for LALEA, BLOM, and LALEA+BLOM over REDU, and for LALEA-LAD over LAD. The latency to commit a transaction is defined as the period from the issue of its last instruction to the completion of its all log requests. On average, LALEA, BLOM, and LALEA+BLOM decrease the transaction commit latency by 86.55%, 28.74%, and 88.74%, respectively. In addition, LALEA reduces the commit latency by 79.02%, when it is applied to LAD. Since LALEA can reduce the log entry persistence latency, a transaction's log requests take a shorter time to complete, leading to a lower commit latency. BLOM avoids persisting log metadata blocks and thus also decreases the finish time of log requests. When these two optimizations work together, the commit latency is further reduced.

4.5.5 Throughput under the Alternative Address Mapping Scheme

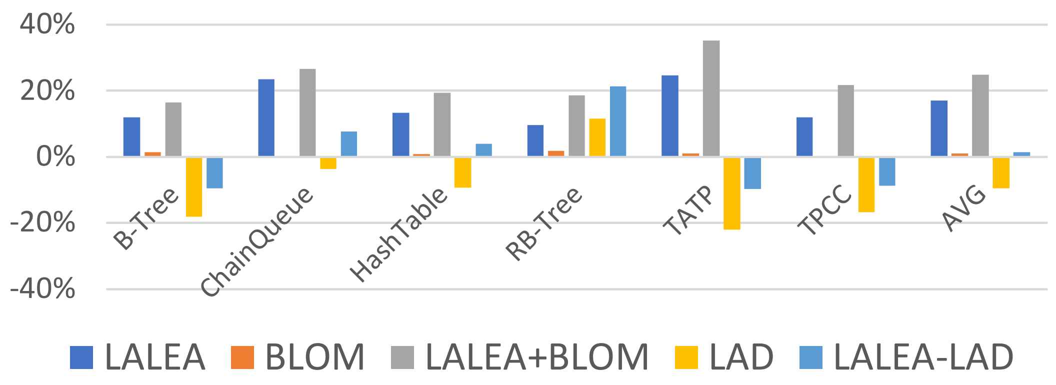


Figure 4.8: Improving transaction throughput with cache line level interleaving

We have demonstrated that LALEA and BLOM can improve the transaction execution performance, by decreasing the log entry persistence latency. These two optimizations work well when the imbalanced bank workload causes a longer log persistence latency. We believe that uneven bank workload commonly exists in programs, even though a physical memory address mapping scheme could affect the workload distribution over banks. To demonstrate LALEA and BLOM capability, we evaluate them under an aggressive address mapping scheme, cache line level interleaving, which distributes contiguous cache lines to different banks. Due to space limitations, we only present LALEA, BLOM, and LALEA+BLOM throughput improvement over REDU under this aggressive address mapping scheme. Figure 4.8 indicates that they improve the throughput by 17.01%, 0.99%, and 24.81% on average, respectively. In addition, LALEA improves LAD's throughput by 10.96%. As expected, the aggressive address mapping scheme can mitigate the imbalance of bank workload and achieve less performance gain for these optimizations than page-level address mapping. The more balanced bank workload makes BLOM achieve marginal performance improvement. However, LALEA still achieves significant performance gains.

4.6 Summary

Logging schemes are widely used to ensure crash consistency for persistent memory. The ordering constraints between log operations and transaction execution place some log operations on the I/O critical path, thus introducing severe performance overhead. We have identified two fundamental and common reasons for the inefficiency of log operations. First, the over-loaded NVM banks increase the queuing time for log requests served by these banks, leading to a high persistence latency for log requests. Second, the intra-record ordering serializes the persisting of log entries and the header, increasing transaction commit latency. To address the first issue, we propose LALEA which balances banks' workload through the novel log entries allocation method. To address the second issue, we propose BLOM that removes intra-record ordering constraints by buffering headers in the ADR buffer. Our evaluation shows that LALEA can achieve 54.04% and 17.16% higher transaction throughput on average compared to prior designs, respectively.

Chapter 5

Improving Neural Network Accelerator Performance by Optimizing Memory Accesses

5.1 Overview

This work introduces two optimizations to alleviate the bottleneck resulting from off-chip memory access during DNN inference tasks. We observed that the execution of layers with substantial input channels on edge NNAs causes significant performance overhead, and revealed that the execution of such layer introduces repeated off-chip memory accesses to IFMAP. To **A**lleviate the issue of **R**epeated **A**ccess to **I**FMAP, we propose an enhanced Weight Stationary strategy, named ARAI, achieved by reordering the iteration order within the loop-nest for wider layers. ARAI reduces repeated IFMAP access, thereby minimizing the number of off-chip memory accesses. Taking into account the concurrent off-chip memory access by multiple data tiles, we present an innovative **L**oad-**A**ware **P**lacement (LAP) scheme for feature map tiles in off-chip memory to improve the memory parallelism. LAP dynamically places tiles in optimal memory banks on-the-fly, enhancing off-chip memory throughput. We evaluate the proposed designs and compare them with STOA optimizations. Our evaluation shows that ARAI reduces inference latency by 33.37% on average. LAP outperforms the state-of-art off-chip tile mapping scheme[14] by an average of 42.00%. Since these two optimizations are orthogonal, the combination of them can achieve 61.90% performance improvement on average.

5.2 Motivation

5.2.1 Wide Layers

The NHWC *im2col* requires that pixels of a filter are firstly unrolled in the direction of depth, corresponding to different input channels. With WS, each unrolled filter is

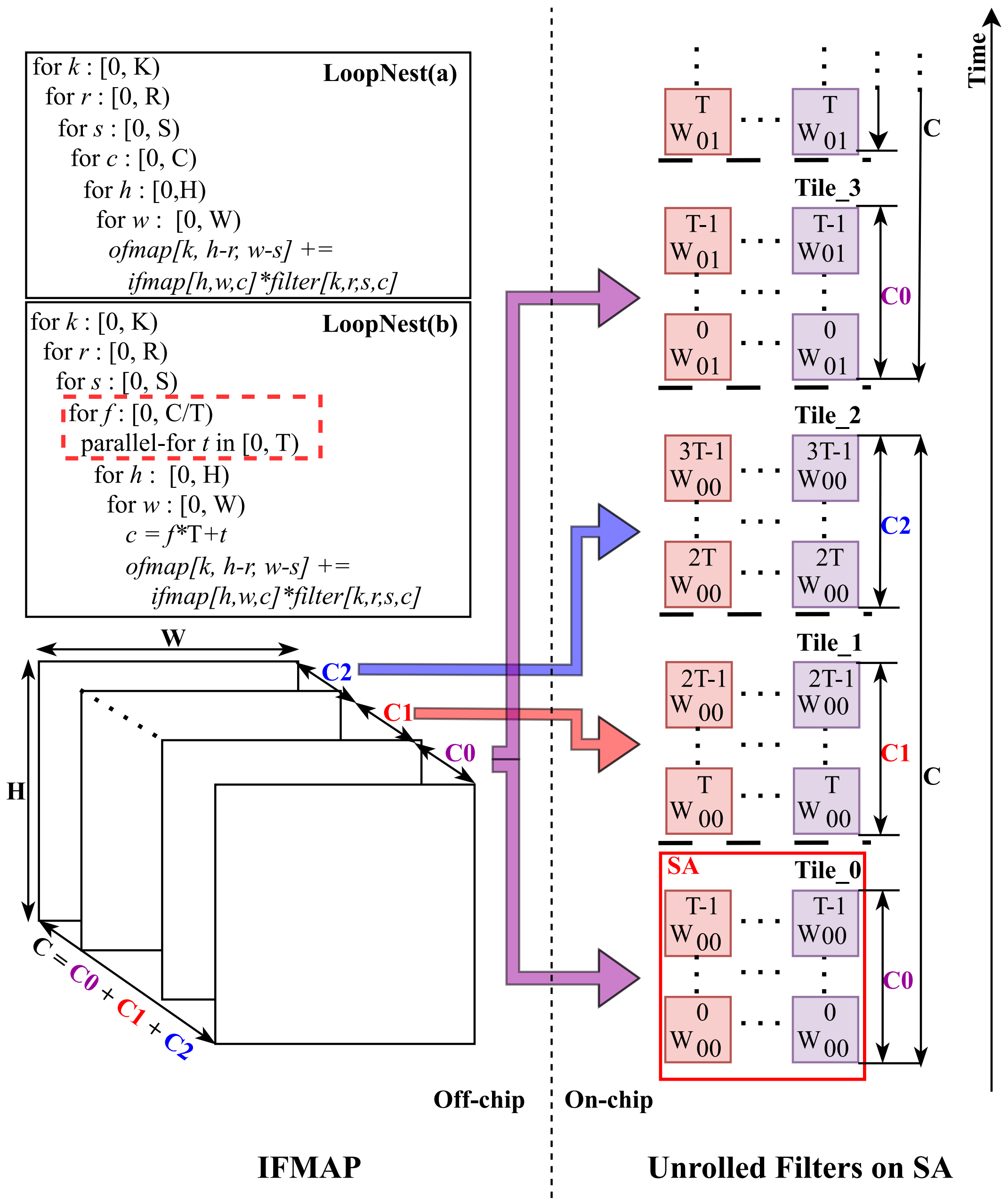


Figure 5.1: Example of repeated off-chip memory access to IFMAP in a Wide Layer. LoopNest(a) and LoopNest(b) present the convolution computation with Weight Stationary before and after mapping filters on the Systolic Array (SA), respectively. The sequence number of Tiles indicates their execution order. IFMAP is read from off-chip memory to on-chip memory. Arrows with different colors correspond to different tiles of IFMAP.

mapped to one column of SA to perform GEMM [13]. When the size of the unrolled filter is larger than the size of the SA column, each filter is partitioned into many tiles, illustrated by the LoopNests shown in Figure 5.1. LoopNest(a) depicts the convolution computation before mapping unrolled filters onto the SA. The order of iteration, $[k, r, s, c, h, w]$, from the outermost to the innermost, signifies the order of weight pixels to be loaded into the SA to complete the convolution computation. LoopNest(b), $[k, r, s, f, t, h, w]$, shows the loop nest with weight tiles after they are mapped onto the SA. Here, T represents the number of PEs in a column of the SA, which serves as the partition factor for each filter. A convolution layer is termed a wide layer when the number of input channels, C, exceeds T.

In this setup, the weights in each filter tile perform GEMM with inputs from a subset of IFMAP channels, resulting in IFMAP tiles. As the example shown in Figure 2.14, the channels of IFMAP are divided into 3 portions, C0, C1, and C2, while the first filter tile, Tile_0, does GEMM with the inputs on C0 of IFMAP. Inputs from all channels of IFMAP are processed after the f loop. The *parallel-for* t loop highlights the concurrent execution on rows of the PE matrix, where weights corresponding to the same input channel from different filters are stationed and perform MAC operations with the same inputs. Each IFMAP tile conducts GEMM with at least one filter tile, while different filter tiles may perform GEMM with the same IFMAP tile. In Figure 2.14, the inputs on C0 of IFMAP are accessed for both Tile_0 and Tile_3.

However, due to the limited size of on-chip memory, it is challenging to buffer an entire IFMAP on-chip. Inputs that are not involved in neural network computation with the current on-chip weights are released to facilitate the prefetching of other inputs for subsequent computations. In this scenario, repeated access to the same IFMAP tile located in off-chip memory could happen in wide layers.

Figure 5.1 illustrates the repeated off-chip memory accesses to IFMAP tiles with a simplified example. The shape of the filter is 3x3 on each input channel, indicated by weight pixels $W_{00}{\sim}W_{08}$. The superscript of a Weight pixel indicates the input channel which it will do MAC with. Weights from different filters are presented in different colors. The number of IFMAP channels, C, is divided into three parts by T, which are $C0$, $C1$, and $C2$. Due to the double-buffering on-chip memory, IFMAP tiles are repeatedly released from on-chip memory and fetched from off-chip memory between two iterations in the s loop. In this example, after the execution of the filter tile $Tile\_0$, the inputs on C0 (presented by purple arrow) are released to save on-chip memory space for the coming inputs on C2 (presented by blue arrow) at the current iteration in the s loop. But at the next iteration, the filter tile $Tile\_3$ will do GEMM with the inputs on C0 again, leading to repeated reading from off-chip memory. The inputs on C1 (presented by red arrow) and C2 meet the same problem.

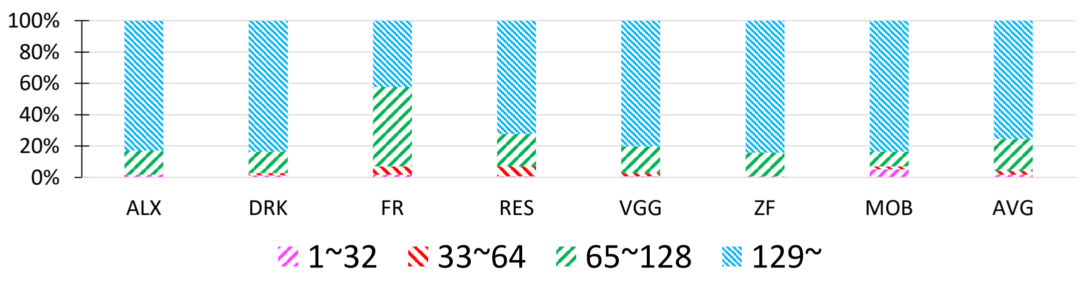


Figure 5.2: Execution time distribution by layer types

5.2.2 Performance Impact of Wide Layers.

In modern DNN models [40, 62, 63, 64, 65, 66, 67], except for the initial layers, the convolution layers typically feature a substantial number of input channels, reaching up to 1024. This increase in input channels is indicative of the abundance of features captured by the preceding layers. The prevalence of a large number of input channels often leads to the occurrence of wide layers, especially in edge devices where the PE matrix of the SA typically ranges from 16×16 to 64×64[68, 69, 70].

We have observed that wide layers experience significant memory-stall time when executing DNN models on edge NNAs. Figure 5.2 and Figure 5.3 illustrate the distribution of execution time and memory-stall time for layers with varying numbers of IFMAP channels when these NN models are run on an NNA with a 32x32 SA.

Layers in all the NN models have been categorized into four types based on the number of IFMAP channels: narrow layer (1∼32), medium layer (33∼64), wide layer

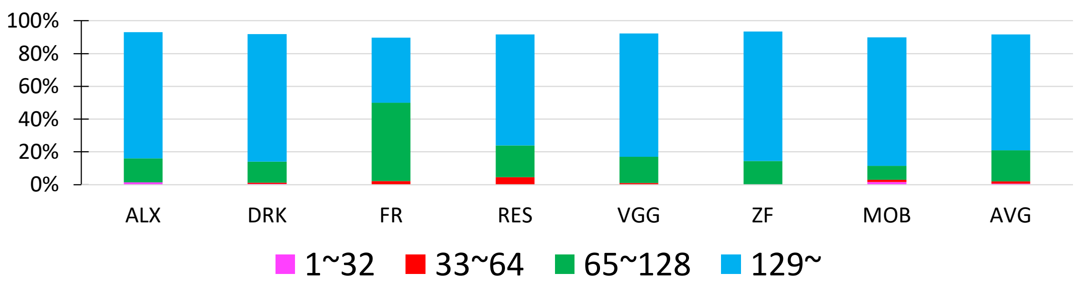


Figure 5.3: Memory-stall time distribution by layer types

(65∼128), and wider layer (more than 128). The distribution of execution time reflects the percentage of execution time that different layer types consume during the NN inference. Meanwhile, the distribution of memory-stall time denotes the percentage of time that different layer types take to access off-chip memory.

As presented by Figure 5.2, wide and wider layers account for the highest percentage of the execution time, averaging 20.31% and 75.53%, respectively. This means that a substantial portion of the inference time is primarily dedicated to the execution of wide and wider layers.

Additionally, the distribution of memory-stall time in Figure 5.3 shows that off-chip memory access time of wide layers and wider layers constitutes an average of 19.03% and 70.75% of the model execution time, respectively. This reveals the significant overhead caused by off-chip memory access when processing wide and wider layers. These observations motivate us to explore strategies to alleviate the performance bottleneck caused by wide layers when running NN inference on edge devices.

To tackle the performance bottleneck on edge NNAs arising from off-chip memory access, we begin by introducing an optimization for Weight Stationary to **A**lleviate the issue of **R**epeated **A**ccess to **I**FMAP (ARAI) in off-chip memory. Subsequently, we present the concept of **L**oad-**A**ware **P**lacement (LAP). LAP is designed to optimize the placement of tiles on off-chip memory, thereby reducing the time required for accessing data in off-chip memory.

5.3 Alleviate Repeated Access to IFMAP (ARAI)

To **A**lleviate the issue of **R**epeated **A**ccess to **I**FMAP caused by wide layers, we introduce ARAI as an optimized Weight Stationary dataflow. While preserving the NHWC *im2col* format, filter tiles undergoing GEMM operations with the same IFMAP tile are processed continuously. Unlike the loop order $[k, r, s, f, t, h, w]$ employed by WS in LoopNest(b) as shown in Figure 5.1, ARAI reorders the loops as $[k, f, r, s, t, h, w]$ to facilitate the reuse of on-chip IFMAP tiles, thereby minimizing off-chip memory traffic.

Figure 5.4 provides an example to explain how ARAI operates, using the same illustrative example presented in Section 5.2. In the LoopNest, c represents the input channel to be accessed. With the use of *parallel-for*, inputs from IFMAP channels ranging from $f * T$ to $f * T + T - 1$ are concurrently streamed to various rows of

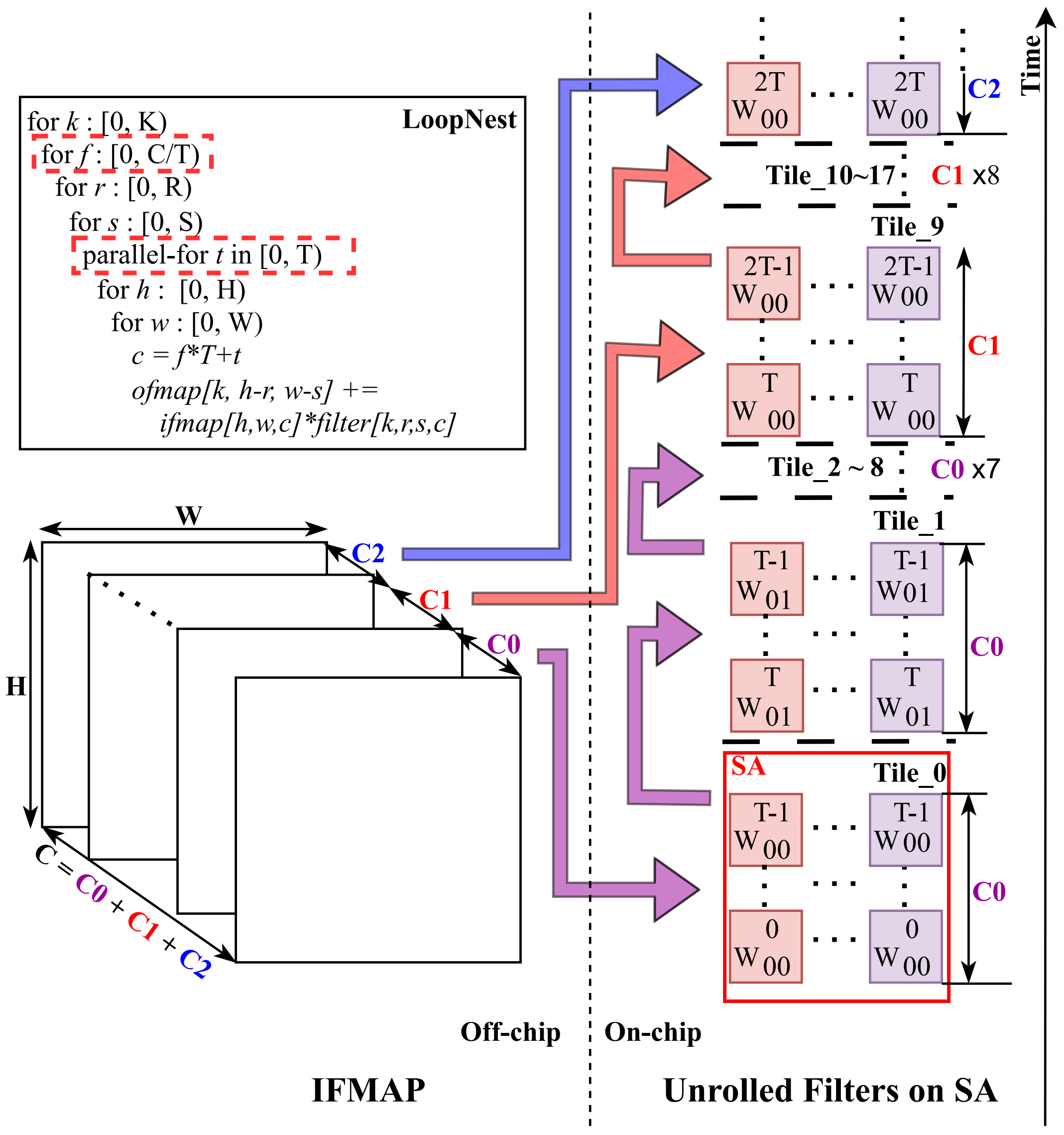


Figure 5.4: Illustration of ARAI

the SA. By moving the f loop outside the r loop and s loop, ARAI enables inputs from these IFMAP channels to continually participate in GEMM operations with different filter tiles, as indicated by all iterations of the r and s loops. This process is visualized with different colored arrows in Figure 5.4. For instance, after inputs on C0 (depicted by the purple arrow) are read from off-chip memory for Tile_0, they are retained on-chip and continually utilized until the final filter tile in the r and s loops, Tile_8. The subsequent iteration in the f loop requires input from C1 (represented by the red arrow), which remains in on-chip memory until the conclusion of Tile_17. During this process, inputs on C0 in on-chip memory are released to allow for the prefetching of inputs on C2 (illustrated by the blue arrow).

With ARAI, each IFMAP tile is read only once during the r loop and s loop, thus preventing the repeated off-chip memory access for the same IFMAP tiles, as illustrated in Figure 2.14. Importantly, ARAI can be seamlessly integrated into Deep Learning frameworks that support the NHWC format, such as Tensorflow [25].

5.4 Load-Aware Placement of Data Tiles (LAP)

ROMANet [14] has introduced an off-chip memory mapping policy that leverages the multi-bank burst feature of DRAM. This approach improves DRAM Bandwidth by exploiting access parallelism across multiple banks. However, ROMANet performs off-chip memory mapping for each data tile separately, without considering the potential intra/inter memory contention of data tiles. During inference, both filter tiles and IFMAP tiles are read from off-chip memory. Additionally, intermediate results,

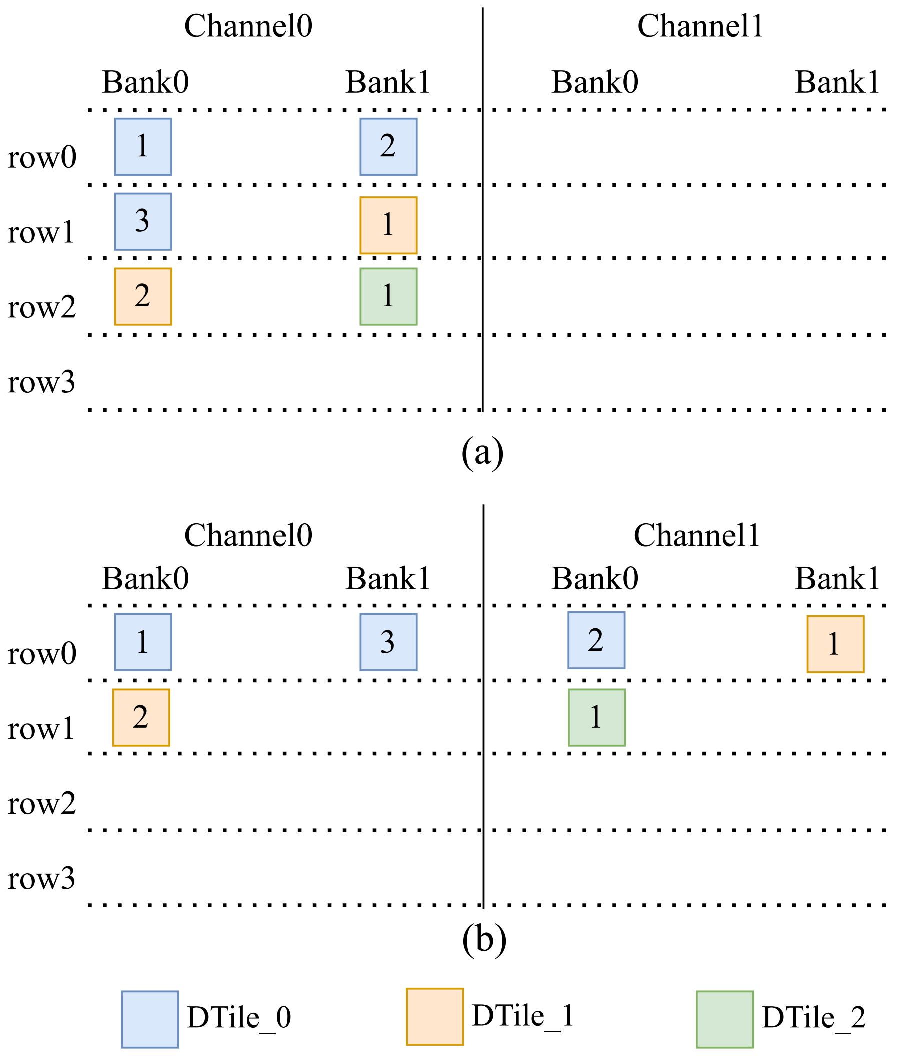


Figure 5.5: Off-chip memory mapping examples. Off-chip memory has 2 channels, 2 banks per channel, and 4 rows per bank. There are three concurrent data tiles. The numbers inside blocks indicate their order during memory allocation. (a) is based on ROMANet. (b) is based on LAP.

namely OFMAP tiles, need to be written back to off-chip memory due to the constraints of on-chip memory size. Such concurrent off-chip memory access could serialize the access to different data tiles on banks. For instance, as shown in Figure 5.5(a), three data tiles, DTile_0, DTile_1, and DTile_2, on bank1 of channel0 are serially accessed. Another limitation of ROMANet is the underutilization of parallelism on the channel level within the current multichannel off-chip memory system. It prioritizes placing a data tile across banks of a DRAM rank over ranks and channels. The shared bus among banks within a channel restricts DRAM Bandwidth.

To address memory contention and leverage channel-level parallelism, we introduce a novel approach called **L**oad-**A**ware **P**lacement (LAP) for feature maps in DRAM. The objective of LAP is to enhance DRAM Bandwidth. Given that the OFMAP of the intermediate layers becomes the IFMAP for another layer, we simplify our discussion by collectively referring to both IFMAP tiles and OFMAP tiles as DTiles.

LAP determines the DRAM locations for OFMAP tiles based on the existing layout of DTile and the load status of DRAM. When a DTile is generated, LAP divides its DRAM accesses into multiple tasks, each at the granularity of the DRAM page size. Then, LAP allocates a free DRAM row to each of these tasks, referred to as rowTasks. The allocation process for a DTile is outlined by the pseudo-code in Algorithm 2. During the allocation for each rowTask, LAP first ensures that there is no DRAM access contention with other rowTasks originating from the same DTile (line: 2 and line: 19). It then identifies the DRAM banks with the lightest load as potential allocation positions (lines: 4 $\sim$ 6). A DRAM location, including the indices of channel, rank, bank, and row, is selected from these potential positions while prioritizing channel-level parallelism (lines: 7 $\sim$ 17). In the example illustrating DRAM allocation with LAP, as shown in Figure 5.5(b), three rowTasks belonging to DTile_0

Algorithm 2: Pseudo code of DRAM Allocation for a DTile in LAP

```
Data:
RowTask: DRAM access task at the granularity of DRAM page size.
SelfMinBkRkCh: DRAM locations with minimum rowTasks of current DTile.
MemMinBkRkCh: DRAM locations with minimum rowTasks scheduled to
access.
$\#Channel$: The number of channels in DRAM.
$\#Rank$: The number of ranks per DRAM Channel.
$\#Bank$: The number of banks per DRAM Rank
$\#Row$: The number of rows per DRAM Bank.
1  for rowTask in RowTasks do
2      for BkRkCh in SelfMinBkRkCh do
3          //Check the layout of DTile itself.
4          if BkRkCh is in MemMinBkRkCh then
5              //Check current DRAM Status.
6              BkRkChRecords.insert(BkRkCh);
7      for $bk \leftarrow 0$ to $(\#Bank - 1)$ do
8          for $rk \leftarrow 0$ to $(\#Rank - 1)$ do
9              for $ch \leftarrow 0$ to $(\#Channel - 1)$ do
10                 //Channel Level Parallelism.
11                 $tmpBkRkCh \leftarrow (bk, rk, ch)$;
12                 if tmpBkRkCh is in BkRkChRecords then
13                     for $ro \leftarrow 0$ to $(\#Row - 1)$ do
14                         $memLocation \leftarrow (tmpBkRkCh, ro)$;
15                         if memLocation is free then
16                             //Allocate.
17                             rowTask.setMemInfo(memLocation);
18                             goto Foo;
19     Foo: Update SelfMinBkRkCh;
```

are prioritized to different DRAM channels, Channel0 and Channel1. Subsequently, they are assigned to different banks within the same channel. When it comes to DTile_1, the first rowTask is allocated to the DRAM bank with the lightest load, which is Bank1 in Channel1. In situations where all banks have an equal load status,

Algorithm 3: Generate DRAM Address to Access from rowTasks in LAP

```
Data:
rTaskListOnBkRkCh: List of scheduled read rowTasks on all DRAM banks.
wTaskListOnBkRkCh: List of scheduled write rowTasks on all DRAM banks.
NextBkRkChToCheck: Containing IDs of Bank, Rank, and Channel.
$\#Channel$: The number of channels in DRAM.
$\#Rank$: The number of ranks per DRAM Channel.
$\#Bank$: The number of banks per DRAM Rank
Result:
Addr: address of a DRAM block.
1  $nextBk \leftarrow Bk$ in NextBkRkChToCheck;
2  $nextRk \leftarrow Rk$ in NextBkRkChToCheck;
3  $nextCh \leftarrow Ch$ in NextBkRkChToCheck;
4  $rowTaskPtr \leftarrow NULL$;
5  //Search the rowTask from NextBkRkChTocheck;
6  for $bkOffset \leftarrow 0$ to $(\#Bank - 1)$ do
7      $bk \leftarrow (bkOffset + nextBk)\text{mod}(\#Bank)$;
8      for $rkOffset \leftarrow 0$ to $(\#Rank - 1)$ do
9          $rk \leftarrow (rkOffset + nextRk)\text{mod}(\#Rank)$;
10         for $chOffset \leftarrow 0$ to $(\#Channel - 1)$ do
11             $ch \leftarrow (chOffset + nextCh)\text{mod}(\#Channel)$;
12             $BkRkCh \leftarrow (bk, rk, ch)$;
13             if have valid rowTask in rTaskListOnBkRkCh[BkRkCh] then
14                 $rowTaskPtr \leftarrow$ rTaskListOnBkRkCh[$BkRkCh$].front();
15             else if has valid rowTask in wTaskListOnBkRkCh[BkRkCh] then
16                 $rowTaskPtr \leftarrow$ wTaskListOnBkRkCh[$BkRkCh$].front();
17             //Generate DRAM Address to access
18             if $rowTaskPtr$ is not $NULL$ then
19                 Addr $\leftarrow rowTaskPtr$.nextAddrToAccess();
20                 //update $NextBkRkChToCheck$ with $(bk, rk, ch)$;
21                 $(nextBk, nextRk, nextCh) \leftarrow (bk, rk, ch)$;
22                 $nextCh \leftarrow (ch + 1)$;
23                 if $nextCh >= \#Channel$ then
24                     $nextCh \leftarrow 0$;
25                     $nextRk \leftarrow (rk + 1)\text{mod}\#Rank$;
26                     if $nextRk == 0$ then
27                         $nextBk \leftarrow (bk + 1)\text{mod}\#Bank$;
28                 $NextBkRkChToCheck \leftarrow (nextBk, nextRk, nextCh)$;
29                 goto: Foo;
30 ... //Search from $(0, 0, 0)$ to NextBkRkChTocheck;
31 Foo: return Addr;
```

the second rowTask of DTile_1 is allocated to the bank of another channel, specifically Bank0 in Channel0. In scenarios where the load status varies among DRAM Channels, rowTask will be allocated to the bank of the channel with the lightest load, as illustrated in the allocation of the rowTask for DTile_2.

Based on the allocation of a DTile, its mapping information is stored to support write-back to DRAM and future retrieval as input from DRAM. The mapping information for a DTile is released after it is no longer needed as input.

Algorithm 3 outlines the logic for maximizing DRAM access parallelism with LAP. DRAM access requests are prioritized to be sent to different channels, then ranks, and banks (lines: $5 \sim 11$ and lines: $20 \sim 26$) if there is a scheduled rowTask for that specific position. Similar to ROMANet, LAP utilizes the FCFS policy for DRAM access. This choice ensures that the DRAM access requests are served by DRAM in the same order as they are sent, maximizing parallelism on both the channel and bank levels.

5.4.1 Microarchitecture of LAP

To facilitate Load-Aware Placement (LAP), we introduce the DTile Manager Unit (DMU) on the chip. The DMU supports both DRAM allocation and address mapping for DTiles, extending the address translation supported by DMA [17, 18, 29, 32, 41].

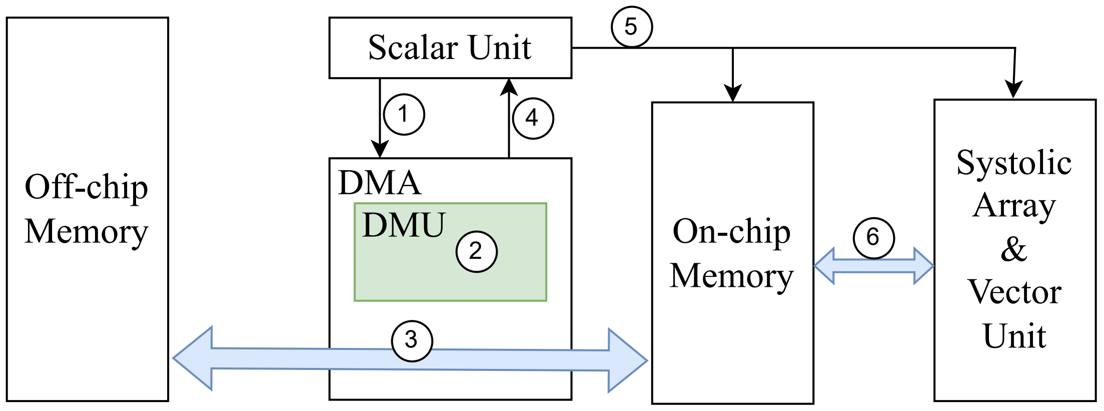


Figure 5.6: Architecture overview of LAP

Integrated with software-managed DMA, the DMU is involved in the data transfer between on-chip memory and off-chip memory. Figure 5.6 depicts an architectural overview of LAP.

The scalar unit manages the compiler-controlled memory hierarchy and on-chip computation. To facilitate off-chip memory access for DTiles, relevant instructions issued to the DMA are expanded to include a description of the DTile, incorporating details such as the operation type (Read/Write), the global unique index of the DTile, and the size of the data, as depicted in Figure 5.8 and Figure 5.9. Instructions originating from the scalar unit for off-chip memory access of DTile (1) prompt the DMU to execute DRAM allocation for the DTile intended for writing back to off-chip memory or address mapping for the DTile read from off-chip memory (2). Additionally, the DMU generates the sequence of memory accesses (2), as illustrated in Algorithm 3. Subsequently, data is transferred between on-chip memory and off-chip memory (3).

Upon completion of the data transfer of DTile, a completion notification is sent

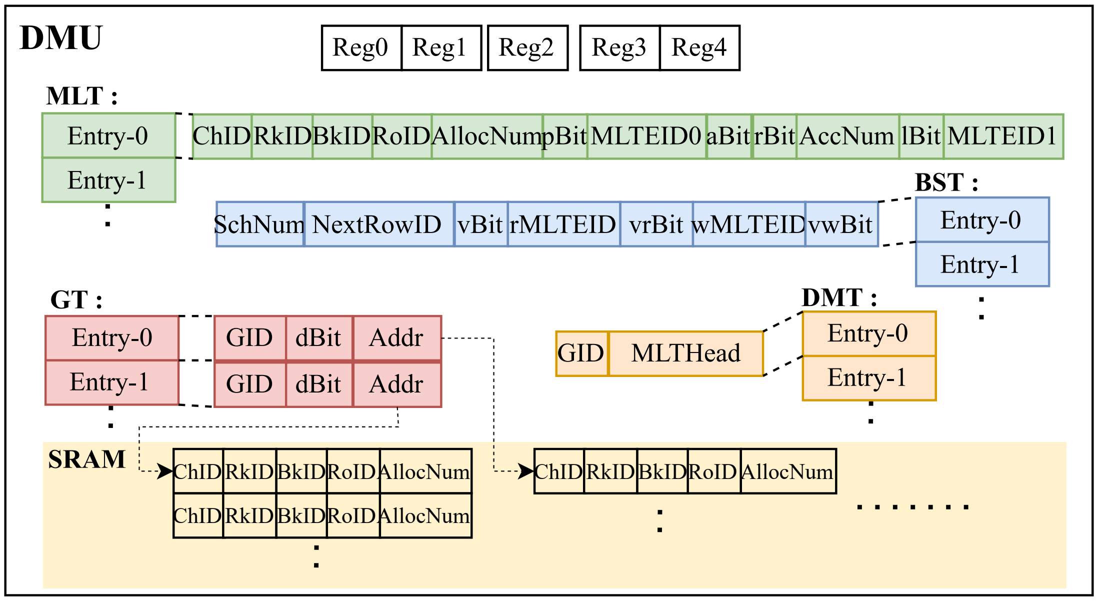


Figure 5.7: Key components in DTile Manager Unit (DMU): Memory Location Table (MLT), DTile Mapping Table (DMT), Bank Status Table (BST), Off-chip DTile Tracking Table (GT), Reg0: The head of free MLTEntries, Reg1: The tail of free MLTEntries, Reg2: SelfMinBkRkCh in Algorithm 2, Reg3: MemMinBkRkCh in Algorithm 2, Reg4: NextBkRkChToCheck in Algorithm 3

back to the scalar unit (4), allowing for the continuation of the program execution (5)(6). For instance, after DMA completes the off-chip memory read of DTile, the scalar unit could issue instructions to initiate on-chip computation (5). During the computation, input is streamed through the computation units, while intermediate results are buffered in the on-chip memory (6).

***DTile* Manager Unit:** DMU is responsible for storing and managing data structures related to LAP, ensuring efficient allocation and DRAM access for DTiles. Figure 5.7 provides a detailed overview of the DMU.

On-chip DTile Tracking: In LAP, DRAM access from a DTile is divided into multiple rowTasks, each with separately determined locations. To keep track of the DRAM locations and manage DRAM access for on-chip DTiles, we introduce two essential data structures: the Memory Location Table (MLT) and the DTile Mapping Table (DMT). These tables are crucial for efficient DRAM allocation, as outlined in Algorithm 2, and for managing DRAM access as described in Algorithm 3. The MLT is responsible for recording all DRAM locations allocated for on-chip DTiles and monitoring the status of DRAM access for rowTasks. Each entry in the MLT, referred to as an MLTEntry, contains detailed information about the DRAM location allocated to a rowTask, including the IDs of the channel (ChID), rank (RkID), bank (BkID), and row (RoID). Additionally, the number of memory blocks allocated (AllocNum) for each rowTask is recorded in the MLTEntry. Since multiple rowTasks can be generated from a single DTile, the DRAM locations of one DTile could be recorded in multiple MLTEntries. These entries are organized into a list by including the index of the subsequent MLTEntry (MLTEID0), whose validity is indicated by a flag bit (pBit). To distinguish the operations in which MLTEntry is involved, two additional flag bits (aBit and rBit) are introduced. The aBit is introduced to indicate whether the DTile is during allocation of LAP. The rBit is to denote whether the given rowTask generates read or write DRAM access requests. To monitor the progress of DRAM access for each rowTask, the count of memory blocks to be accessed (AccNum) is maintained within each MLTEntry. Upon reaching zero, the DRAM access for the respective rowTask is considered complete. During DRAM access from DTiles, the scenario may arise where a single DRAM bank is accessed by multiple rowTasks. MLTEntries associated with rowTasks that access the same DRAM bank are categorized into read and write lists. To form a list, each MLTEntry records another index of MLTEntry (MLTEID1), corresponding to the subsequent rowTask scheduled for the same bank. A flag bit (lBit) is to denote whether the MLTEID1 is valid.

The DTile Mapping Table (DMT) is responsible for tracking DRAM locations for on-chip DTiles. Each on-chip DTile corresponds to a DMT entry, which contains the global unique index of this DTile (GID) and the index of the first MLTEntry in the list of its MLTEntries (MLTHead). The GID serves as the sequence number for DTile generation during NN inference. Once the inference is compiled, the execution order of filter tiles is determined, which in turn indicates the sequence of DTile generation. To facilitate the DRAM allocation of LAP, we introduce Reg0 and Reg1, which record the head and tail of the list of free MLTEntries used to track DRAM locations for rowTasks. As DRAM locations are sequentially determined for DTiles by LAP, we introduce a bitmap register, Reg2, to indicate which banks have had the minimum rowTasks allocated to the current DTile. This information is used as SelfMinBkRkCh in Algorithm 2.

DRAM Status Tracking: As shown in Algorithm 2 and Algorithm 3, the status of DRAM banks is referred for both DRAM allocation for DTiles and DRAM access from DTiles in LAP. To track the status of DRAM, we introduce the Bank Status Table (BST) and two bitmap registers, Reg3 and Reg4. Each individual DRAM bank corresponds to a unique entry in BST and a bit in both Reg3 and Reg4. For Algorithm 2, each BST entry records information about the bank, such as the number of rowTasks scheduled to be accessed on this bank (SchNum), the index of the next DRAM row to be allocated for a rowTask (NextRowID), and a flag bit (vBit) indicating the validity of NextRowID for the allocation. NextRowID is updated during the allocation of a rowTask and the release of a rowTask. Its value is set in ascending order within the range of rows per bank. With the configuration of off-chip memory in Table 5.3, inference tasks for Neural Network models listed in Table 5.4 cannot exceed the space of DRAM. The preliminary analysis of these workloads indicates that VGG16 has the largest storage requirement, which is 80.56MB. With the parallelism achieved by LAP, it is hard for NextRowID to exceed the number of rows per bank during the inference task. Thus, the row indicated by NextRowID is always free for allocation. Once NextRowID has exceeded the range of rows in one bank, this indicates that the range of rows in all other banks is going to be exceeded. In this case, the storage requirement of the inference task is over the capacity of DRAM, terminating program execution. Reg3 is responsible for summarizing the banks with the minimum number of rowTasks scheduled to access. Each bit in Reg3 indicates whether a specific bank has the minimum number of rowTasks scheduled to access.

With Reg3, it is straightforward to obtain MemMinBkRkCh in Algorithm 2. MemMinBkRkCh, as indicated by Reg3, is based on the value of SchNum for all banks. To achieve maximum parallelism in DRAM access, the rowTasks scheduled to access each bank should be tracked. In DMU, each rowTask from an on-chip DTile has a unique MLTEntry to indicate its DRAM location. Each BST entry tracks two lists of MLTEntries for read and write. In a layer, DRAM read access is prioritized over DRAM write access when there is contention on a DRAM bank, as shown in Algorithm 3. This is because, during the execution of the layer, the time to complete NN computation is affected by reading related DTiles. For each list, MLTEID1 in each MLTEntry plays the role of the pointer to the subsequent MLTEntry. RowTasks represented by each list of MLTEntries are served in the order in which they appear in the list. The index of MLTEntry to read (rMLTEID) indicates the first rowTask in the read list which has DRAM read accesses to this bank. Similarly, the first rowTask in the write list is tracked by wMLTEID. Two flag bits, vrBit and vwBit, indicate whether rMLTEID and wMLTEID are valid, respectively. Additionally, the bit in Reg4 indicates whether a specific bank is actively serving access. NextBkRkChToCheck, as indicated by Reg4, is updated each time a DRAM access request is generated.

Off-chip DTile Tracking: Due to the limited size of on-chip memory, the OFMAP needs to be written back to off-chip memory after generation. To track the off-chip DTiles for future reading, each rowTask is tracked by its DRAM location information, including ChID, RkID, BkID, RoID, and AllocNum. The DRAM locations of rowTasks originating from the same DTile are continuously stored either in the SRAM of DMU or in DRAM. Thus, we introduce a table indexed by GID (GT). Each GT entry contains the unique GID for a DTile, a flag bit (dBit), and an address (Addr). To minimize the time overhead of DRAM mapping, the DRAM location information for off-chip DTiles is prioritized to be stored in the SRAM of DMU. Once the limit of SRAM capacity is going to be exceeded, subsequent DTiles' location information is directed to be stored in DRAM. The dBit indicates whether the DRAM locations of rowTasks are stored in the SRAM of DMU or in DRAM. The Addr specifies the starting SRAM/DRAM address where the tracking information of rowTasks from this DTile is stored. The off-chip tracking information of a DTile will only be released when its data has been released from DRAM.

The DMU plays a crucial role in the generation and reading of DTiles, using the data structures mentioned above. We outline how DMU operates during these processes:

Generating DTile: Figure 5.8 illustrates the procedure by which DMU facilitates the DRAM allocation for DTile. In this process, a DMT entry is allocated. Additionally, free MLTEntries are assigned to record DRAM locations and monitor the status of DRAM access for these rowTasks. While MLTEntries are assigned to rowTasks, the head of the free MLTEntries list is updated and recorded in Reg0. Values pertaining to DRAM locations in each MLTEntry are set according to Algorithm 2,

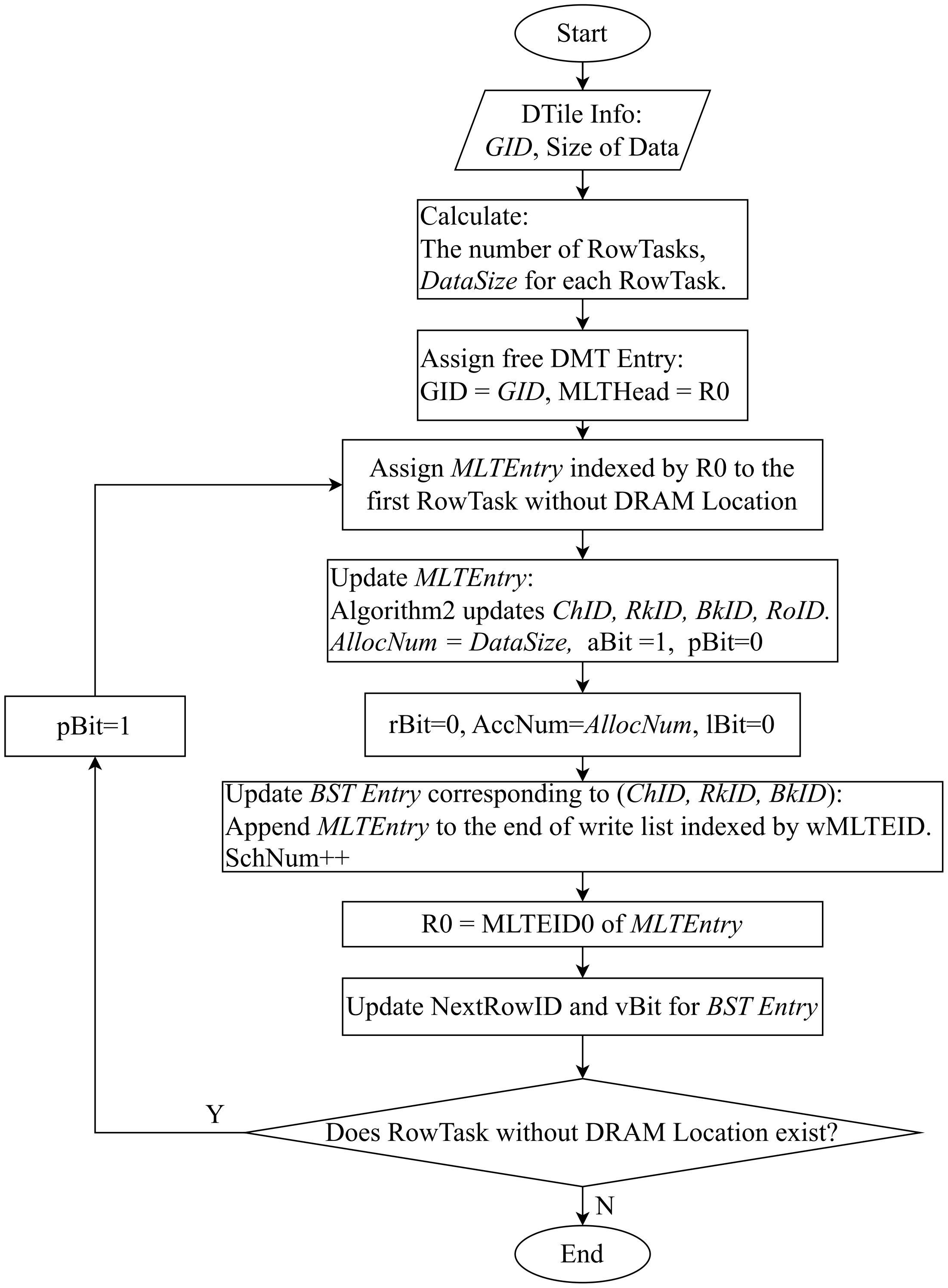

Start
DTile Info:
GID, Size of Data
Calculate:
The number of RowTasks,
DataSize for each RowTask.
Assign free DMT Entry:
GID = *GID*, MLTHead = R0
Assign *MLTEntry* indexed by R0 to the first RowTask without DRAM Location
Update *MLTEntry*:
Algorithm2 updates *ChID, RkID, BkID, RoID.*
AllocNum = DataSize, aBit =1, pBit=0
rBit=0, AccNum=*AllocNum*, lBit=0
Update *BST Entry* corresponding to (*ChID, RkID, BkID*):
Append *MLTEntry* to the end of write list indexed by wMLTEID.
SchNum++
R0 = MLTEID0 of *MLTEntry*
Update NextRowID and vBit for *BST Entry*
Does RowTask without DRAM Location exist?
Y
N
pBit=1
End

Figure 5.8: Main steps of DRAM allocation for DTile in LAP

including ChID, RkID, BkID, RoID, and AllocNum. The aBit is set to 1, indicating that this DTile is undergoing DRAM allocation. Reg2 and Reg3 provide SelfMinBkRkCh and MemMinBkRkCh for Algorithm 2, respectively. NextRowID and vBit in a BST entry are updated after an MLTEntry is allocated to this bank. During the writeback to DRAM, all rBits in MLTEntries are set to 0, indicating that they are write tasks. Based on the DRAM bank to be accessed, each MLTEntry is appended to the write lists of MLTEntries scheduled for that bank. SchNum in this BST entry is also updated. Algorithm 3 determines whether a DRAM write request is sent to the DRAM location indicated by the MLTEntry with wMLTEID of a BST entry. NextBkRkChToCheck provided by Reg4 is checked and updated during this process. AccNum in the MLTEntry is initialized to the value of AllocNum and decreased during DRAM access. When the value of AccNum reaches 0, it signifies that all data for this DRAM location has been written back. The wMLTEID in this BST entry is then moved to the index of the next MLTEntry in its write list. Once all the data of a DTile has been successfully written back to DRAM, its DRAM location information is transferred to either the SRAM or DRAM, tracked by the GT. The MLTEntries associated with this DTile are then reset and appended to the tail of the list of free MLTEntries, making them available for future allocations or reading.

Reading DTile: As shown in Figure 5.9, when a request to read a DTile from off-chip memory is received, DMU first transfers the mapping information tracked by a GT entry to a DMT entry and multiple MLTEntries. The values of DRAM locations,

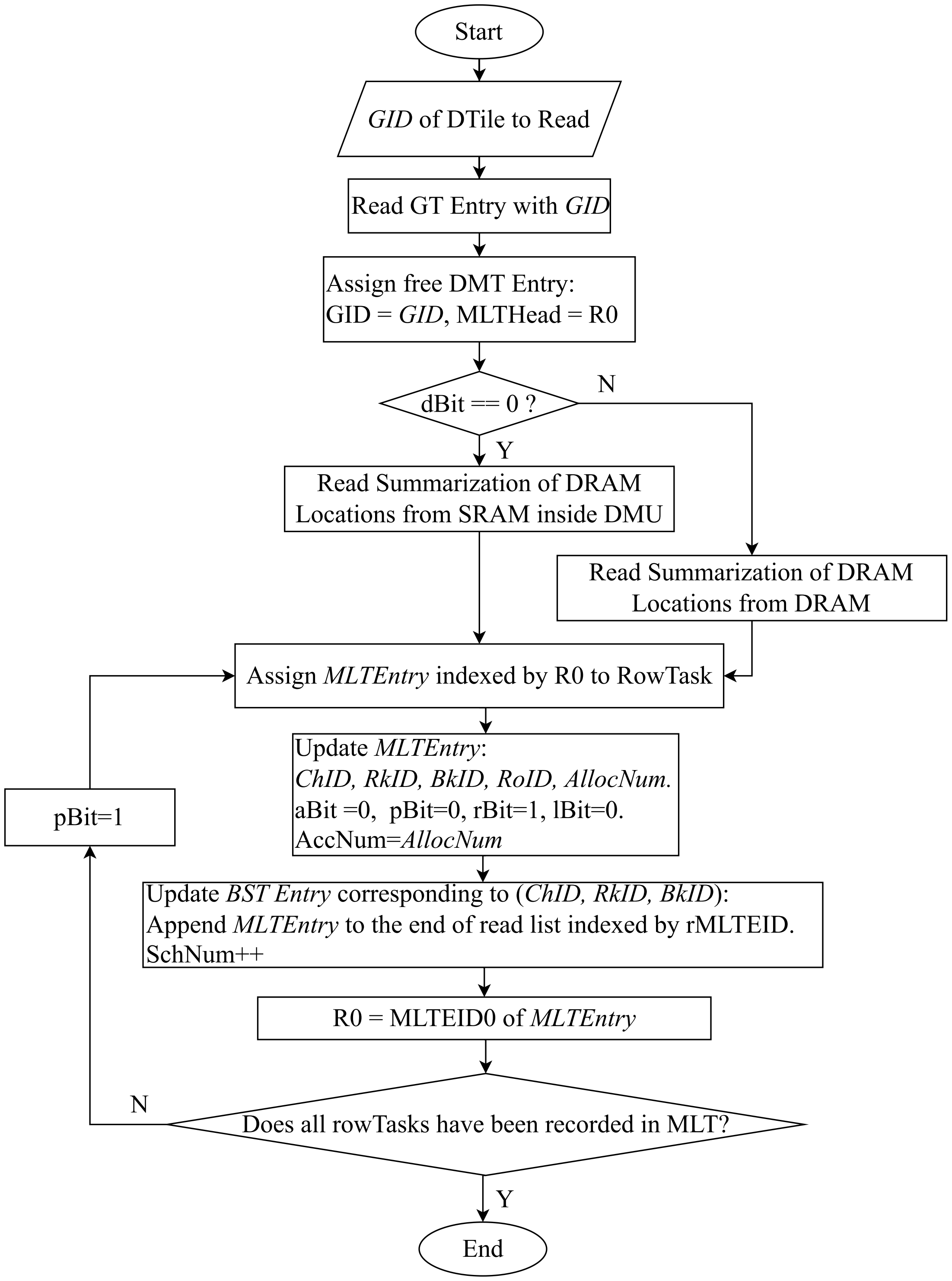

Start
GID of DTile to Read
Read GT Entry with GID
Assign free DMT Entry:
GID = GID, MLTHead = R0
dBit == 0 ?
N
Y
Read Summarization of DRAM Locations from SRAM inside DMU
Read Summarization of DRAM Locations from DRAM
Assign MLTEntry indexed by R0 to RowTask
Update MLTEntry:
ChID, RkID, BkID, RoID, AllocNum.
aBit =0, pBit=0, rBit=1, lBit=0.
AccNum=AllocNum
pBit=1
Update BST Entry corresponding to (ChID, RkID, BkID):
Append MLTEntry to the end of read list indexed by rMLTEID.
SchNum++
R0 = MLTEID0 of MLTEntry
N
Does all rowTasks have been recorded in MLT?
Y
End

Figure 5.9: Main steps of reading DRAM location for off-chip DTile

including ChID, RkID, BkID, RoID, and AllocNum, are copied from the off-chip DTile tracking information pointed to by Addr in the GT entry. The MLTEntries are allocated from the free list of MLTEntries and initialized for DRAM read access with the DRAM locations. Similar to the writeback to DRAM, based on the banks to be accessed, these MLTEntries of DTile to be read from off-chip memory are appended to different read lists recorded by BST entries and served. When a DTile needs to be released from on-chip memory, either due to the limited size of on-chip memory or the absence of further need for its data, the MLTEntries of this DTile are reset and appended to the tail of the list of free MLTEntries.

Overhead: The space requirements for the key components in DMU, based on the NNA system configuration in Table 5.3, are detailed in Table 5.1. For tracking 2MB of on-chip memory (double-buffering SPM), 2048 MLTEntries are required, requiring 16KB of SRAM. BST comprises 64 entries and consumes 0.5KB of SRAM. Though the number of on-chip and off-chip DTiles varies, 64KB of SRAM is sufficient to accommodate all the data structures mentioned above, which is based on the observation of workloads in Table 5.4. According to Cacti [49], the area of 64KB SRAM in DMU is approximately 0.35 mm^2 when implemented with 22 nm technology. After optimization by neural network compilers, the execution of the inference task becomes static, leading to a predetermined order for generating and reading DTiles. Moreover, since DRAM is dedicated to a single Systolic Array, both DRAM allocation and the status of DRAM access are entirely determined by the inference run on this Systolic array.

Components	Filed	Size Requirement
Reg0∼4	64-bit ×5	
Memory Location Table (MLT) 64-bit Entry×2048	ChID	1 Bit for 2 Channels
	RkID	1 Bit for 2 Ranks
	BkID	4 Bits for 16 Banks
	RoID	15 Bits for 32768 Rows
	AllocNum	7 Bits for 128 Mem Blocks
	pBit	1 Bit
	MLTEID0	11 Bits for 2048 Entries
	aBit	1 Bit
	rBit	1 Bit
	AccNum	7 Bits 128 Mem Blocks
	lBit	1 Bit
	MLTEID1	11 Bits for 2048 Entries
DTile Mapping Table (DMT) 32-bit Entry×N, N≤34	GID	15 Bits
	MLTHead	11 Bits for 2048 Entries
Bank Status Table (BST) 64-bit Entry×64	SchNum	11 Bits for 2048 Entries
	NextRowID	15 Bits for 32768 Rows
	vBit	1 Bit
	rMLTEID	11 Bits for 2048 Entries
	vrBit	1 Bit
	wMLTEID	11 Bits for 2048 Entries
	vwBit	1 Bit
DTile Tracking Table (GT) 64-bit Entry, Dynamic	GID	15 Bits
	dBit	1 Bit
	Addr	48 Bits

Table 5.1
Space requirement of key components in DMU

Consequently, address translation for DTiles can be completed in advance by the DMU. Table 5.2 presents the average number of rowTasks per DTile under different operations (generation and reading) for Neural Network models in the evaluation. Algorithm 3 outlines the concept of generating the sequence of DRAM accesses. With the support of the DMU, the update of NextBKRKChToCheck is reflected by the change of bits in Reg4. The pipeline execution of access request generation enables

NN Models	Generation	Read,DWS	Read, ARAI
AlexNet	1.71	2.21	1.20
DarkNet19	8.03	3.56	3.13
FaceRecognition	10.35	4.95	5.96
Resnet18	11.17	4.08	4.22
VGG16	21.68	11.38	2.98
ZFNet	2.34	2.22	1.39
MobileNet-v1	12.77	4.33	4.33

Table 5.2
Average number of rowTasks per DTile under different operations. Generation indicates *Generating DTile*. Read indicates *Reading DTile*. DWS and ARAI indicate that the different dataflow adopted. DWS and ARAI are explained in the evaluation 5.5.1.

the generation of a single DRAM block access request about each cycle. Additionally, off-chip memory accesses run in parallel with the generation of memory access requests. This implies that after a rowTask has been scheduled to the corresponding bank, the time overhead of the subsequent address translation can be disregarded for that rowTask. Coupled with an analysis of the time during inference execution, the time overhead caused by LAP can be effectively concealed. For instance, in AlexNet, the average time for on-chip computation is approximately 9000 cycles before generating a DTile, with a DTile having an average of 1.71 rowTasks. In modern SRAMs [71], the estimated latencies for read and write operations in a 64KB SRAM are estimated to be 1ns. Regarding the DRAM allocation for each rowTask, Algorithm 2 takes approximately 128 cycles for a 64-bank DRAM system, as illustrated in Table 5.3. Following the key steps outlined in Figure 5.8, the DRAM allocation for each rowTask consumes approximately 140 cycles at a clock frequency of 1GHz. Given the average of 1.71 rowTasks per DTile in AlexNet, the estimated time for DRAM allocation is roughly 250 cycles, significantly less than the 9000 cycles. Figure 5.8 indicates that during the DRAM allocation of a DTile, the rowTasks slated for write-back are scheduled to corresponding DRAM banks. With time allocated for reading input and weight elements, there is ample time available before DTile generation, making the time overhead introduced by DRAM allocation in LAP trivial. In contrast to DTile generation, reading a DTile incurs much less time overhead. Figure 5.9 outlines the steps to initialize address mapping information for reading. Concurrently, rowTasks designated for reading are scheduled to their respective DRAM banks. To facilitate the translation of information from the summarization in SRAM to MLTEntries for a DTile, the latency is estimated to be 5 cycles. As all DRAM mapping information for off-chip DTiles can be accommodated in DMU with 64KB on-chip SRAM, the translation latency for each rowTask is estimated to be 5 cycles. For AlexNet with ARAI, the average latency of address translation is estimated to be 11 cycles, representing a negligible portion of the overall estimated 1K cycles involved in DTile reading.

Limitation: It is important to note that, although we simplified the discussion of LAP by focusing on the DRAM allocation for individual DTiles, the partitioning of OFMAP into DTiles is a complex task that has not been addressed in this chapter. The above discussion of LAP was based on the assumption that the tiling of OFMAP is the same as that of IFMAP in the layer that reads the OFMAP as IFMAP. This assumption holds for many models [40, 62, 63, 64, 66, 67] but may not cover all scenarios, such as NN models with residual blocks [65]. In cases where the output of one layer serves as the input for multiple layers or a part of the input for another layer, a more complex approach to address the allocation and reading of DTiles is required. This could involve treating each output to be written back from on-chip memory as a special DTile to apply LAP Algorithm 2 for DRAM allocation. The reading process differs. Instead of simply reading complete rowTasks, DTile is read based on the correspondences of pixels between the layer generating intermediate data and the layer reading this data. Each input pixel of IFMAP tiles is mapped to a pixel of OFMAP or the initial IFMAP. Additionally, the partitioning of feature maps and the generation of rowTasks for DTiles are determined. With the given data layout format as NHWC, the aim rowTask and the offset inside rowTask can be obtained for each input pixel.

5.5 Evaluation

5.5.1 Evaluation Setup

To evaluate our designs, we developed an in-house cycle-accurate NNA simulator integrated with the DRAM simulator DRAMSim3 [72]. This setup allows for precise simulation of off-chip memory accesses. The simulator encompasses modeling the execution of the System Array, on-chip memory, DMA Unit, off-chip memory, data

Systolic Array	32x32 PEs, 1GHz
On-chip Memory	2MB Scratchpad Memory, double buffering, NHWC *im2col* format
DMA Unit	1 pair of send/receive ports, 1ns SRAM read/write latency [71] inside DMU
Off-chip Memory	2400 MHz DDR4 [72], robabgrachco, FCFS ro=32768, ba=4, bg=4, ra=2, ch=2, co=128 1KB page size, 8B block size

Table 5.3
System parameters of NNA evaluation

ALX	AlexNet [62]. Conv×5, FC×3.
DRK	DarkNet19 [63]. Conv×19, FC×1.
FR	FaceRecognition [64]. Conv×5
RES	Resnet18 [65]. Conv×1, Res×8, FC×1
VGG	VGG16 [66]. Conv×13, FC×3
ZF	ZFNet [67]. Conv×5, FC×3
MOB	MobileNet-v1 [40]. Conv×14, ConvDW×13, FC×2

Table 5.4
Summarization of NN workloads. Conv: the standard convolution layer. ConvDW: the depthwise convolution layer. Res: the residual block containing Conv and skip connection. FC: the fully connected layer.

movements directed by the dataflows, and the proposed DTile Manager Unit, which is described in detail in section 5.4. Configuration parameters for the simulation are provided in Table 5.3. The evaluated DNN models are listed in Table 5.4.

In our evaluation, we adopt the NHWC data layout format [25, 43, 45]. The baseline dataflow is Weight Stationary with fused operators implemented using DNNFusion [1], against which we compare ARAI. The default DRAM mapping policy is based on ROMANet [14]. For clarity in the subsequent discussion, we introduce the following abbreviations to refer to different designs:

† DWS: Weight Stationary with DNNFusion optimized Computational Graph, in which NN operators are classified into five mapping types. The fusion opportunities are exploited based on the mapping types. For example, the Batch-Normalization is fused with the convolution operation, while two convolution operations will not be fused.

† ROMA: DRAM mapping policy proposed by ROMANet.

† DWS_ROMA: Combination of DWS and ROMA, serving as the baseline for evaluation.

† ARAI_ROMA: Combination of ARAI and ROMA, indicating the application of our dataflow optimization, ARAI.

† DWS_LAP: Combination of DWS and LAP, signifying that the DRAM placement of DTiles is managed by LAP.

† ARAI_LAP: Combination of ARAI and LAP, indicating the simultaneous application of both designs.

Our evaluation employs the following metrics:

† *Inference time*: This metric quantifies the time required to complete the simulation of NN inference, serving as an indicator of the overall system performance.

† *DRAM Read Traffic*: Signifies the number of DRAM read accesses.

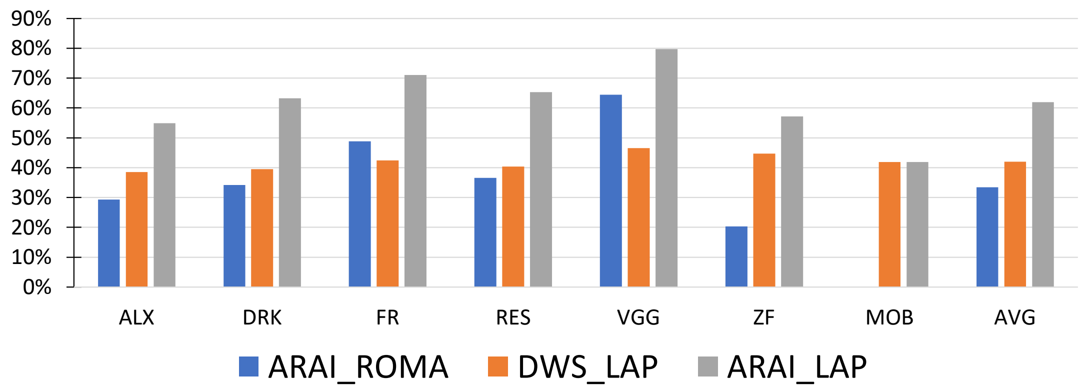


Figure 5.10: Reduction in *Inference Time*. Baseline is DWS_ROMA.

† *DRAM Bandwidth*: Represents the working bandwidth of DRAM when being accessed.

5.5.2 Improvement in System Performance

Figure 5.10 illustrates that all the evaluated designs outperform DWS_ROMA in terms of *Inference Time*. Specifically, ARAI enhances system performance by an average of 33.37%, as demonstrated by ARAI_ROMA. LAP, on the other hand, enhances system performance by an average of 42.00%, as evidenced by DWS_LAP. The combined application of ARAI and LAP results in a 61.90% average increase in system performance, as shown by ARAI_LAP.

The performance improvement achieved by ARAI can be attributed to its reduction in *DRAM Read Traffic*. This reduction stems from the permutation of loop orders as depicted in Figure 5.4, which increases the reuse of on-chip input data. This, in turn, reduces the need for repeated data transfers between on-chip and off-chip memory.

The performance gain from LAP is attributed to its ability to enhance *DRAM Bandwidth* by improving both the channel-level parallelism and the bank-level parallelism of DRAM.

Upon closer examination, it becomes evident that the combined improvement in system performance achieved by applying ARAI and LAP separately surpasses the performance gain obtained when both optimizations are applied together. This outcome can be attributed to the fact that ARAI's focus on reducing off-chip memory access diminishes the benefits of LAP.

Furthermore, it's noteworthy that ARAI has no impact on MobileNet [40], in which a standard convolution is factorized into a depthwise convolution and a 1×1 pointwise convolution. The computation of both components cannot be optimized by ARAI.

5.5.3 Reduction in DRAM Read Traffic

Figure 5.11 demonstrates the percentage reduction in *DRAM Read Traffic* when ARAI is applied. On average, ARAI achieves a 41.12% reduction in DRAM read

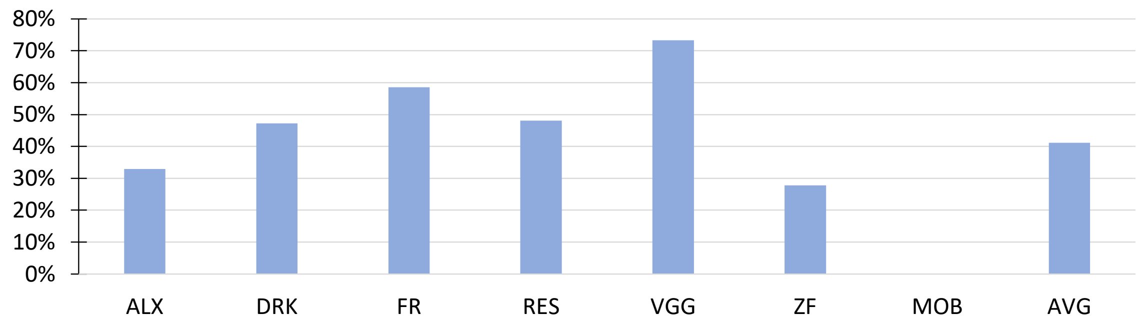


Figure 5.11: Reduction in *DRAM Read Traffic*.

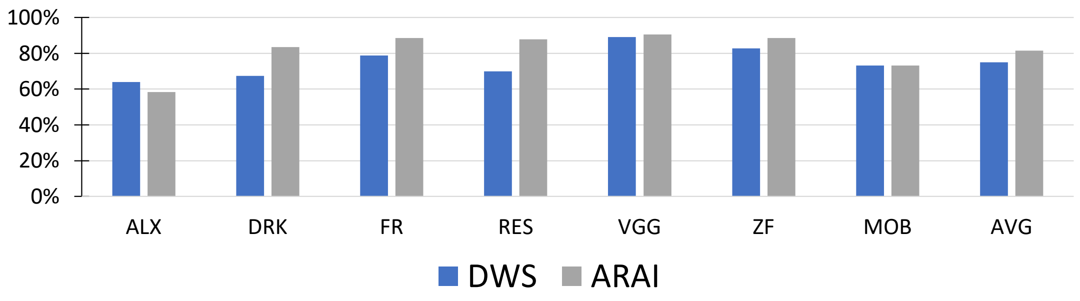


Figure 5.12: Increased *DRAM Bandwidth* by LAP. Baseline is ROMA

accesses during NN inference. This outcome proves ARAI's effectiveness in eliminating redundant accesses to IFMAP tiles in DRAM, thereby enhancing system performance. This is consistent with the trend observed in Figure 5.10. Notably, there is no reduction observed for MobileNet, and this finding aligns with the aforementioned results.

5.5.4 Increased Bandwidth of Off-chip Memory

In Figure 5.12, we illustrate the percentage increase in *DRAM Bandwidth* achieved by LAP compared to ROMA. The bars, DWS and ARAI, represent the enhancements in

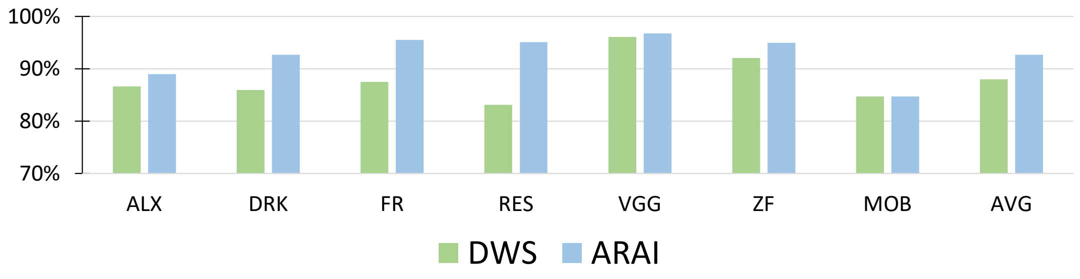


Figure 5.13: Increased Channel-Level Parallelism of DRAM. Baseline is ROMA.

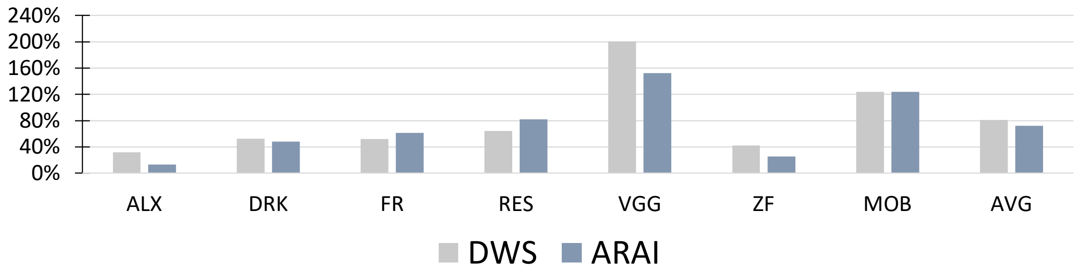


Figure 5.14: Increased Bank-Level Parallelism of DRAM. Baseline is ROMA.

DRAM Bandwidth when comparing DWS_LAP with DWS_ROMA and ARAI_LAP with ARAI_ROMA.

With the DWS dataflow, LAP demonstrates an average improvement of 75.04%, and with ARAI optimization, it exhibits a significant average improvement of 81.55%. Notably, compared with DWS_LAP, the reported value of *DRAM Bandwidth* by DRAMSim3 remains lower in ARAI_LAP. A comparison between these scenarios suggests that LAP's impact is somewhat enhanced in ARAI_LAP. This enhancement can be attributed to ARAI's effective improvement in on-chip input data reuse, which reduces the need for repeated access to the same DRAM positions. While DWS involves more off-chip memory access, repeated access to the same positions can lead to concentrated concurrent access to the same DRAM channel.

The more off-chip memory access in DWS results in higher *DRAM Bandwidth* values in both DWS_ROMA and DWS_LAP compared to ARAI_ROMA and ARAI_LAP. However, the improvement achieved by LAP is limited due to the repeated access to the same DRAM positions. The increased channel-level parallelism, as depicted in Figure 5.13, leads to a lower average improvement of 87.98% for LAP with DWS. In contrast, ARAI_LAP demonstrates an average increase of 92.66% in the channel-level parallelism of DRAM compared to ROMA_LAP. However, in the case of ALX, the increase in *DRAM Bandwidth* is less when the dataflow is ARAI. This discrepancy is attributed to the influence of bank-level parallelism in DRAM.

As illustrated in Figure 5.14, LAP achieves a substantial improvement in the bank-level parallelism of DRAM. In comparison to ROMA, LAP's improvement in bank-level parallelism is 80.93% for DWS and 72.29% for ARAI. For ALX, LAP increases bank-level parallelism by 31.82% for DWS and 12.95% for ARAI. The reduced bank-level parallelism in the case of ARAI is a result of the reduced off-chip memory access in ARAI. While LAP demonstrates higher increased bank-level parallelism for FR and RES in the ARAI dataflow, the number of concurrent working banks is still less compared to the DWS dataflow.

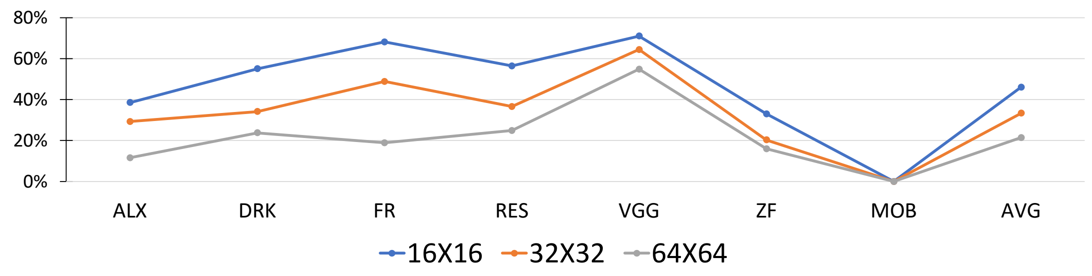


Figure 5.15: Reductions in *Inference Time* with different sizes of Systolic Array

5.5.5 Sensitivity Analysis

The effectiveness of ARAI is contingent upon the percentage of wide layers during neural network inference, which is directly influenced by the matrix size of the systolic array (SA). Figure 5.15 illustrates the impact of different SA sizes on the reduction of *Inference Time*. It is evident that as the SA size increases, the reduction achieved by ARAI decreases. Specifically, when the SA is equipped with a 16×16 matrix of processing elements (PEs), the average reduction in *Inference Time* amounts to 46.03%. In the case of a 32×32 PE matrix, the average reduction is 33.37%, and with a 64×64 PE matrix, the average reduction dwindles to 21.43%.

5.5.6 Discussion

As the above evaluation shows, ARAI can effectively increase the system performance for neural network accelerators whose size of the systolic array is small. A typical application scenario is on edge devices. Compared with ARAI, the application scenario of LAP is more general. However, as discussed in section 5.4, the layer partition could be a challenge. Additionally, when LAP is applied on edge devices, the introduced energy consumption should be considered, which is our future work.

5.6 Summary

This work proposes two optimizations to mitigate the off-chip memory bottleneck for DNN inference on edge NNAs. We observe the execution of wide layers on edge NNAs causes significant performance overhead and reveal that the execution of wide layer introduces repeated accesses to IFMAP tiles in off-chip memory. To address this issue, we propose ARAI to remove redundant accesses to IFMAP by permuting the iteration order of the convolution computation. To further boost the memory bandwidth, we introduce the load-aware tile placement on off-chip memory, LAP, that reduces intra/inter contentions caused by concurrent accesses from multiple tiles and improves the off-chip memory device parallelism during access. The evaluation shows that ARAI reduces inference latency by 33.37% on average and LAP reduces 42.00% on average. The combination of ARAI and LAP achieves an average 61.90% performance improvement over the prior works.

Chapter 6

Conclusion

This dissertation presents multiple memory optimizations for high-performance computing systems, focusing on the non-volatile memory systems and the neural network accelerators.

Chapter 3 and Chapter 4 summarize our previous works [73, 74, 75] on computing systems with Non-Volatile Memory, where logging is introduced to ensure crash consistency, resulting in additional memory overhead. We proposed TSTE and VADR to eliminate unnecessary log operations, LALEA to reduce log persistence time, and BLOM in ADR to address intra-record ordering issues. Compared with STOA works at that time, all these designs achieved significant improvement in the system throughput.

However, with the development of NVM[76], our focus shifted to the memory system of the neural network accelerator, where we found that off-chip memory access is a performance bottleneck in edge devices. To minimize the impact caused by off-chip memory on the inference time, we introduce two designs, ARAI and LAP, in Chapter 5. Our evaluation shows that ARAI effectively reduces redundant accesses to IFMAP in off-chip memory by reordering the loops in the standard convolution operation. LAP mitigates intra/inter access contentions from data tiles and enhances the access parallelism for off-chip memory.

References

[1] W. Niu, J. Guan, Y. Wang, G. Agrawal, and B. Ren, "Dnnfusion: Accelerating deep neural networks execution with advanced operator fusion," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 883–898. [Online]. Available: https://doi.org/10.1145/3453483.3454083

[2] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "ATOM: atomic durability in non-volatile memory through hardware logging," in *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017*, 2017, pp. 361–372.

[3] K. Doshi, E. Giles, and P. Varman, "Atomic persistence for scm with a non-intrusive backend controller," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 77–89.

[4] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A flexible and fast software supported hardware logging approach for nvm," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17, 2017, pp. 178–190.

[5] S. Shin, J. Tuck, and Y. Solihin, "Hiding the long latency of persist barriers using speculative execution," in *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*, 2017, pp. 175–186.

[6] J. Jeong, C. H. Park, J. Huh, and S. Maeng, "Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory," in *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, 2018, pp. 520–532.

[7] A. D. Siddharth Gupta and B. Falsafi, "Distributed logless atomic durability with persistent memory," in *the Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '19, 2019.

[8] D. Mulnixl. Intel xeon processor d product family technical overview. https://software.intel.com/en-us/articles/intel-xeon-processor-dproduct-family-technical-overview/.

[9] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao:

A small-footprint high-throughput accelerator for ubiquitous machine-learning," *SIGARCH Comput. Archit. News*, vol. 42, no. 1, p. 269–284, feb 2014.

[10] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 367–379.

[11] S. Li, X. Xie, J. Huang, Y. Zhang, X. Hu, and H. Yang, "Smartshuttle: Optimizing off-chip memory accesses for deep learning accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[12] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 161–170.

[13] A. Samajdar, J. M. Joseph, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "A systematic methodology for characterizing scalability of dnn accelerators using scale-sim," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020, pp. 58–68.

[14] R. V. W. Putra, M. A. Hanif, and M. Shafique, "Romanet: Fine-grained reuse-driven off-chip memory access management and data organization for deep neural network accelerators," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 4, pp. 702–715, 2021.

[15] V. Sze, Y. Chen, T. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *CoRR*, vol. abs/1703.09039, 2017.

[16] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Weight-stationary architecture for deep learning," in *Proceedings of the 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture.* IEEE Press, 2016, pp. 13–24.

[17] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox,

and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, p. 1–12, jun 2017. [Online]. Available: https://doi.org/10.1145/3140659.3080246

[18] T. Norrie, N. Patil, D. H. Yoon, G. Kurian, S. Li, J. Laudon, C. Young, N. Jouppi, and D. Patterson, "The design process for google's training chips: Tpuv2 and tpuv3," *IEEE Micro*, vol. 41, no. 2, pp. 56–63, 2021.

[19] N. Corporation, "Nvidia, NVDLA Open Source Project," http://nvdla.org/69, 76,92,94,96,97,113,114, 2017.

[20] "Nvidia T4 GPU," https://www.nvidia.com/en-us/data-center/t4-tesla/, Insert Year, accessed: Insert Date.

[21] "Nvidia V100 GPU," https://www.nvidia.com/en-us/data-center/v100/, Insert Year, accessed: Insert Date.

[22] Y. S. Shao, J. Clemons, R. Venkatesan, B. Zimmer, M. Fojtik, N. Jiang, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, S. G. Tell, Y. Zhang, W. J. Dally, J. Emer, C. T. Gray, B. Khailany, and S. W. Keckler, "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 14–27. [Online]. Available: https://doi.org/10.1145/3352460.3358302

[23] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *2015*

ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA), 2015, pp. 92–104.

[24] H. Yang, C. Yuan, J. Xing, and W. Hu, "Scnn: Sequential convolutional neural network for human action recognition in videos," in *2017 IEEE International Conference on Image Processing (ICIP)*, 2017, pp. 355–359.

[25] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. USA: USENIX Association, 2016, p. 265–283.

[26] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 579–594.

[27] Z. Jia, T. Yu, H. Zhang, Y. Wang, and H. Yang, "Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2018, pp. 798–807.

[28] S. Tewari, A. Kumar, and K. Paul, “Bus width aware off-chip memory access minimization for cnn accelerators,” in *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2020, pp. 240–245.

[29] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, “Dadiannao: A machine-learning supercomputer,” in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 609–622.

[30] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, “Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory,” in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 380–392.

[31] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, “Tetris: Scalable and efficient neural network acceleration with 3d memory,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 751–764. [Online]. Available: https://doi.org/10.1145/3037697.3037702

[32] B. Hyun, Y. Kwon, Y. Choi, J. Kim, and M. Rhu, “Neummu: Architectural support for efficient address translations in neural processing units,” in *Proceedings of the Twenty-Fifth International Conference on Architectural*

Support for Programming Languages and Operating Systems, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1109–1124. [Online]. Available: https://doi.org/10.1145/3373376.3378494

[33] Y. Yang, J. S. Emer, and D. Sanchez, "Isosceles: Accelerating sparse cnns through inter-layer pipelining," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 598–610.

[34] T. Nguyen and D. Wentzlaff, "Picl: A software-transparent, persistent cache log for nonvolatile main memory," in *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, 2018, pp. 507–519.

[35] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17, 2017, pp. 135–148.

[36] J. Yang, J. Kim, and etc, "An empirical guide to the behavior and use of scalable persistent memory," in *18th USENIX Conference on File and Storage Technologies, FAST 2020, Santa Clara, CA, USA, February 24-27, 2020*, 2020.

[37] I. Corporation, "Intel optane dc persistent memory sampling today revenue delivery 2018," 2018. [Online]. Available: https://www.servethehome.com/intel-optane-dc-persistent-memory-sampling-today-revenue-delivery-2018

[38] J. R. S. Liu, A. Kolli and S. Khan, "Crash consistency in encrypted non- volatile main memory systems," in *IEEE International Symposium on High Performance Computer Architecture*, ser. HPCA '18, 2018.

[39] J. T. M. Alshboul and Y. Solihin, "Lazy persistency: A high-performing and write-efficient software persistency technique," in *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, ser. ISCA '18, 2018.

[40] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," 2017.

[41] N. P. Jouppi, D. Hyun Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma, T. Norrie, N. Patil, S. Prasad, C. Young, Z. Zhou, and D. Patterson, "Ten lessons from three generations shaped google's tpuv4i : Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 1–14.

[42] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, 2013, pp. 13–19.

[43] Y. Zhou, M. Yang, C. Guo, J. Leng, Y. Liang, Q. Chen, M. Guo, and Y. Zhu, "Characterizing and demystifying the implicit convolution algorithm on commercial matrix-multiplication accelerators," in *2021 IEEE International*

Symposium on Workload Characterization (IISWC). Los Alamitos, CA, USA: IEEE Computer Society, nov 2021, pp. 214–225. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/IISWC53511.2021.00029

[44] S. Lym, D. Lee, M. Ox27;Connor, N. Chatterjee, and M. Erez, "Delta: Gpu performance model for deep learning applications with in-depth memory system traffic analysis," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).* Los Alamitos, CA, USA: IEEE Computer Society, mar 2019, pp. 293–303. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ISPASS.2019.00041

[45] N. Corporation. (2023) Convolutional layers user's guide. Oct.2023. [Online]. Available: https://docs.nvidia.com/deeplearning/performance/dl-performance-convolutional/index.html

[46] S. Liu, K. Seemakhupt, G. Pekhimenko, A. Kolli, and S. Khan, "Janus: Optimizing memory and storage support for non-volatile memory systems," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19, 2019.

[47] P. Zuo, Y. Hua, M. Zhao, W. Zhou, and Y. Guo, "Improving the performance and endurance of encrypted non-volatile main memory through deduplicating writes," in *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, 2018, pp. 442–454.

[48] P. Mahapatra, M. D. Hill, and M. M. Swift, "Don't persist all : Efficient persistent data structures," *CoRR*, vol. abs/1905.13011, 2019.

[49] R. Balasubramonian, B. A. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *TACO*, pp. 14:1–14:25, 2017.

[50] Champsim. https://github.com/ChampSim/.

[51] Dramsim. https://github.com/umd-memsys/DRAMSim2.

[52] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05, 2005.

[53] S. O. J. H. Ahn, S. Li and N. P. Jouppi, "Mcsima+: a manycore simulator with application-level+ simulation and detailed microarchitecture modeling," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, ser. pp. 74-85, 2013.

[54] D. H. Y. Y. X. J. Zhao, S. Li and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *in Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, ser. MICRO'2013, 2013.

[55] J. Z. S. L. A. B. Y. X. X. Hu, M. Ogleari, “Persistence parallelism optimization: A holistic approach from memory bus to rdma network,” in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchit.*, ser. pp. 494-506, 2018.

[56] J. Z. M. A. Ogleari, E. L. Miller, “Steal but no force: Efficient hardware undo+redo logging for persistent memory systems,” in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, ser. HPCA ’18, 2018.

[57] A. W. Markku manner Vilho Raatikka Simo Neuvonen. Tatp telecommunication application transaction processing (benchmark description). http://tatpbenchmark.sourceforge.net/TATP-Description.pdf.

[58] Transaction processing performance council(tpc), tpc-c. http://www.tpc.org/tpcc/default.asp.

[59] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, “A durable and energy efficient main memory using phase change memory technology,” in *ISCA*. ACM, 2009, pp. 14–23.

[60] P. Zuo, Y. Hua, and Y. Xie, “Supermem: Enabling application-transparent secure persistent memory with low overheads,” in *MICRO*. ACM, 2019, pp. 479–492.

[61] S. Liu, A. Kolli, J. Ren, and S. Khan, “Crash consistency in encrypted non-volatile main memory systems,” in *HPCA*, 2018, pp. 310–323.

[62] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, p. 84–90, may 2017. [Online]. Available: https://doi.org/10.1145/3065386

[63] J. Redmon and A. Farhadi, "YOLO9000: better, faster, stronger," *CoRR*, vol. abs/1612.08242, 2016. [Online]. Available: http://arxiv.org/abs/1612.08242

[64] B. Research, "Deepbench: Quantifying representational similarity across deep learning architectures," https://github.com/baidu-research/DeepBench, 2020.

[65] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015. [Online]. Available: https://arxiv.org/abs/1512.03385

[66] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2015.

[67] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," *CoRR*, vol. abs/1311.2901, 2013. [Online]. Available: http://arxiv.org/abs/1311.2901

[68] S. Cho, H. Choi, E. Park, H. Shin, and S. Yoo, "Mcdram v2: In-dynamic random access memory systolic array accelerator to address the large model problem in deep neural networks on the edge," *IEEE Access*, vol. 8, pp. 135 223–135 243, 2020.

[69] U. S. Solangi, M. Ibtesam, M. A. Ansari, J. Kim, and S. Park, “Test architecture for systolic array of edge-based ai accelerator,” *IEEE Access*, vol. 9, pp. 96 700–96 710, 2021.

[70] Google coral edge tpu explained in depth. Accessed on: Oct. 2023. [Online]. Available: https://qengineering.eu/google-corals-tpu-explained.html

[71] M. Imani, S. Patil, and T. Rosing, “Low power data-aware stt-ram based hybrid cache architecture,” in *2016 17th International Symposium on Quality Electronic Design (ISQED)*, 2016, pp. 88–94.

[72] A. Li and et al., “Dramsim3: A cycle accurate memory system simulator,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 406–418.

[73] Z. Lu, J. Yue, Y. Deng, and Y. Zhu, “Improving the performance of nvm crash consistency under multicore,” in *2020 IEEE 38th International Conference on Computer Design (ICCD)*, 2020, pp. 561–564.

[74] ——, “Efficient nvm crash consistency by mitigating resource contention,” in *2021 IEEE International Conference on Networking, Architecture and Storage (NAS)*, 2021, pp. 1–8.

[75] ——, “Accelerate hardware logging for efficient crash consistency in persistent memory,” in *2022 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2022, pp. 388–393.

[76] I. Corporation, "eadr: New opportunities for persistent memory applications," *Intel Developer Zone*, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html

Appendix A

ACM Copyright Permission

The publisher of Paper [75] is ACM. ACM grants gratis permission for individual digital or hard copies made without fee for use in academic classrooms and for use by individuals in personal research and study. Further reproduction or distribution requires explicit permission and possibly a fee.

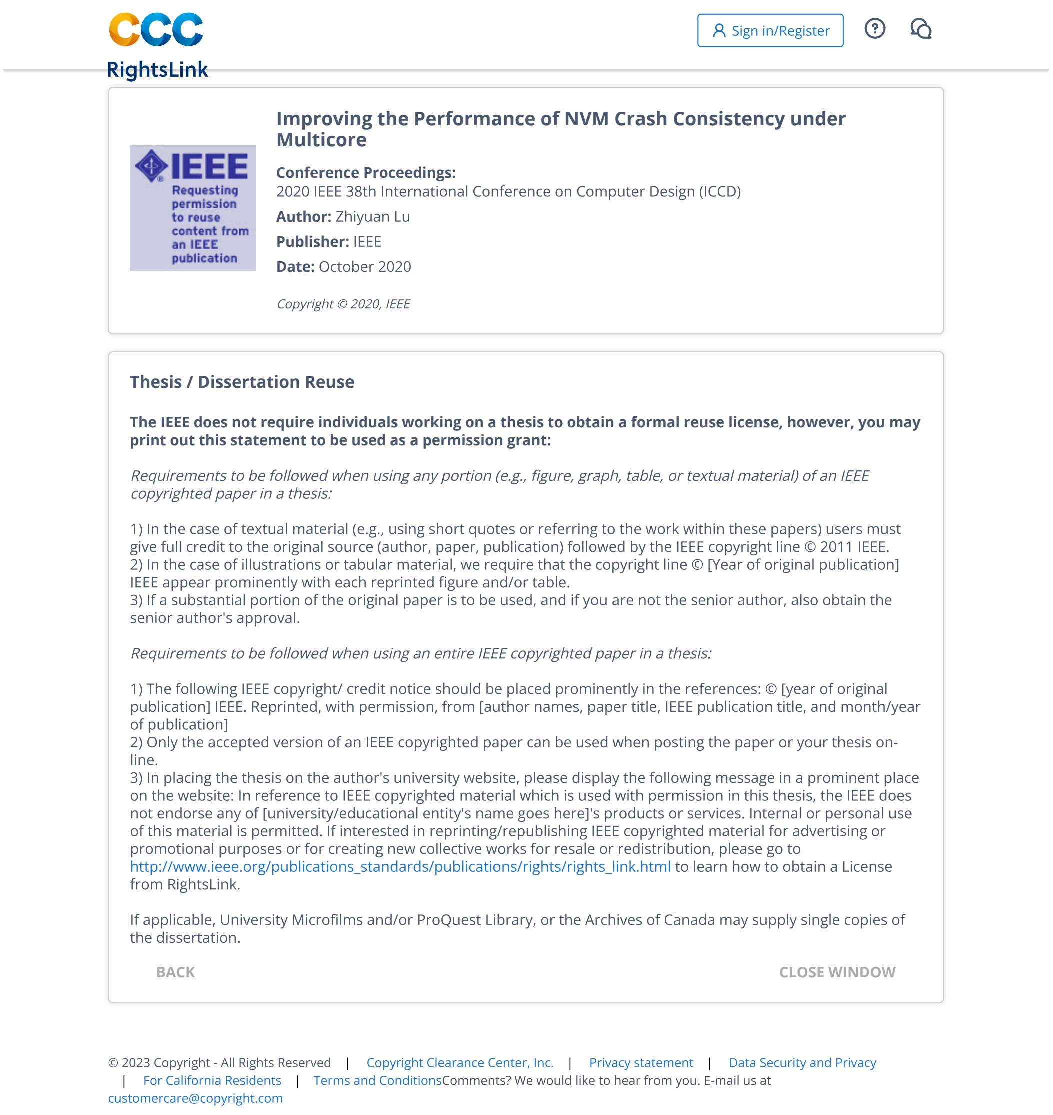
All copies should carry the original citation, the appropriate copyright and notice of permission on the first page or initial screen of the document. (See §2.2 Copyright Notice)

Most permission requests should go through ACM's automated rights system available in the ACM Digital Library and pointed to by permissions@acm.org. Requests that cannot be handled through the online system will take longer to resolve: requestors may expect a response to their inquiry within seven business days. More info refer:

https://cacm.acm.org/help/copyright-policy

Appendix B

IEEE Copyright Permission


RightsLink
Sign in/Register

Improving the Performance of NVM Crash Consistency under Multicore

IEEE
Requesting permission to reuse content from an IEEE publication

Conference Proceedings:
2020 IEEE 38th International Conference on Computer Design (ICCD)
Author: Zhiyuan Lu
Publisher: IEEE
Date: October 2020

Copyright © 2020, IEEE

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

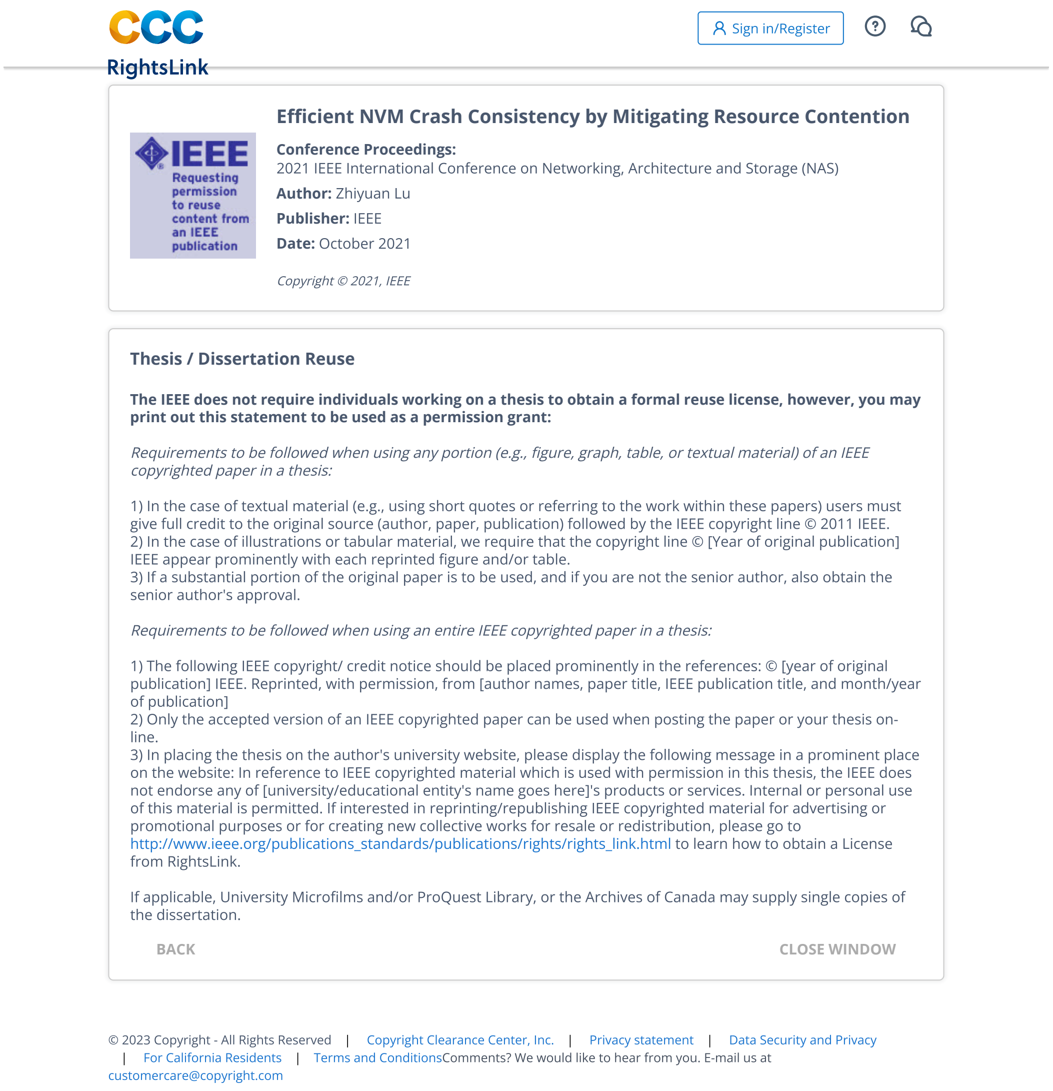
Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.


RightsLink
Sign in/Register
IEEE
Requesting permission to reuse content from an IEEE publication

Efficient NVM Crash Consistency by Mitigating Resource Contention

Conference Proceedings:
2021 IEEE International Conference on Networking, Architecture and Storage (NAS)

Author: Zhiyuan Lu

Publisher: IEEE

Date: October 2021

Copyright © 2021, IEEE

Thesis / Dissertation Reuse

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.
2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.
3) If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]
2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.
3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.