Data-Centric and Data-Aware Frameworks for Fundamentally Efficient Data Handling in Modern Computing Systems

by

Nastaran Hajinazar

M.Sc., Sharif University of Technology, 2011
B.Sc., Shahid Chamran University, 2008

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

in the
School of Computing Science
Faculty of Applied Sciences

© Nastaran Hajinazar 2021
SIMON FRASER UNIVERSITY
Summer 2021

Copyright in this work is held by the author. Please ensure that any reproduction or re-use is done in accordance with the relevant national copyright legislation.

Declaration of Committee

Name:	**Nastaran Hajinazar**
Degree:	**Doctor of Philosophy**
Thesis title:	**Data-Centric and Data-Aware Frameworks for Fundamentally Efficient Data Handling in Modern Computing Systems**
Committee:	**Chair:** Zhenman Fang Assistant Professor, Engineering Science

Onur Mutlu
Co-Supervisor
Professor, Computing Science
ETH Zurich

Arrvindh Shriraman
Co-Supervisor
Associate Professor, Computing Science

Saugata Ghose
Committee Member
Assistant Professor, Computer Science
University of Illinois at Urbana–Champaign

Vivek Seshadri
Committee Member
Senior Researcher
Microsoft Research India

Alaa Alameldeen
Examiner
Associate Professor, Computing Science

Myoungsoo Jung
External Examiner
Associate Professor, Electrical Engineering
Korea Advanced Institute of Technology (KAIST)

Abstract

There is an explosive growth in the size of the input and/or intermediate data used and generated by modern and emerging applications. Unfortunately, modern computing systems are not capable of handling large amounts of data efficiently. Major concepts and components (e.g., the virtual memory system) and predominant execution models (e.g., the processor-centric execution model) used in almost all computing systems are designed without having modern applications' overwhelming data demand in mind. As a result, accessing, moving, and processing large amounts of data faces important challenges in today's systems, making data a first-class concern and a prime performance and energy bottleneck in such systems. This thesis studies the root cause of inefficiency in modern computing systems when handling modern applications' data demand, and aims to fundamentally address such inefficiencies, with a focus on two directions.

First, we design a new framework that aids the widespread adoption of processing-using-DRAM, a data-centric computation paradigm that improves the overall performance and efficiency of the system when computing large amounts of data by minimizing the cost of data movement and enabling computation where the data resides. To this end, we introduce SIMDRAM, an end-to-end processing-using-DRAM framework that (1) efficiently computes complex operations required by modern data intensive applications, and (2) provides the ability to implement new arbitrary operations as required, all in an in-DRAM massively-parallel Single Instruction Multiple Data (SIMD) substrate that requires minimal changes to the DRAM (Dynamic Random Access Memory) architecture.

Second, we design a new, more scalable virtual memory framework that (1) eliminates the inefficiencies of the conventional virtual memory frameworks when handling the high memory demand in modern applications, and (2) is built from the ground up to understand, convey, and exploit data properties, to create opportunities for performance and efficiency improvements. To this end, we introduce the Virtual Block Interface (VBI), a novel virtual memory framework that (1) efficiently handles modern applications' high data demand, (2) conveys properties of different pieces of program data (e.g., data structures) to the hardware and exploits this knowledge for performance and efficiency optimizations, (3) better extracts performance from the wide variety of new system configurations that are designed to process large amounts of data (e.g., hybrid memory systems), and (4) provides all the key features of the conventional virtual memory frameworks, at low overhead.

Keywords: Data-centric; Data-aware; Efficient data handling; Virtual memory; Computing-using-DRAM; Processing-in-memory

Dedication

Dedicated to

my beloved parents, Ezzatollah and Homa,
my wonderful brothers, Samad and Siavash,
my lovely husband, Amir,
and my most precious son, Nick Ray.

Acknowledgements

Pursuing a PhD was a significant source of learning, inspiration and growth for me, to which many people have contributed in different ways. This is a humble attempt to thank them for their contribution and for helping me become who I am today.

First and foremost, I would like to express my deep and sincere gratitude to my advisor, Prof. Onur Mutlu, who believed in me, even when I did not. His unwavering support, despite the tough initial years, was my prime source of courage towards the completion of my PhD. Onur generously provided me with invaluable guidance, exceptional opportunities, incredible resources, and more importantly, extraordinary freedom to carry out my research. I also thank him for teaching me how to think critically, write thoroughly, speak clearly, and perform impactful research. His influence in shaping me certainly goes above and beyond this dissertation and extends to countless real-life lessons that I have learned from him.

I would like to thank my advisor, Prof. Arrvindh Shriraman, for all his help and support. I thank him for allowing me to find and follow the research direction that interested me. I also thank him for being open to my collaborations with students and researchers from other institutions.

I would also like to thank the members of my supervisory committee, Prof. Saugata Ghose and Dr. Vivek Seshadri. I thank Prof. Ghose for providing me with incredible technical and moral support and countless pieces of critical advice throughout my PhD journey. He helped me stay focused and navigate through my research with a clear and determined mind. My earnest thanks to Dr. Seshadri for teaching me how to find the right research problem to work on and how to perform quality research. I thank him for all his help and support through the times that I needed it the most.

During my PhD, I had the chance to work alongside many wonderful fellow graduate students at the SAFARI research group whom I am grateful to. My great friend, Giray Yaglikci was always amazingly kind and selfless in helping me and listening to me during the times when graduate school felt dark and lonely. I would like to thank him for all his support, as well as the hot cups of tea and delicious food that rescued me with when working late at school. Geraldo De Oliveira was my PIM guru, whose kind presence as well as his amazing home-made cakes and cookies brightened up my days at ETH. Minesh Patel has been a great friend with whom I have also enjoyed many research collaborations and real-life conversations. Juan Gómez Luna was my mentor and collaborator who assisted me in many

ways. Kevin Hsieh was an extraordinary friend and mentor who was there to listen to me and support me when I was in need. Rachata Ausavarungnirun was a good friend who also taught me a great deal about how to conduct thorough simulations and evaluations. Damla Senol Cali was my sweet and kind friend and collaborator who was always there to support me and celebrate my achievements with me. Gagandeep Singh was a wonderful collaborator and friend who always amused me with his great sense of humor. I thank Can Firtina for his delightful friendship as well as the high-tech Tesla rides. I would also like to thank many other SAFARI members for their friendship and collaboration: Jeremie Kim, Mohammed Alser, Jawad Haj-Yahya, Lois Orosa, Jisung Park, Hasan Hassan, Tracy Ewen, and Christian Rossi.

I would also like to thank many wonderful people who made the long and at times tough journey of pursuing PhD easier for me, and I am lucky to call them my friends: Rajesh Rao, Sogol Barazandegan, Nasibeh Teimouri, Amir Shabani, and Ahmed Hamza.

And last, but not least, I would like to express my profound gratitude to my family for their unconditional and continued love and support. I am most grateful to my parents, Ezzatollah Hajinazar and Homa Golabian, for enabling me to pursue my passion for learning, and for being by my side at each and every step of this journey. I thank my mother for selflessly caring for me and for teaching me by example, the importance of hard work, perseverance, and integrity. I thank my father for always believing in me and for teaching me the important lesson that nothing is impossible if you fight for it. I thank my lovely brothers, Samad and Siavash Hajinazar, for their immense and unwavering love and support which has been a sublime source of motivation for me. I would like to give my heartfelt thanks to my wonderful husband, Amir Pourmand, for being my rock, my number one supporter, and my safe place. This thesis would have been impossible without him by my side. Finally, I thank my most precious son, Nick Ray, whose little kicks kept me motivated during the writing of this dissertation. Words will never be able to describe my love for him. He will always be the reason behind my smile and I am eternally proud and grateful to be his mother.

Contents

Declaration of Committee **ii**

Abstract **iii**

Dedication **v**

Acknowledgements **vi**

Table of Contents **viii**

List of Tables **xi**

List of Figures **xii**

1 Introduction **1**

1.1 Motivation: Existing Computing Systems Are Designed Without Having Modern Applications' Data Demand in Mind . . . 1

1.2 Our Approach: Data-Centric and Data-Aware Architectures for Fundamentally Efficient Data Handling . . . 3

1.2.1 Thesis Statement . . . 4

1.3 Overview of Research . . . 4

1.3.1 SIMDRAM: A Data-Centric Framework for Bit-Serial SIMD Processing using DRAM (Chapter 2) . . . 5

1.3.2 The Virtual Block Interface: A Flexible Data-Aware Alternative to the Conventional Virtual Memory Framework (Chapter 3) . . . 6

1.4 Contributions . . . 8

2 SIMDRAM **9**

2.1 Background . . . 12

2.1.1 DRAM Basics . . . 12

2.1.2 Processing-using-DRAM . . . 13

2.2 SIMDRAM Overview . . . 15

2.2.1 Subarray Organization . . . 15

2.2.2 Framework Overview . . . 16
2.2.3 Integrating SIMDRAM in a System . . . 17
2.3 SIMDRAM Framework . . . 18
2.3.1 Step 1: Efficient MAJ/NOT Implementation . . . 18
2.3.2 Step 2: µProgram Generation . . . 19
2.3.3 Step 3: Operation Execution . . . 23
2.3.4 Supported Operations . . . 25
2.4 System Integration of SIMDRAM . . . 25
2.4.1 Data Layout . . . 25
2.4.2 ISA Extensions and Programming Interface . . . 28
2.4.3 Handling Page Faults, Address Translation, Coherence, and Interrupts 30
2.4.4 Handling Limited Subarray Size . . . 31
2.4.5 Security Implications . . . 31
2.4.6 SIMDRAM Limitations . . . 31
2.5 Methodology . . . 32
2.6 Evaluation . . . 33
2.6.1 Throughput Analysis . . . 33
2.6.2 Energy Analysis . . . 34
2.6.3 Effect on Real-World Kernels . . . 35
2.6.4 Comparison to DualityCache . . . 36
2.6.5 Reliability . . . 38
2.6.6 Data Movement Overhead . . . 39
2.6.7 Data Transposition Overhead . . . 40
2.6.8 Area Overhead . . . 40
2.7 Related Work . . . 41
2.8 Summary and Contributions . . . 43

3 The Virtual Block Interface 44
3.1 Design Principles . . . 47
3.2 Virtual Block Interface: Overview . . . 48
3.2.1 VBI Address Space . . . 49
3.2.2 VBI Access Permissions . . . 50
3.2.3 Memory Translation Layer . . . 50
3.2.4 Implementing Key OS Functionalities . . . 50
3.2.5 Optimizations Supported by VBI . . . 51
3.3 VBI: Detailed Design . . . 53
3.3.1 Architectural Components . . . 53
3.3.2 Life Cycle of Allocated Memory . . . 55
3.3.3 CVT Cache . . . 57

3.3.4 Processor, OS, and Process Interactions 57
3.3.5 Memory Translation Layer 59
3.4 Allocation and Translation Optimizations 61
3.4.1 Delayed Physical Memory Allocation 61
3.4.2 Flexible Address Translation Structures 61
3.4.3 Early Reservation of Physical Memory 62
3.5 VBI in Other System Architectures 63
3.5.1 Supporting Virtual Machines 63
3.5.2 Supporting Multi-Node Systems 63
3.6 Evaluation 64
3.6.1 Methodology 64
3.6.2 Use Case 1: Address Translation 65
3.6.3 Use Case 2: Memory Heterogeneity 68
3.7 Related Work 70
3.8 Summary and Contributions 72

4 Conclusions and Future Work **73**
4.1 Conclusions 73
4.2 Future Work 74
4.2.1 Data-Aware Memory Architectures 74
4.2.2 Enabling Support for Designing New Unconventional Memory Subsystems 75
4.2.3 Virtual Memory Support for Processing-Using-Memory architectures 75

5 Other Works of the Author **76**

Bibliography **78**

Appendix A AIG-to-MIG Conversion **106**

Appendix B Row-to-Operand Allocation **109**

Appendix C Scalability of Operations **112**

Appendix D Evaluated Real-World Applications **113**

List of Tables

Table 2.1 SIMDRAM ISA extensions. 29
Table 2.2 Evaluated system configurations. 33
Table 2.3 Process variation's effect on TRA/QRA failure rates. 38

Table 3.1 Simulation configuration. 64
Table 3.2 Multiprogrammed workload bundles. 68

List of Figures

Figure 2.1 High-level overview of DRAM organization. 12
Figure 2.2 SIMDRAM subarray organization [338]. 15
Figure 2.3 Overview of the SIMDRAM framework. 16
Figure 2.4 Data layout: horizontal vs. vertical. 17
Figure 2.5 (a) Optimized MIG; (b) row-to-operand allocation; (c) µProgram for full addition. 19
Figure 2.6 µOps and µRegisters in SIMDRAM. 20
Figure 2.7 SIMDRAM control unit. 24
Figure 2.8 Major components of the data transposition unit. 26
Figure 2.9 Normalized throughput of 16 operations. SIMDRAM:X uses X DRAM banks for computation. 34
Figure 2.10 Normalized energy efficiency of 16 operations. 35
Figure 2.11 Normalized speedup of real-world kernels. 36
Figure 2.12 Latency and energy to execute 64M operations. 37
Figure 2.13 Latency overhead distribution of worst-case intra-bank (left) and inter-bank (right) data movement for *SIMDRAM:1*. Error bars depict the 25th and 75th percentiles. 39
Figure 2.14 Worst-case latency (left) and worst-case latency overhead distribution (right) of data transposition in 16 SIMDRAM operations for *SIMDRAM:1*. Error bars depict the 25th and 75th percentiles, and a bubble depicts the 50th percentile. 41

Figure 3.1 Virtual memory management in x86-64 and in VBI. 47
Figure 3.2 Overview of VBI. *Lat-Sen* and *Band-Sen* represent latency-sensitive and bandwidth-sensitive, respectively. 49
Figure 3.3 Components of a VBI address. 54
Figure 3.4 Reference microarchitectural implementation of the Virtual Block Interface. 56
Figure 3.5 Partitioning the VBI address space among virtual machines, using the 4 GB size class (100) as an example. 63
Figure 3.6 Performance of systems with 4KB pages (normalized to Native). . . 64
Figure 3.7 Performance with large pages (norm. to Native-2M). 67

Figure 3.8 Multiprogrammed workload performance (normalized to Native). . 68
Figure 3.9 Performance of VBI PCM-DRAM (normalized to data-hotness-unaware mapping). 69
Figure 3.10 Performance of VBI TL-DRAM (normalized to data-hotness-unaware mapping). 69

Chapter 1

Introduction

Modern computing systems need to process increasingly large amounts of data. Many key applications and workloads of important and wide range of domains (e.g., data mining, machine learning, graph and text analytics, databases, augmented reality applications, and genome analysis) and their potential improvement depend on fast and efficient processing of large data volumes. With the advent of such applications, computing in modern systems is primarily bottlenecked by data. In other words, by how fast and efficient we are in accessing, moving and processing data. Unfortunately, modern computing systems do not handle (i.e., store, access, and process) data well. The large amount of input and intermediate data required by modern applications overwhelms many of the key components of the modern computing systems. As a result, data has become a prime bottleneck in today's computing systems, making it challenging to efficiently support important emerging applications with high data demand.

The importance of handling the large amount of data processed by modern applications in an efficient manner has inspired a large body of research in processor design, memory and storage architectures, and key system components. However, we argue that fundamentally efficient handling of the increasing data demand in modern applications requires a holistic rethinking of the key concepts and components used in modern computing systems.

1.1 Motivation: Existing Computing Systems Are Designed Without Having Modern Applications' Data Demand in Mind

Today's computing systems have two important characteristics that make it significantly challenging to efficiently handle large amounts of data:

Characteristic 1: Processor-Centric Architectures. Modern computing systems follow the *processor-centric paradigm* in which computation is performed only in the processor (or compute-centric accelerators) and every piece of data needs to be transferred to/from main memory to enable the computation. The increasing prevalence and growing size of data in modern applications has made data movement between memory devices (e.g., DRAM)

and the processor across bandwidth-limited memory channels a first-class performance and energy bottleneck. For example, a recent work [51] shows that the energy and performance costs of data movement across the memory hierarchy are significantly higher than that of computation, consuming more than 60% of the total system energy, when executing four major commonly-used consumer workloads, including machine learning inference, video processing and playback, and web browsing. Furthermore, in a processor-centric configuration, every component in the system except the processor is designed to serve the processor by storing and accessing the data or moving it to the processor for computation. This leads to about 80-95% of the chip area to be consumed by the components that are solely responsible for storing, accessing and moving the data to the processor [270]. Spending the majority of the chip resources on elements that are not able to process data or understand and take advantage of the properties of the data is not the right mindset considering the advent of applications that require fast, efficient, and intelligent computation of significantly large volumes of data.

We conclude that the processor-centric paradigm as the predominant execution model used in almost all computing systems is designed without having modern applications' overwhelming data demand in mind, causing significant waste in terms of energy and performance by requiring frequent data movement across the entire system. This causes data to become a first-class concern and a prime performance and energy bottleneck in the system, which makes it challenging to efficiently support important emerging applications with high data demand in today's computing systems.

Characteristic 2: Data-Oblivious Policies. In order to cater to the high and diverse memory requirements of modern applications, today's computing systems employ increasingly larger main memories [56,165,211,212,261,266–268,273,277,318,346] and heterogeneous main memory architectures (e.g., [56,61,62,64,195,214,216,218,219,232,258,311,315,318,358,392, 393]). Efficiently exploiting the significantly larger main memory capacities and the increasing heterogeneity in the main memory architectures requires careful memory management that is conventionally performed using virtual memory. However, conventional virtual memory frameworks are designed without considering modern applications' overwhelmingly high memory demand, and thus, without considering the new larger more complex main memory designs. Therefore, continuing to adopt the conventional approach to virtual memory with the increasing capacity and heterogeneity in today's main memory architectures requires a lot of effort and often leads to important challenges and inefficiencies. Furthermore, in addition to the growth in the size of data that modern applications process, prior works [201, 225, 226, 249, 375, 377] show that different pieces of program data have different performance characteristics (latency/bandwidth/parallelism sensitivity), and other inherent properties (e.g., compressibility, persistence, approximability). As highlighted by recent works [201, 249, 375, 377], conveying semantic information about application's data to the hardware that manages the physical memory resources can enable vastly more intelligent *data-*

aware management of the underlying hardware resources (e.g., better address translation, data mapping, migration, and scheduling decisions) and a host of new optimization opportunities.

Unfortunately, conventional virtual memory frameworks [37,38,74,78,79,98,182,183,366] as the key interface between the software stack and the hardware are not capable of conveying any insights regarding the properties and memory behaviour of different pieces of program data. Instead, programs are traditionally conveyed to the hardware in the form of ISA instructions and a set of memory accesses to virtual addresses. This semantic gap leads to hardware treating *all data* as the same, thereby being unable to exploit data's semantics properties to employ more intelligent management or optimization policies. Accordingly, the management and optimization policies used in existing systems are *data-oblivious* and mainly component-aware [271], i.e., designed according to the characteristics of the system component as apposed to the properties of the data that it handles (e.g., tuning tile size to fit a specific cache size). By ignoring the valuable memory characteristics and semantic properties of application's data, each component of the system is required to *predict* the application's data behaviour in order to optimize its policies. Such a strategy is quite challenging and often not very effective due to three main problems. First, each component in the system has a *limited* and *localized* view of the data and is not aware of the overall behaviour of the application. Therefore, its decisions may not be ideal when considering the big picture. Second, each component requires *separate* resources for inferring and predicting the behaviour of the data. This leads to repeated overhead in every component that can be avoided using a unified and expressive interface that connects different layers of the computing stack. Third, the optimizations made by different components mainly *react* to the behaviour of the data as the overall application's behaviour is not available or predictable. This makes it challenging to make timely optimization/management decisions.

Data-oblivious policies in modern systems are a direct result of how poor today's systems are at exploiting the valuable properties of different pieces of application's data, which results in ineffective policies and lost performance optimization opportunities that can be achieved by exploiting data properties to improve the computing systems policies. We posit that conventional virtual memory frameworks, as a critical component of the existing computing systems cannot efficiently support the high data demand and diversity in modern applications, as well as the diversity in today's system configurations that have evolved in response to the modern application's memory needs.

1.2 Our Approach: Data-Centric and Data-Aware Architectures for Fundamentally Efficient Data Handling

In this thesis, we argue that, moving forward, computing systems need to consider large amounts of data and the efficient computation of data as the ultimate priority of the system. In particular, modern computing systems should follow two main directions. (1) data-centric architectures, and (2) data-aware architectures.

Data-Centric Architectures. In contrast to the dominant processor-centric design paradigm, we believe that, in order to efficiently handle large amounts of data, modern computing systems need to be *data-centric*, meaning that they should (1) minimize data movement, and (2) compute data in or near where the data resides. The data-centric approach to computing is highly effective as (1) it improves performance by reducing/eliminating the need to move data to the processor for computation, and (2) provides the ability to take advantage of the large internal bandwidth in the main memory to increase the efficiency of the computation. For example, we show that a processing-using-DRAM architecture that efficiently implements and computes complex operations in DRAM, and provides the ability to support new arbitrary operations can significantly improve the overall performance and efficiency of the system (Chapter 2).

Data-Aware Architectures. In contrast to the dominant data-oblivious policies in existing systems, we believe that modern computing systems should enable *data-aware* policies, by allowing the software to easily communicate properties and semantic information about each application's and system's data to the hardware. A data-aware architecture (1) understands what it can do with and to each piece of data, and (2) makes use of different properties of data (e.g., compressibility, approximability, locality, sparsity, access semantics) to improve performance, efficiency and other metrics. For example, we show that a more scalable data-aware virtual memory framework that (1) is fundamentally designed to handle large amounts of data more efficiently, and (2) understands, conveys and exploits the properties of program's data to enable more intelligent memory management and optimizations, can significantly improve performance for both native execution and virtual machine environments, and significantly improve the effectiveness of heterogeneous main memory architectures (Chapter 3).

1.2.1 Thesis Statement

This thesis, hence, provides evidence for the following thesis statement:

The performance and energy efficiency of computing systems can improve significantly when handling the increasingly large amounts of data in modern applications by employing data-centric and data-aware architectures that can (1) remove the overheads associated with data movement by processing data where it resides, (2) efficiently adapting to the diversity in today's system configurations and memory architectures that are designed to process large amounts of data, and (3) understand, convey, and exploit the characteristics of the data to make more intelligent memory management decisions.

1.3 Overview of Research

In this thesis, we propose a novel data-centric processing-using-memory framework and a novel data-aware virtual memory framework, which we briefly describe next. We also put

these contributions in the context of relevant prior work in Sections 1.3.1 and 1.3.2. We provide detailed discussions of and comparisons to prior work in Chapters 2 and 3.

1.3.1 SIMDRAM: A Data-Centric Framework for Bit-Serial SIMD Processing using DRAM (Chapter 2)

In order to provide processing capability in or near where data resides, many prior works have explored DRAM designs (as well as other memory technologies) that are capable of performing computation using memory [4, 11, 19–22, 66, 71, 77, 82, 85, 94, 103, 108, 141, 157, 172, 188, 197, 227–229, 286, 292, 333, 335, 336, 338, 340–343, 345, 347, 359, 360, 364, 380, 389]. However, these works suffer from three major shortcoming. First, they support only basic operations and fall short on efficiently supporting more complex operations, which limits their applicability [11, 18–22, 77, 108, 141, 157, 229, 337, 338, 389]. Second, they support only a limited and specific set of operations, lacking the flexibility to support new operations and cater to the wide variety of applications that can potentially benefit from processing-using-DRAM [77, 228]. Third, they often require significant changes to the DRAM subarray, which makes them costly [77, 228]. These shortcomings highlight the need for a framework that aids the general adoption of processing-using-DRAM by efficiently implementing complex operations and providing the flexibility to support new desired operations, while requiring minimal changes to the DRAM architecture.

To this end, this thesis introduces SIMDRAM, a flexible general-purpose processing-using-DRAM framework that (1) enables the efficient implementation of complex operations, (2) provides a flexible mechanism to support the implementation of arbitrary user-defined operations, and (3) uses an in-DRAM massively-parallel SIMD substrate that requires minimal changes to the DRAM architecture. We build the in-DRAM substrate used in the SIMDRAM framework around two key techniques. The first key technique is *vertical data layout* in DRAM. Prior works show that employing a vertical layout for the data in DRAM [11, 35, 85, 103, 108, 144, 145, 170, 350, 373] eliminates the need for adding extra logic in DRAM to implement the bit-shift operation [77, 228] which is essential for many complex operations. Employing vertical data layout provides SIMDRAM with two key benefits: (1) implicit shift operation, and (2) massive parallelism, wherein each DRAM column operates as a SIMD lane by placing the source and destination operands of an operation on top of each other in the same DRAM column. The second key technique used in SIMDRAM substrate is *majority-based computation*. As opposed to using basic logic operations such as AND/OR/NOT as building blocks to implement in-DRAM computation [108, 228, 335, 338], SIMDRAM uses logically complete set of majority (MAJ) and NOT operations to implement in-DRAM computation. Majority-based computation enables SIMDRAM to achieve higher performance, higher throughput, and lower energy consumption compared to using basic logical operations as building blocks for in-DRAM computation.

The SIMDRAM framework we introduce is composed of three main steps. The first step of the framework builds an efficient MAJ/NOT representation of a desired operation from

its AND/OR/NOT-based implementation. The second step allocates DRAM rows to the operation's inputs and outputs and generates the required sequence of DRAM commands to execute the desired operation, which is called µProgram. The third step executes the µProgram to perform the operation. SIMDRAM uses a control unit in the memory controller that transparently issues the sequence of commands to DRAM, as dictated by the µProgram.

We provide a detailed reference implementation of SIMDRAM in this thesis, including required hardware, programming, and ISA support, to (1) address key system integration challenges, and (2) allow programmers to define new operations without hardware changes. We demonstrate the generality of the SIMDRAM framework using 16 complex in-DRAM operations, and seven commonly-used real-world applications. We show that SIMDRAM is a promising processing-using-memory framework that (1) can ease the adoption of processing-using-DRAM architectures, and (2) improve the performance and efficiency of processing-using-DRAM architectures.

The SIMDRAM framework is introduced, discussed and evaluated in detail in Chapter 2 of this thesis (as well as Appendixes A, B, C, and D). An earlier version of SIMDRAM was presented at the ASPLOS 2021 conference [131].

1.3.2 The Virtual Block Interface: A Flexible Data-Aware Alternative to the Conventional Virtual Memory Framework (Chapter 3)

Considering the key role that virtual memory has in the overall performance of the modern computing systems, a wide body of research (e.g., [1–3, 10, 17, 27, 28, 30, 32–34, 38, 42–47, 49, 59, 60, 69, 72, 78, 79, 84, 105–107, 111, 126, 128, 129, 134, 146, 149, 171, 174–176, 182, 200, 204–207, 230, 231, 234, 250–254, 257, 259, 297, 304–306, 308, 309, 312, 313, 321–323, 329, 339, 344, 349, 351, 356, 367–369, 378, 383–386, 391, 401]) propose mechanisms to alleviate the various overheads associated with it (in some cases when handling the large data demand in modern applications). However, despite notable improvements, these solutions suffer from three major shortcomings. First, they are mainly designed based on specific system or workload characteristics and, thus, are applicable to only a limited set of problems or applications. Second, each solution requires specialized and not necessarily compatible changes to either the operating system or hardware or both. Therefore, implementing a combination of these proposals at the same time in a system is a daunting prospect. Third, these proposals do *not* support understanding, conveying, and exploiting the properties of program data in order to enable more intelligent memory management decisions (i.e., data-aware architectures). These shortcomings highlight the need for a holistic solution to efficiently support modern applications in today's diverse system configurations, by (1) eliminating the inefficiencies of the conventional virtual memory framework when handling modern applications' large amount of data, (2) exploiting the properties of different pieces of program data to improve performance, efficiency and other metrics.

To this end, in this thesis, we introduce the Virtual Block Interface (VBI), a general-purpose alternative virtual memory framework that has three major properties. First, VBI

is able to understand, convey, and exploit the properties of different pieces of program data to enable more intelligent management of main memory. Second, VBI efficiently and flexibly supports increasingly diverse system configurations that are employed today to process the high data demand in modern applications. Third, VBI provides the key features of the conventional virtual memory framework while eliminating its key inefficiencies when handling large amounts of data in modern applications. The key idea in VBI is to delegate the physical memory allocation and address translation to dedicated hardware in the memory controller.

The VBI design is driven by three key guiding principles. First, programs should be allowed to choose the size of their virtual address space. This mitigates the translation overheads associated with unnecessarily large and fixed-sized virtual address spaces in current systems that results in increasingly large inefficiencies with the diverse memory requirements of modern applications. Second, address translation should be decoupled from memory protection, as they are logically separate. This enables opportunities to remove address translation from the critical path of an access protection check, and defer the address translation until physical memory must be accessed, thereby lowering the performance overheads of virtual memory when handling large amounts of data in modern applications. It also enables the flexibility of managing address translation and access protection using separate structures customized to their characteristics, which again helps with more efficient address translation mechanisms which reduce the overhead of processing large data volumes in modern applications. Third, software should be allowed to communicate semantic information about application data to the hardware. This helps hardware to exploit the rich properties of different pieces of data to manage the underlying memory resources more intelligently.

VBI naturally enables a variety of important optimizations that improve overall system performance when handling the high memory demand in modern applications, including: (1) enabling benefits akin to using virtually-indexed virtually-tagged (VIVT) caches (e.g., reduced address translation overhead), (2) eliminating two-dimensional page table walks in virtual machine environments, (3) delaying physical memory allocation until the first dirty last-level cache line eviction, and (4) flexibly supporting different virtual-to-physical address translation structures for different memory regions.

We demonstrate the benefits of VBI with two example use cases. First, we experimentally show that VBI significantly improves performance for both native execution and virtual machine environments. Second, we show that VBI significantly improves the effectiveness of heterogeneous main memory architectures. We demonstrate that VBI is a promising new virtual memory framework, that can enable several important optimizations, and increase the design flexibility for virtual memory to support efficient handling of data in modern computing systems.

The VBI virtual memory framework is introduced, discussed, and evaluated in detail in Chapter 3 of this thesis. An earlier version of VBI was presented at the ISCA 2020 conference [132].

1.4 Contributions

To our knowledge, this thesis is the first to propose and study new frameworks for fundamentally-efficient data handling in modern computing systems with a focus on two key directions at the same time, i.e., data-centric and data-aware architectures. In this thesis, we make two major contributions.

- We present SIMDRAM, an end-to-end processing-using-DRAM framework that aids the widespread adoption of processing-using-DRAM, a data-centric computation paradigm that improves the overall performance and efficiency of the system when computing on large amounts of data by minimizing the cost of data movement and enabling computation where the data resides. To this end, SIMDRAM (1) efficiently computes complex operations required by modern data intensive applications, (2) provides the ability to implement new arbitrary operations as required, and (3) uses an in-DRAM massively-parallel SIMD substrate that requires minimal changes to the DRAM architecture. We provide a detailed reference implementation of SIMDRAM, including required changes to applications, ISA, and hardware. We demonstrate the effectiveness and generality of the SIMDRAM framework at improving the system performance and efficiency using a wide range of complex operations and commonly-used real-world applications.

- We introduce VBI, a novel data-aware scalable general-purpose virtual memory framework that enables efficient handling of large amounts of data in modern applications by (1) efficiently understanding, conveying, and exploiting the properties of different pieces of program data to enable more intelligent management of main memory, (2) efficiently and flexibly supporting increasingly diverse system configurations and memory architectures that are employed today to process the high data demand in modern applications, and (3) providing the key features of the conventional virtual memory framework while eliminating its key inefficiencies when handling large amounts of data in modern applications. We provide a detailed reference implementation of VBI, including required changes to applications, system software, ISA, and hardware. We demonstrate the effectiveness of VBI at significantly improving the overall system performance in native and virtualized environments. We also show that VBI significantly improves the effectiveness of heterogeneous main memory architectures.

  Other contributions of this thesis are listed in Chapters 2 and 3. We specifically direct the reader to the final subsections in each chapter for a concise summary and contributions of each chapter, i.e., Sections 2.8 and 3.8 in this thesis.

Chapter 2

SIMDRAM

As discussed in Chapter 1, the increasing prevalence and growing size of data in modern applications has led to high energy and latency costs for computation in traditional computer architectures. Moving large amounts of data between memory (e.g., DRAM) and the CPU across bandwidth-limited memory channels can consume more than 60% of the total energy in modern systems [51, 272]. To mitigate such costs, the *processing-in-memory* (PIM) paradigm moves computation closer to where the data resides, reducing (and in some cases eliminating) the need to move data between memory and the processor.

There are two main approaches to PIM [112, 273]: (1) processing-near-memory, where PIM logic is added to the same die as memory or to the logic layer of 3D-stacked memory [7–9, 14, 15, 26, 29, 31, 39, 40, 50–55, 73, 80, 83, 92, 109, 110, 113, 116, 118, 124, 127, 136, 137, 147, 148, 150–152, 166, 184, 185, 189, 190, 215, 221, 239–241, 243, 278, 279, 287–289, 302, 314, 327, 331, 353–355, 365, 397, 399, 402, 403]; and (2) processing-using-memory, which makes use of the operational principles of the memory cells themselves to perform computation by enabling interactions between cells [4, 11, 19–22, 39, 40, 66, 71, 77, 82, 85, 94, 103, 108, 141, 157, 172, 188, 197, 227–229, 286, 333, 335, 336, 338, 340–343, 345, 347, 359, 360, 364, 380, 389]. Since processing-using-memory operates directly in the memory arrays, it benefits from the large internal bandwidth and parallelism available inside the memory arrays, which processing-near-memory solutions cannot take advantage of.

A common approach for processing-using-memory architectures is to make use of bulk bitwise computation. Many widely-used data-intensive applications (e.g., databases, neural networks, graph analytics) heavily rely on a broad set of simple (e.g., AND, OR, XOR) and complex (e.g., equality check, multiplication, addition) bitwise operations. Ambit [335, 338], an in-DRAM processing-using-memory accelerator, was the first work to propose exploiting DRAM's analog operational principles to perform bulk bitwise AND, OR, and NOT logic operations. Inspired by Ambit, many prior works have explored DRAM (as well as NVM) designs that are capable of performing in-memory bitwise operations [11, 18–22, 77, 108, 141, 157, 229, 333, 335–338, 340–343, 389]. However, a major shortcoming prevents these proposals from becoming widely applicable: they support only basic operations (e.g., Boolean

operations, addition) and fall short on flexibly and easily supporting new and more complex operations. Some prior works propose processing-using-DRAM designs that support more complex operations [77, 228]. However, such designs (1) require significant changes to the DRAM subarray, and (2) support only a limited and specific set of operations, lacking the flexibility to support new operations and cater to the wide variety of applications that can potentially benefit from in-memory computation.

Our goal in this work is to design a framework that aids the adoption of processing-using-DRAM by efficiently implementing complex operations and providing the flexibility to support new desired operations. To this end, we propose SIMDRAM, an end-to-end processing-using-DRAM framework that provides the programming interface, the ISA, and the hardware support for (1) efficiently computing *complex* operations, and (2) providing the ability to implement *arbitrary* operations as required, all in an in-DRAM massively-parallel SIMD substrate. At its core, we build the SIMDRAM framework around a DRAM substrate that enables two previously-proposed techniques: (1) vertical data layout in DRAM, and (2) majority-based logic for computation.

Vertical Data Layout. Supporting bit-shift operations is essential for implementing complex computations, such as addition or multiplication. Prior works show that employing a vertical layout [11, 35, 85, 103, 108, 144, 145, 170, 350, 373] for the data in DRAM, such that all bits of an operand are placed in a single DRAM column (i.e., in a single bitline), eliminates the need for adding extra logic in DRAM to implement shifting [77, 228]. Accordingly, SIMDRAM supports efficient bit-shift operations by storing operands in a vertical fashion in DRAM. This provides SIMDRAM with two key benefits. First, a bit-shift operation can be performed by simply copying a DRAM row into another row (using RowClone [336], LISA [66], NoM [345] or FIGARO [380]). For example, SIMDRAM can perform a left-shift-by-one operation by copying the data in DRAM row *j* to DRAM row *j+1*. (Note that while SIMDRAM supports bit shifting, we can optimize many applications to avoid the need for explicit shift operations, by simply changing the row indices of the SIMDRAM commands that read the shifted data). Second, SIMDRAM enables massive parallelism, wherein each DRAM column operates as a SIMD lane by placing the source and destination operands of an operation on top of each other in the same DRAM column.

Majority-Based Computation. Prior works use majority operations to implement basic logical operations [108, 228, 335, 338] (e.g., AND, OR) or addition [11, 19, 77, 104, 108, 228]. These basic operations are then used as basic building blocks to implement the target in-DRAM computation. SIMDRAM extends the use of the majority operation by directly using only the logically complete set of majority (MAJ) and NOT operations to implement in-DRAM computation. Doing so enables SIMDRAM to achieve higher performance, throughput, and reduced energy consumption compared to using basic logical operations as building blocks for in-DRAM computation. We find that a computation typically requires fewer

DRAM commands using MAJ and NOT than using basic logical operations AND, OR, and NOT.

To aid the adoption of processing-using-DRAM by flexibly supporting new and more complex operations, SIMDRAM addresses two key challenges: (1) how to synthesize new arbitrary in-DRAM operations, and (2) how to exploit an optimized implementation and control flow for such newly-added operations while taking into account key limitations of in-DRAM processing (e.g., DRAM operations that destroy input data, limited number of DRAM rows that are capable of processing-using-DRAM, and the need to avoid costly in-DRAM copies). As a result, SIMDRAM is the first end-to-end framework for processing-using-DRAM. SIMDRAM provides (1) an effective algorithm to generate an efficient MAJ/NOT-based implementation of a given desired operation; (2) an algorithm to appropriately allocate DRAM rows to the operands of the operation and an algorithm to map the computation to an efficient sequence of DRAM commands to execute *any* MAJ-based computation; and (3) the programming interface, ISA support and hardware components required to (i) compute any new user-defined in-DRAM operation without hardware modifications, and (ii) program the memory controller for issuing DRAM commands to the corresponding DRAM rows and correctly performing the computation. Such end-to-end support enables SIMDRAM as a holistic approach that facilitates the adoption of processing-using-DRAM through (1) enabling the flexibility to support new in-DRAM operations by providing the user with a simplified interface to add desired operations, and (2) eliminating the need for adding extra logic to DRAM.

The SIMDRAM framework efficiently supports a wide range of operations of different types. In this work, we demonstrate the functionality of the SIMDRAM framework using an example set of 16 operations including (1) N-input logic operations (e.g., AND/OR/XOR of more than 2 input bits); (2) relational operations (e.g., equality/inequality check, greater than, maximum, minimum); (3) arithmetic operations (e.g., addition, subtraction, multiplication, division); (4) predication (e.g., if-then-else); and (5) other complex operations such as bitcount and ReLU [120]. The SIMDRAM framework is not limited to these 16 operations, and can enable processing-using-DRAM for other existing and future operations. SIMDRAM is well-suited to application classes that (i) are SIMD-friendly, (ii) have a regular access pattern, and (iii) are memory bound. Such applications are common in domains such as database analytics, high-performance computing, image processing, and machine learning.

We compare the benefits of SIMDRAM to different state-of-the-art computing platforms (CPU, GPU, and the Ambit [338] in-DRAM computing mechanism). We comprehensively evaluate SIMDRAM's reliability, area overhead, throughput, and energy efficiency. We leverage the SIMDRAM framework to accelerate seven application kernels from machine learning, databases, and image processing (VGG-13 [352], VGG-16 [352], LeNET [209], kNN [222], TPC-H [372], BitWeaving [233], brightness [119]). Using a single DRAM bank, SIMDRAM provides (1) 2.0× the throughput and 2.6× the energy efficiency of Ambit [338],

averaged across the 16 implemented operations; and (2) 2.5× the performance of Ambit, averaged across the seven application kernels. Compared to a CPU and a high-end GPU, SIMDRAM using 16 DRAM banks provides (1) 257× and 31× the energy efficiency, and 88× and 5.8× the throughput of the CPU and GPU, respectively, averaged across the 16 operations; and (2) 21× and 2.1× the performance of the CPU and GPU, respectively, averaged across the seven application kernels. SIMDRAM incurs no additional area overhead on top of Ambit, and a total area overhead of only 0.2% in a high-end CPU. We also evaluate the reliability of SIMDRAM under different degrees of manufacturing process variation, and observe that it guarantees correct operation as the DRAM process technology node scales down to smaller sizes.

2.1 Background

We first briefly explain the architecture of a typical DRAM chip. Next, we describe prior processing-using-DRAM works that SIMDRAM builds on top of (RowClone [336] and Ambit [335, 338, 342]) and explain the implications of majority-based computation. For an even more detailed operation of DRAM, we refer the reader to many prior works [61, 63–66, 138–140, 168, 178–181, 186, 188, 191, 193, 195–197, 214–216, 216–219, 237, 238, 299–301, 333, 336, 338, 339, 341, 381].

2.1.1 DRAM Basics

A DRAM system comprises a hierarchy of components, as Figure 2.1 shows, starting with *channels* at the highest level. A channel is subdivided into *ranks*, and a rank is subdivided into multiple *banks* (e.g., 8-16). Each bank is composed of multiple (e.g., 64-128) 2D arrays of cells known as *subarrays*. Cells within a subarray are organized into multiple *rows* (e.g., 512-1024) and multiple *columns* (e.g., 2-8 kB) [186, 216, 217]. A cell consists of an *access transistor* and a *storage capacitor* that encodes a single bit of data using its voltage level. The source nodes of the access transistors of all the cells in the same column connect the cells' storage capacitors to the same *bitline*. Similarly, the gate nodes of the access transistors of all the cells in the same row connect the cells' access transistors to the same *wordline*.

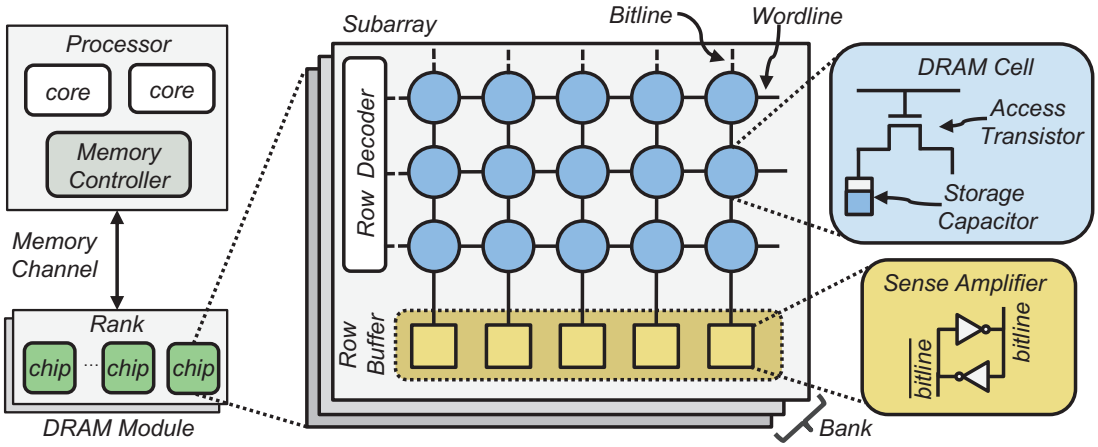


Figure 2.1: High-level overview of DRAM organization.

When a wordline is asserted, all cells along the wordline are connected to their corresponding bitlines, which perturbs the voltage of each bitline depending on the value stored in each cell's capacitor. A two-terminal *sense amplifier* connected to each bitline senses the voltage difference between the bitline (connected to one terminal) and a reference voltage (typically $\frac{1}{2}V_{DD}$; connected to the other terminal) and amplifies it to a CMOS-readable value. In doing so, the sense amplifier terminal connected to the reference voltage is amplified to the *opposite* (i.e., *negated*) value, which is shown as the $\overline{\text{bitline}}$ terminal in Figure 2.1. The set of sense amplifiers in each subarray forms a logical *row buffer*, which maintains the sensed data for as long as the row is *open* (i.e., the wordline continues to be asserted). A read or write operation in DRAM includes *three* steps:

1. `ACTIVATE`. The *wordline* of the target row is asserted, which connects all cells along the row to their respective bitlines. Each bitline shares charge with its corresponding cell capacitor, and the resulting bitline voltage shift is sensed and amplified by the bitline's sense amplifier. Once the sense amplifiers finish amplification, the row buffer contains the values originally stored within the cells along the asserted wordline.
2. `RD/WR`. The memory controller then issues read or write commands to columns within the activated row (i.e., the data within the row buffer).
3. `PRECHARGE`. The capacitor is disconnected from the bitline by disabling the wordline, and the bitline voltage is restored to its quiescent state (e.g., typically $\frac{1}{2}V_{DD}$).

2.1.2 Processing-using-DRAM

In-DRAM Row Copy.

RowClone [336] is a mechanism that exploits the vast internal DRAM bandwidth to efficiently copy rows inside DRAM without CPU intervention. RowClone enables copying a source row A to a destination row B in the same subarray by issuing two consecutive `ACTIVATE` commands to these two rows, followed by a `PRECHARGE` command. This command sequence is called `AAP` [338]. The first `ACTIVATE` command copies the contents of the source row A into the row buffer. The second `ACTIVATE` command connects the cells in the destination row B to the bitlines. Because the sense amplifiers have already sensed and amplified the source data by the time row B is activated, the data (i.e., voltage level) in each cell of row B is overwritten by the data stored in the row buffer (i.e., row A's data). Recent work [108] experimentally demonstrates the feasibility of executing in-DRAM row copy operations in unmodified off-the-shelf DRAM chips.

In-DRAM Bitwise Operations.

Ambit [335, 338, 342] shows that simultaneously activating *three* DRAM rows (via a DRAM operation called *Triple Row Activation, TRA*) can be used to perform bitwise Boolean AND, OR, and NOT operations on the values contained within the cells of the three rows. When activating three rows, three cells connected to each bitline share charge simultaneously and contribute to the perturbation of the bitline. Upon sensing the perturbation, the sense amplifier amplifies the bitline voltage to V_{DD} or 0 if at least two of the capacitors of the

three DRAM cells are charged or discharged, respectively. As such, a TRA results in a Boolean *majority operation* (MAJ) among the three DRAM cells on each bitline. A majority operation MAJ outputs a 1 (0) only if more than half of its inputs are 1 (0). In terms of AND ($\cdot$) and OR ($+$) operations, a 3-input majority operation can be expressed as `MAJ(A, B, C) = A · B + A · C + B · C`.

Ambit implements MAJ by introducing a custom row decoder (discussed in Section 2.2.1) that can perform a TRA by simultaneously addressing three wordlines. To use this decoder, Ambit defines a new command sequence called `AP`, which issues (1) a TRA to compute the MAJ of three rows, followed by (2) a `PRECHARGE` to close all three rows.[1] Ambit uses `AP` command sequences to implement Boolean AND and OR operations by simply setting one of the inputs (e.g., C) to 1 or 0. The AND operation is computed by setting C to 0 (i.e., `MAJ(A, B, 0) = A AND B`). The OR operation is computed by setting C to 1 (i.e., `MAJ(A, B, 1) = A OR B`).

To achieve functional completeness alongside AND and OR operations, Ambit implements NOT operations by exploiting the differential design of DRAM sense amplifiers. As Section 2.1.1 explains, the sense amplifier already generates the complement of the sensed value as part of the activation process ($\overline{\text{bitline}}$ in Figure 2.1). Therefore, Ambit simply forwards $\overline{\text{bitline}}$ to a special DRAM row in the subarray that consists of DRAM cells with *two* access transistors, called *dual-contact cells* (DCCs). Each access transistor is connected to one side of the sense amplifier and is controlled by a separate wordline (*d-wordline* or *n-wordline*). By activating either the d-wordline or the n-wordline, the row of DCCs can provide the true or negated value stored in the row's cells, respectively.

Majority-Based Computation.

Activating multiple rows simultaneously reduces the reliability of the value read by the sense amplifiers due to manufacturing process variation, which introduces non-uniformities in circuit-level electrical characteristics (e.g., variation in cell capacitance levels) [338]. This effect worsens with (1) an increased number of simultaneously activated rows, and (2) more advanced technology nodes with smaller sizes. Accordingly, although processing-using-DRAM can potentially support majority operations with more than three inputs (as proposed by prior works [11, 19, 229]) our realization of processing-using-DRAM uses the minimum number of inputs required for a majority operation (N=3) to maintain the reliability of the computation. In Section 2.6.5, we demonstrate via SPICE simulations that using 3-input MAJ operations provides higher reliability compared to designs with more than three inputs per MAJ operation. Using 3-input MAJ, a processing-using-DRAM substrate does not require modifications to the subarray organization (Figure 2.2) beyond the ones proposed

[1] Although the '`A`' in `AP` refers to a TRA operation instead of a conventional `ACTIVATE` command, we use this terminology to remain consistent with the Ambit paper [338], since an `ACTIVATE` command can be internally translated to a TRA operation by the DRAM chip [338].

by Ambit (Section 2.2.1). Recent work [108] experimentally demonstrates the feasibility of executing MAJ operations by activating three rows in unmodified off-the-shelf DRAM chips.

2.2 SIMDRAM Overview

SIMDRAM is a processing-using-DRAM framework whose goal is to (1) enable the efficient implementation of complex operations and (2) provide a flexible mechanism to support the implementation of arbitrary user-defined operations. We present the subarray organization in SIMDRAM, describe an overview of the SIMDRAM framework, and explain how to integrate SIMDRAM into a system.

2.2.1 Subarray Organization

In order to perform processing-using-DRAM, SIMDRAM makes use of a subarray organization that incorporates additional functionality to perform logic primitives (i.e., MAJ and NOT). This subarray organization is *identical* to Ambit's [338] and is similar to DRISA's [228]. Figure 2.2 illustrates the internal organization of a subarray in SIMDRAM, which resembles a conventional DRAM subarray. SIMDRAM requires only minimal modifications to the DRAM subarray (namely, a small row decoder that can activate three rows simultaneously) to enable computation. Like Ambit [338], SIMDRAM divides DRAM rows into *three groups*: the **D**ata group (D-group), the **C**ontrol group (C-group) and the **B**itwise group (B-group).

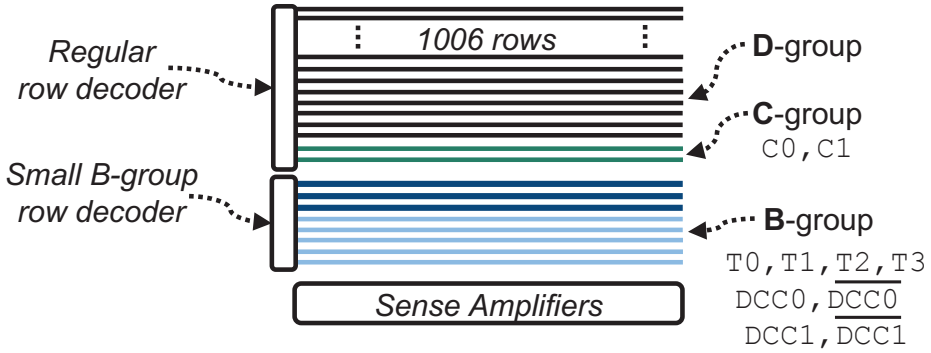
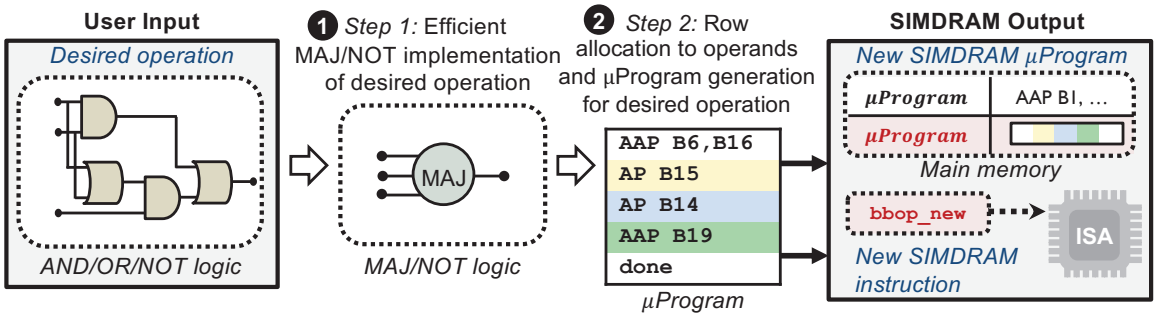


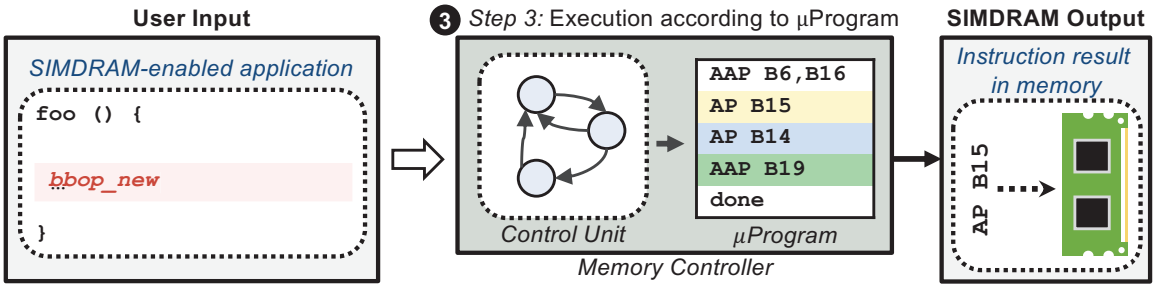
Figure 2.2: SIMDRAM subarray organization [338].

The D-group contains regular rows that store program or system data. The C-group consists of two constant rows, called C0 and C1, that contain all-0 and all-1 values, respectively. These rows are used (1) as initial input values for a given SIMDRAM operation (e.g., the initial carry-in bit in a full addition), or (2) to perform operations that naturally require AND/OR operations (e.g., AND/OR reductions). The D-group and the C-group are connected to the regular row decoder, which selects a single row at a time.

The B-group contains six regular rows, called T0, T1, T2, and T3; and two rows of dual-contact cells (see Section 2.1.2), whose d-wordlines are called DCC0 and DCC1, and whose n-wordlines are called $\overline{\text{DCC0}}$ and $\overline{\text{DCC1}}$, respectively. The B-group rows, called *compute rows*, are designated to perform bitwise operations. They are all connected to a special row decoder that can simultaneously activate three rows using a single address (i.e., perform a TRA)



(a) SIMDRAM Framework: Steps 1 and 2



(b) SIMDRAM Framework: Step 3

Figure 2.3: Overview of the SIMDRAM framework.

Using a typical subarray size of 1024 rows [65, 186, 188, 195, 218], SIMDRAM splits the row addressing into 1006 D-group rows, 2 C-group rows, and 16 B-group rows.

2.2.2 Framework Overview

SIMDRAM is an end-to-end framework that provides the user with the ability to implement an *arbitrary* operation in DRAM using the `AAP`/`AP` command sequences. The framework comprises three key steps, which are illustrated in Figure 2.3. The first two steps of the framework give the user the ability to efficiently implement any desired operation in DRAM, while the third step controls the execution flow of the in-DRAM computation transparently from the user. We briefly describe these steps below, and discuss each step in detail in Section 2.3.

The first step (❶ in Figure 2.3a; Section 2.3.1) builds an efficient MAJ/NOT representation of a given desired operation from its AND/OR/NOT-based implementation. Specifically, this step takes as input a desired operation and uses logic optimization to minimize the number of logic primitives (and, therefore, the computation latency) required to perform the operation. Accordingly, for a desired operation input into the SIMDRAM framework by the user, the first step derives its *optimized* MAJ/NOT-based implementation.

The second step (❷ in Figure 2.3a; Section 2.3.2) allocates DRAM rows to the operation's inputs and outputs and generates the required sequence of DRAM commands to execute the desired operation. Specifically, this step translates the MAJ/NOT-based implementation of the operation into `AAP`s/`AP`s. This step involves (1) allocating the designated compute rows in DRAM to the operands, and (2) determining the optimized sequence of `AAP`s/`AP`s that

are required to perform the operation. While doing so, SIMDRAM minimizes the number of `AAP`s/`AP`s required for a specific operation. This step's output is a µProgram, i.e., the optimized sequence of `AAP`s/`AP`s that is stored in main memory and will be used to execute the operation at runtime.

The third step (❸ in Figure 2.3b; Section 2.3.3) executes the µProgram to perform the operation. Specifically, when a user program encounters a *bbop* instruction (Section 2.4.2) associated with a SIMDRAM operation, the *bbop* instruction triggers the execution of the SIMDRAM operation by performing its µProgram in the memory controller. SIMDRAM uses a *control unit* in the memory controller that transparently issues the sequence of `AAP`s/`AP`s to DRAM, as dictated by the µProgram. Once the µProgram is complete, the result of the operation is held in DRAM.

2.2.3 Integrating SIMDRAM in a System

As we discuss earlier, SIMDRAM operates on data using a vertical layout. Figure 2.4 illustrates how data is organized within a DRAM subarray when employing a horizontal data layout (Figure 2.4a) and a vertical data layout (Figure 2.4b). We assume that each data element is four bits wide, and that there are four data elements (each one represented by a different color). In a conventional horizontal data layout, data elements are stored in different DRAM rows, with the contents of each data element ordered from the most significant bit to the least significant bit (or vice versa) in a single row. In contrast, in a vertical data layout, the DRAM row holds only the i-th bit of *multiple* data elements (where the number of elements is determined by the bit width of the row). Therefore, when activating a single DRAM row in a vertical data layout organization, a *single* bit of data from each data element is read at once, which enables in-DRAM bit-serial parallel computation [11, 35, 124, 228, 342, 350].

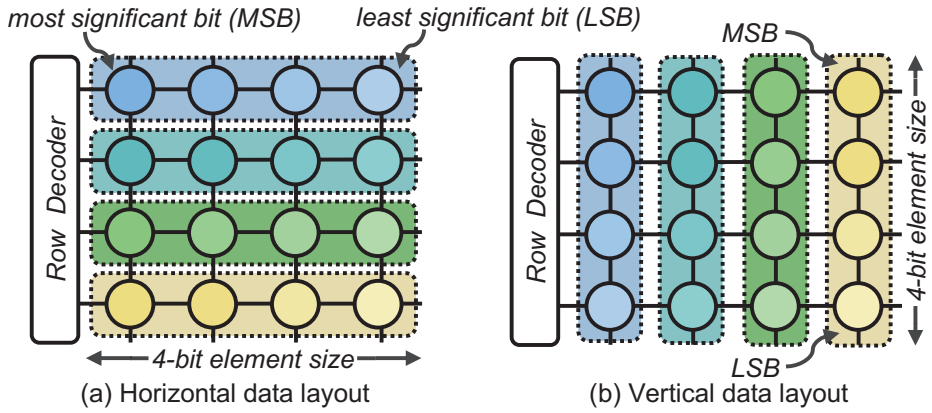


Figure 2.4: Data layout: horizontal vs. vertical.

To maintain compatibility with traditional system software, we store regular data in the conventional horizontal layout and provide hardware support (explained in Section 2.4.1) to transpose horizontally-laid-out data into the vertical layout for in-DRAM computation. To

simplify program integration, we provide ISA extensions that expose SIMDRAM operations to the programmer (Section 2.4.2).

2.3 SIMDRAM Framework

We describe the three steps of the SIMDRAM framework introduced in Section 2.2.2, using the full addition operation as a running example.

2.3.1 Step 1: Efficient MAJ/NOT Implementation

SIMDRAM implements in-DRAM computation using the logically-complete set of MAJ and NOT logic primitives, which requires fewer `AAP`/`AP` command sequences to perform a given operation when compared to using AND/OR/NOT. As a result, the goal of the first step in the SIMDRAM framework is to build an optimized MAJ/NOT implementation of a given operation that executes the operation using as few `AAP`/`AP` command sequences as possible, thus minimizing the operation's latency. To this end, Step 1 transforms an AND/OR/NOT representation of a given operation to an optimized MAJ/NOT representation using a transformation process formalized by prior work [16].

The transformation process uses a graph-based representation of the logic primitives, called an *AND–OR–Inverter Graph* (AOIG) for AND/OR/NOT logic, and a *Majority–Inverter Graph* (MIG) for MAJ/NOT logic. An AOIG is a logic representation structure in the form of a directed acyclic graph where each node represents an AND or OR logic primitive. Each edge in an AOIG represents an input/output dependency between nodes. The incoming edges to a node represent input operands of the node and the outgoing edge of a node represents the output of the node. The edges in an AOIG can be either regular or complemented (which represents an inverted input operand; denoted by a bubble on the edge). The direction of the edges follows the natural direction of computation from inputs to outputs. Similarly, a MIG is a directed acyclic graph in which each node represents a three-input MAJ logic primitive, and each regular/complemented edge represents one input or output to the MAJ primitive that the node represents. The transformation process consists of two parts that operate on an input AOIG.

The first part of the transformation process naively substitutes AND/OR primitives with MAJ primitives. Each two-input AND or OR primitive is simply replaced with a three-input MAJ primitive, where one of the inputs is tied to logic 0 or logic 1, respectively. This naive substitution yields a MIG that *correctly* replicates the functionality of the input AOIG, but the MIG is likely *inefficient*.

The second part of the transformation process takes the inefficient MIG and uses a greedy algorithm (see Appendix A) to apply a series of transformations that identifies how to consolidate multiple MAJ primitives into a smaller number of MAJ primitives with identical functionality. This yields a smaller MIG, which in turn requires fewer logic primitives to perform the same operation that the unoptimized MIG (and, thus, the input AOIG) performs.

Figure 2.5a shows the optimized MIG produced by the transformation process for a full addition operation.

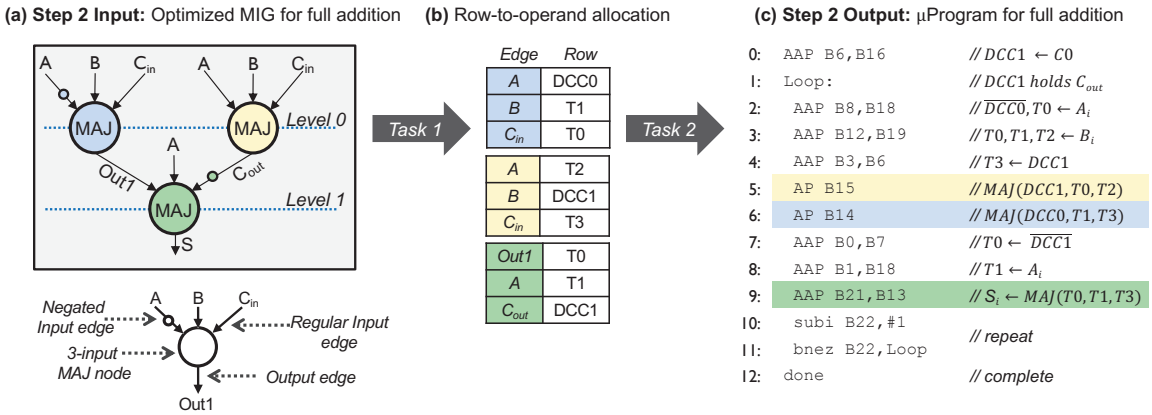


Figure 2.5: (a) Optimized MIG; (b) row-to-operand allocation; (c) µProgram for full addition.

2.3.2 Step 2: µProgram Generation

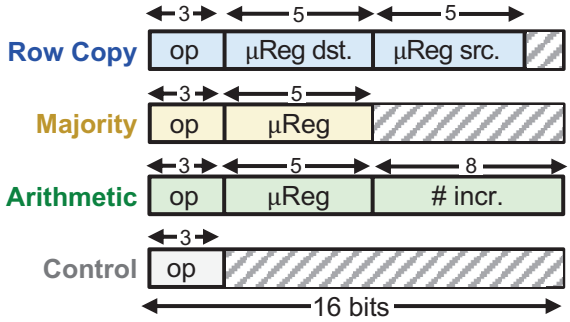
Each SIMDRAM operation is stored as a *µProgram*, which consists of a series of microarchitectural operations (µOps) that SIMDRAM uses to execute the SIMDRAM operation in DRAM. The goal of the second step is to take the optimized MIG generated in Step 1 and generate a µProgram that executes the SIMDRAM operation that the MIG represents. To this end, as shown in Figure 2.5, the second step of the framework performs two key tasks on the optimized MIG: (1) *allocating DRAM rows to the operands*, which assigns each input operand (i.e., an incoming edge) of each MAJ node in the MIG to a DRAM row (Figure 2.5b); and (2) *generating the µProgram*, which creates the series of µOps that perform the MAJ and NOT logic primitives (i.e., nodes) in the MIG, while maintaining the correct flow of the computation (Figure 2.5c). In this section, we first describe the µOps used in SIMDRAM (Section 2.3.2). Second, we explain the process of allocating DRAM rows to the input operands of the MAJ nodes in the MIG to DRAM rows (Section 2.3.2). Third, we explain the process of generating the µProgram (Section 2.3.2).

SIMDRAM µOps.

Figure 2.6a shows the set of µOps that we implement in SIMDRAM. Each µOp is either (1) a *command sequence* that is issued by SIMDRAM to a subarray to perform a portion of the in-DRAM computation, or (2) a *control* operation that is used by the SIMDRAM control unit (see Section 2.3.3) to manage the execution of the SIMDRAM operation. We further break down the command sequence µOps into one of three types: (1) *row copy*, a µOp that performs in-DRAM copy from a source memory address to a destination memory address using an `AAP` command sequence; (2) *majority*, a µOp that performs a majority logic primitive on three DRAM rows using an `AP` command sequence (i.e., it performs a TRA); and (3) *arithmetic*, four µOps that perform simple arithmetic operations on SIMDRAM control unit registers required to control the execution of the operation (`addi`, `subi`, `comp`,

`module`). We provide two control operation µOps to support loops and termination in the SIMDRAM control flow (`bnez`, `done`).

(a) µOps and their description



Opcode	Mnemonic	Type
000	AAP	Row Copy
001	AP	Majority
010	addi	Arithmetic
011	subi	
100	comp	
101	module	
110	bnez	Control
111	done	

(b) µRegisters

µRegister	Contents	µRegister	Contents	µRegister	Contents
B0	$T0\ addr$	B8	$\overline{DCC0}, T0\ addr$	B16	$C0\ addr$
B1	$T1\ addr$	B9	$\overline{DCC1}, T1\ addr$	B17	$C1\ addr$
B2	$T2\ addr$	B10	$T2, T3\ addr$	B18	$input\ addr$
B3	$T3\ addr$	B11	$T0, T3\ addr$	B19	$input\ addr$
B4	$DCC0\ addr$	B12	$T0, T1, T2\ addr$	B20	$input\ addr$
B5	$\overline{DCC0}\ addr$	B13	$T0, T1, T3\ addr$	B21	$output\ addr$
B6	$DCC1\ addr$	B14	$DCC0, T1, T3\ addr$	B22	$element\ size$
B7	$\overline{DCC1}\ addr$	B15	$DCC1, T0, T2\ addr$	B23~B31	$GP\ regs.$

Figure 2.6: µOps and µRegisters in SIMDRAM.

During µProgram generation, the SIMDRAM framework converts the MIG into a series of µOps. Note that MIG represents a 1-bit-wide computation of an operation. Thus, to implement a multi-bit-wide SIMDRAM operation, the framework needs to repeat the series of the µOps that implement the MIG n times, where n is the number of bits in the operands of the SIMDRAM operation. To this end, SIMDRAM uses the arithmetic and control µOps to repeat the 1-bit-wide computation n times, transparently to the programmer.

To support the execution of µOps, SIMDRAM utilizes a set of *µRegisters* (Figure 2.6b) located in the SIMDRAM control unit (Section 2.3.3). The framework uses µRegisters (1) to store the memory addresses of DRAM rows in the B-group and C-group (Figure 2.2.1) of the subarray (µRegisters B0–B17), (2) to store the memory addresses of input and output rows for the computation (µRegisters B18–B22), and (3) as general-purpose registers during the execution of arithmetic and control operations (µRegisters B23–B31).

Task 1: Allocating DRAM Rows to the Operands.

The goal of this task is to allocate DRAM rows to the input operands (i.e., incoming edges) of each MAJ node in the operation's MIG, such that we minimize the total number of µOps needed to compute the operation. To this end, we present a new allocation algorithm inspired by the linear scan register allocation algorithm [310]. However, unlike register allocation algorithms, our allocation algorithm considers two extra constraints that are specific to processing-using-DRAM: (1) performing MAJ in DRAM has destructive behavior, i.e., a TRA overwrites the original values of the three input rows with the MAJ output; and

(2) the number of compute rows (i.e., B-group in Figure 2.2) that are designated to perform bitwise operations is limited (there are only six compute rows in each subarray, as discussed in Section 2.2.1).

The SIMDRAM *row-to-operand allocation algorithm* receives the operation's MIG as input. The algorithm assumes that the input operands of the operation are already stored in separate rows of the D-group in the subarray using vertical layout (Section 2.2.3), before the computation of the operation starts. The algorithm then does a topological traversal starting with the leftmost MAJ node at the highest level of the MIG (e.g., level 0 in Figure 2.5a), allocating compute rows to the input operands of each MAJ node in the current level of the MIG, before moving to the next lower level of the graph. The algorithm finishes once DRAM rows are allocated to all the input operands of all the MAJ nodes in the MIG. Figure 2.5b shows these allocations as the output of Task 1 for the full addition example. The resulting row-to-operand allocation is then used in the second task in step two (Section 2.3.2) to generate the series of µOps to compute the operation that the MIG represents. We describe our row-to-operand allocation algorithm in Appendix B.

Task 2: Generating a µProgram.

The goal of this task is to use the MIG and the DRAM row allocations from Task 1 to generate the µOps of the µProgram for our SIMDRAM operation. To this end, Task 2 (1) translates the MIG into a series of row copy and majority µOps (i.e., `AAP`s/`AP`s), (2) optimizes the series of µOps to reduce the number of `AAP`s/`AP`s, and (3) generalizes the one-bit bit-serial operation described by the MIG into an n-bit operation by utilizing SIMDRAM's arithmetic and control µOps.

Translating the MIG into a Series of Row Copy and Majority µOps. The allocation produced during Task 1 dictates how DRAM rows are allocated to each edge in the MIG during the µProgram. With this information, the framework can generate the appropriate series of row copies and majority µOps to reflect the MIG's computation in DRAM. To do so, we traverse the input MIG in topological order. For *each* node, we *first* assign row copy µOps (using the `AAP` command sequence) to the node's edges. Then, we assign a majority µOp (using the `AP` command sequence) to execute the current MAJ node, following the DRAM row allocation assigned to each edge of the node. The framework repeats this procedure for all the nodes in the MIG. To illustrate, we assume that the SIMDRAM allocation algorithm allocates DRAM rows DCC0, T1, and T0 to edges A, B, and C_{in}, respectively, of the blue node in the full addition MIG (Figure 2.5a). Then, when visiting this node, we generate the following series of µOps:

```
AAP DCC0, A;        // DCC0 ← A
AAP T1, B;          // T1 ← B
AAP T0, C_in;       // T0 ← C_in
AP ¬DCC0, T1, T0    // MAJ(NOT(A), B, C_in)
```

Optimizing the Series of µOps. After traversing all of the nodes in the MIG and generating the appropriate series of µOps, we optimize the series of µOps by coalescing `AAP`/`AP` command sequences, which we can do in one of two cases.

Case 1: we can coalesce a series of row copy µOps if all of the µOps have the same µRegister source as an input. For example, consider a series of two `AAP`s that copy data array A into rows T2 and T3. We can coalesce this series of `AAP`s into a single `AAP` issued to the wordline address stored in µRegister B10 (see Figure 2.6a). This wordline address leverages the special row decoder in the B-group (which is part of the Ambit subarray structure [338]) to activate multiple DRAM rows in the group at once with a single activation command. For our example, activating µRegister B10 allows the `AAP` command sequence to copy array A into both rows T2 and T3 at once.

Case 2: we can coalesce an `AP` command sequence (i.e., a majority µOp) followed by an `AAP` command sequence (i.e., a row copy µOp) when the destination of the `AAP` is one of the rows used by the `AP`. For example, consider an `AP` that performs a MAJ logic primitive on DRAM rows T0, T1, and T2 (storing the result in all three rows), followed by an `AAP` that copies µRegister B12 (which refers to rows T0, T1, and T2) to row T3. The `AP` followed by the `AAP` puts the majority value in all four rows (T0, T1, T2, T3). The two command sequences can be coalesced into a single `AAP` (`AAP` T3, B12), as the first ACTIVATE would automatically perform the majority on rows T0, T1, and T2 by activating all three rows simultaneously. The second ACTIVATE then copies the value from those rows into T3.

Generalizing the Bit-Serial Operation into an n-bit Operation. Once all potential µOp coalescing is complete, the framework now has an optimized 1-bit version of the computation. We generalize this 1-bit µOp series into a loop body that repeats n times to implement an n-bit operation. We leverage the arithmetic and control µOps available in SIMDRAM to orchestrate the n-bit computation. Data produced by the computation of one bit that needs to be used for computation of the next bit (e.g., the carry bit in full addition) is kept in a B-group row across the two computations, allowing for bit-to-bit data transfer without the need for dedicated shifting circuitry.

The final series of µOps produced after this step is then packed into a µProgram and stored in DRAM for future use.[2] Figure 2.5c shows the final µProgram produced at the end of Step 2 for the full addition operation. The figure shows the optimized series of µOps that generates the 1-bit implementation of the full addition (lines 2–9), and the arithmetic and control µOps included to enable the n-bit implementation of the operation (lines 10–11).

Benefits of the µProgram Abstraction. The µProgram abstraction that we use to store SIMDRAM operations provides three main advantages to the framework. First,

[2]In our example implementation of SIMDRAM, a µProgram has a maximum size of 128 bytes, as this is enough to store the largest µProgram generated in our evaluations (the division operation, which requires 56 µOps, each two bytes wide, resulting in a total µProgram size of 112 bytes.)

it allows SIMDRAM to minimize the total number of new CPU instructions required to implement SIMDRAM operations, thereby reducing SIMDRAM's impact on the ISA. While a different implementation could use more new CPU instructions to express finer-grained operations (e.g., an `AAP`), we believe that using a minimal set of new CPU instructions simplifies adoption and software design. Second, the µProgram abstraction enables a smaller application binary size since the only information that needs to be placed in the application's binary is the address of the µProgram in main memory. Third, the µProgram provides an abstraction to relieve the end user from low-level programming with MAJ/NOT operations that is equivalent to programming with Boolean logic. We discuss how a user program invokes SIMDRAM µPrograms in Section 2.4.2.

2.3.3 Step 3: Operation Execution

Once the framework stores the generated µProgram for a SIMDRAM operation in DRAM, the SIMDRAM hardware can now receive program requests to execute the operation. To this end, we discuss the SIMDRAM *control unit*, which handles the execution of the µProgram at runtime. The control unit is designed as an extension of the memory controller, and is transparent to the programmer. A program issues a request to perform a SIMDRAM operation using a *bbop* instruction (introduced by Ambit [338]), which is one of the CPU ISA extensions to allow programs to interact with the SIMDRAM framework (see Section 2.4.2). Each SIMDRAM operation corresponds to a different *bbop* instruction. Upon receiving the request, the control unit loads the µProgram corresponding to the requested *bbop* from memory, and performs the µOps in the µProgram. Since all input data elements of a SIMDRAM operation may not fit in one DRAM row, the control unit repeats the µProgram i times, where i is the total number of data elements divided by the number of elements in a single DRAM row.

Figure 2.7 shows a block diagram of the SIMDRAM control unit, which consists of nine main components: (1) a *bbop FIFO* that receives the *bbop*s from the program, (2) a *µProgram Memory* allocated in DRAM (not shown in the figure), (3) a *µProgram Scratchpad* that holds commonly-used µPrograms, (4) a *µOp Memory* that holds the µOps of the currently running µProgram, (5) a *µRegister Addressing Unit* that generates the physical row addresses being used by the µRegisters that map to DRAM rows (based on the µRegister-to-row assignments for B0–B17 in Figure 2.6), (6) a *µRegister File* that holds the non-row-mapped µRegisters (B18–B31 in Figure 2.6), (7) a *Loop Counter* that tracks the number of remaining data elements that the µProgram needs to be performed on, (8) a µOp Processing FSM that controls the execution flow and issues `AAP`/`AP` command sequences, and (9) a µProgram counter (µPC). SIMDRAM reserves a region of DRAM for the µProgram Memory to store µPrograms corresponding to *all* SIMDRAM operations. At runtime, the control unit stores the most commonly used µPrograms in the µProgram Scratchpad, to reduce the overhead of fetching µPrograms from DRAM.

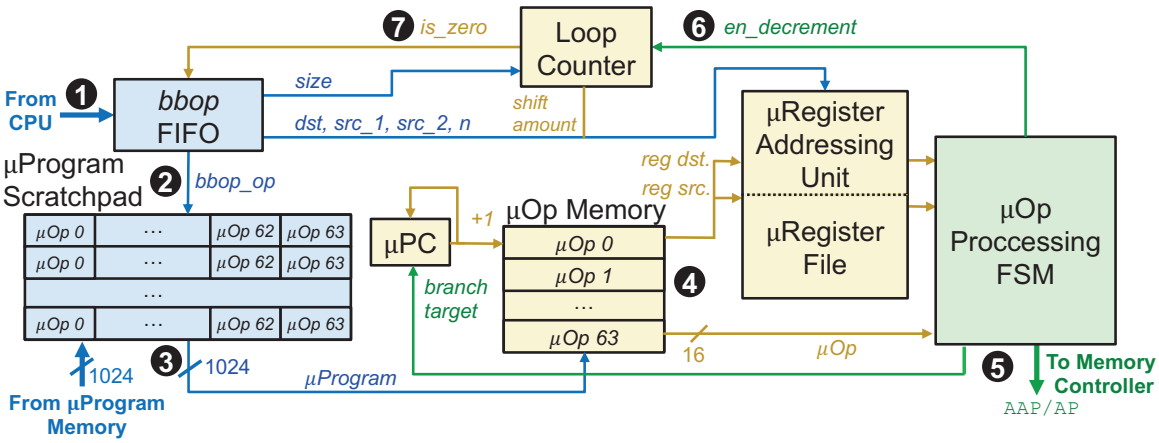


Figure 2.7: SIMDRAM control unit.

At runtime, when a CPU running a user program reaches a *bbop* instruction, it forwards the *bbop* to the SIMDRAM control unit (❶ in Figure 2.7). The control unit enqueues the *bbop* in the *bbop* FIFO. The control unit goes through a four-stage procedure to execute the queued *bbop*s one at a time.

In the first stage, the control unit fetches and decodes the *bbop* at the head of the FIFO (❷). Decoding a *bbop* involves (1) setting the index of the µProgram Scratchpad to the *bbop* opcode; (2) writing the number of loop iterations required to perform the operation on all elements (i.e., the number of data elements divided by the number of elements in a single DRAM row) into the Loop Counter; and (3) writing the base DRAM addresses of the source and destination arrays involved in the computation, and the size of each data element, to the µRegister Addressing Unit.

In the second stage, the control unit copies the µProgram currently indexed in the µProgram Scratchpad to the µOp Memory (❸). At this point, the control unit is ready to start executing the µProgram, one µOp at a time.

In the third stage, the current µOp is fetched from the µOp Memory, which is indexed by the µPC. The µOp Processing FSM decodes the µOp, and determines which µRegisters are needed (❹). For µRegisters B0–B17, the µRegister Addressing Unit generates the DRAM addresses that correspond to the requested registers (see Figure 2.6) and sends the addresses to the µOp Processing FSM. For µRegisters B18–B31, the µRegister File provides the register values to the µOp Processing FSM.

In the fourth stage, the µOp Processing FSM executes the µOp. If the µOp is a command sequence, the corresponding commands are sent to the memory controller's request queue (❺) and the µPC is incremented. If the µOp is a `done` control operation, this indicates that all of the command sequence µOps have been performed for the current iteration. The µOp Processing FSM then decrements the Loop Counter (❻). If the decremented Loop Counter is greater than zero, the µOp Processing FSM shifts the base source and destination addresses

stored in the µRegister Addressing Unit to move onto the next set of data elements,[3] and resets the µPC to the first µOp in the µOp Memory. If the decremented Loop Counter equals zero, this indicates that the control unit has completed executing the current *bbop*. The control unit then fetches the next *bbop* from the *bbop* FIFO (❼), and repeats all four stages for the next *bbop*.

2.3.4 Supported Operations

We use our framework to efficiently support a wide range of operations of different types. In this work, we evaluate (in Section 2.6) a set of 16 SIMDRAM operations of five different types for n-bit data elements: (1) N-input logic operations (OR-/AND-/XOR-reduction across N inputs); (2) relational operations (equality/inequality check, greater-/less-than check, greater-than-or-equal-to check, and maximum/minimum element in a set); (3) arithmetic operations (addition, subtraction, multiplication, division, and absolute value); (4) predication (if-then-else); and (5) other complex operations (bitcount, and ReLU). We support four different element sizes that correspond to data type sizes in popular programming languages (8-bit, 16-bit, 32-bit, 64-bit).

2.4 System Integration of SIMDRAM

We discuss several challenges of integrating SIMDRAM in a real system, and how we address them: (1) data layout and how SIMDRAM manages storing the data required for in-DRAM computation in a vertical layout (Section 2.4.1); (2) ISA extensions for and programming interface of SIMDRAM (Section 2.4.2); (3) how SIMDRAM handles page faults, address translation, coherence, and interrupts (Section 2.4.3); (4) how SIMDRAM manages computation on large amounts of data (Section 2.4.4); (5) security implications of SIMDRAM (Section 2.4.5); and (6) current limitations of the SIMDRAM framework (Section 2.4.6).

2.4.1 Data Layout

We envision SIMDRAM as *supplementing* (not *replacing*) the traditional processing elements. As a result, a program in a SIMDRAM-enabled system can have a combination of CPU instructions and SIMDRAM instructions, with possible data sharing between the two. However, while SIMDRAM operates on vertically-laid-out data (Section 2.2.3), the other system components (including the CPU) expect the data to be laid out in the traditional horizontal format, making it challenging to share data between SIMDRAM and CPU instructions. To address this challenge, memory management in SIMDRAM needs to (1) support both horizontal and vertical data layouts in DRAM *simultaneously*; and (2) transform vertically-laid-out data used by SIMDRAM to a horizontal layout for CPU

[3]The source and destination base addresses are incremented by n rows, where n is the data element size. This is because each DRAM row contains one bit of a set of elements, so SIMDRAM uses n consecutive rows to hold all n bits of the set of elements.

use, and vice versa. We cannot rely on software (e.g., compiler or application support) to handle the data layout transformation, as this would go through the on-chip memory controller, and would introduce significant data movement, and thus latency, between the DRAM and CPU during the transformation. To avoid data movement during transformation, SIMDRAM uses a specialized hardware unit placed between the last-level cache (LLC) and the memory controller, called the *data transposition unit*, to transform data from horizontal data layout to vertical data layout, and vice versa. The transposition unit ensures that for every SIMDRAM object, its corresponding data is in a horizontal layout whenever the data is in the cache, and in a vertical layout whenever the data is in DRAM.

Figure 2.8 shows the key components of the transposition unit. The transposition unit keeps track of the memory objects that are used by SIMDRAM operations in a small cache in the transposition unit, called the *Object Tracker*. To add an entry to the Object Tracker when allocating a memory object used by SIMDRAM, the programmer adds an initialization instruction called `bbop_trsp_init` (Section 2.4.2) *immediately* after the `malloc` that allocates the memory object (❶ in Figure 2.8). Assuming a system that employs lazy allocation, the `bbop_trsp_init` instruction informs the operating system (OS) that the memory object is a *SIMDRAM object*. This allows the OS to perform virtual-to-physical memory mapping optimizations for the object before the allocation starts (e.g., mapping the arguments of an operation to the same row/column in the physical memory). When the SIMDRAM object's physical memory is allocated, the OS inserts the base physical address, the total size of the allocated data, and the size of each element in the object (provided by `bbop_trsp_init`) into the Object Tracker. As the initially-allocated data is placed in the CPU cache, the data starts in a horizontal layout until it is evicted from the cache.

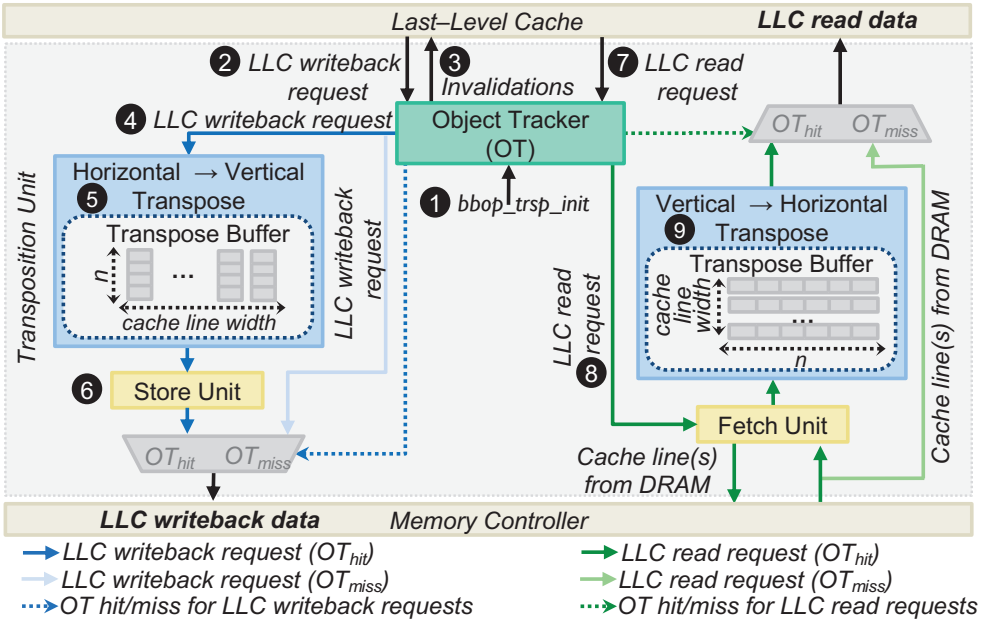


Figure 2.8: Major components of the data transposition unit.

SIMDRAM stores SIMDRAM objects in DRAM using a vertical layout, since this is the layout used for in-DRAM computation (Section 2.2.3). Since a vertically-laid-out n-bit element spans n different cache lines in DRAM (with each cache line in a different DRAM row), SIMDRAM partitions SIMDRAM objects into *SIMDRAM object slices*, each of which is n cache lines in size. Thus, a SIMDRAM object slice in DRAM contains the vertically-laid-out bits of as many elements as bits in a cache line (e.g., 512 in a 64 B cache line). Cache line i ($0 \leq i < n$) of an object slice contains bit i of *all* elements stored in the slice. Whenever any one data element within a slice is requested by the CPU, the entire SIMDRAM object slice is brought into the LLC. Similarly, whenever a cache line from a SIMDRAM object is written back from the LLC to DRAM (i.e., it is evicted or flushed), *all* $n-1$ remaining cache lines of the same SIMDRAM object slice are written back as well.[4] The use of object slices ensures correctness and simplifies the transposition unit.

Whenever the LLC writes back a cache line to DRAM (❷ in Figure 2.8), the transposition unit checks the Object Tracker to see whether the cache line belongs to a SIMDRAM object. If the LLC request misses in the Object Tracker, the cache line does not belong to any SIMDRAM object, and the writeback request is forwarded to the memory controller as in a conventional system. If the LLC request hits in the Object Tracker, the cache line belongs to a SIMDRAM object, and thus must be transposed from the horizontal layout to the vertical layout. An Object Tracker hit triggers two actions.

First, the Object Tracker issues invalidation requests to *all* $n-1$ remaining cache lines of the same SIMDRAM object slice (❸ in Figure 2.8).[4] We extend the LLC to support a special invalidation request type, which sends both dirty *and* unmodified cache lines to the transposition unit (unlike a regular invalidation request, which simply invalidates unmodified cache lines). The Object Tracker issues these invalidation requests for the remaining cache lines, ensuring that *all* cache lines of the object slice arrive at the transposition unit to perform the horizontal-to-vertical transposition correctly.

Second, the writeback request is forwarded (❹ in Figure 2.8) to a horizontal-to-vertical transpose buffer, which performs the bit-by-bit transposition. We design the transpose buffer (❺) such that it can transpose all bits of a horizontally-laid-out cache line in a single cycle. As the other cache lines belonging from the slice are evicted (as a result of the Object Tracker's invalidation requests) and arrive at the transposition unit, they too are forwarded to the transpose buffer, and their bits are transposed. Each horizontally-laid-out cache line maps to a specific set of bit columns in the vertically-laid-out cache line, which is determined using the physical address of the horizontally-laid-out cache line. Once all n cache lines in the SIMDRAM object slice have been transposed, the Store Unit generates DRAM write requests for each vertically-laid-out cache line, and sends the requests to the memory controller (❻).

[4]The Dirty-Block Index [334] could be adapted for this purpose.

When a program wants to read data that belongs to a SIMDRAM object, and the data is not in the CPU caches, the LLC issues a read request to DRAM (❼ in Figure 2.8). If the address of the read request does not hit in the Object Tracker, the request is forwarded to the memory controller, as in a conventional system. If the address of the read request hits in the Object Tracker, the read request is part of a SIMDRAM object, and the Object Tracker sends a signal (❽) to the Fetch Unit. The Fetch Unit generates the read requests for *all* of the vertically-laid-out cache lines that belong to the same SIMDRAM object slice as the requested data, and sends these requests to the memory controller. When the request responses for the object slice's cache lines arrive, the Fetch Unit sends the cache lines to a vertical-to-horizontal transpose buffer (❾), which can transpose all bits of one vertically-laid-out cache line into the horizontally-laid-out cache lines in one cycle. The horizontally-laid-out cache lines are then inserted into the LLC. The $n-1$ cache lines that were *not* part of the original memory request, but belong to the same object slice, are inserted into the LLC in a manner similar to conventional prefetch requests [361].

2.4.2 ISA Extensions and Programming Interface

The lack of an efficient and expressive programmer/system interface can negatively impact the performance and usability of the SIMDRAM framework. This would put data transposition on the critical path of SIMDRAM computation, which would cause large performance overheads. To address such issues and to enable the programmer/system to efficiently communicate with SIMDRAM, we extend the ISA with specialized SIMDRAM instructions. The main goal of the SIMDRAM ISA extensions is to let the SIMDRAM control unit know (1) what SIMDRAM operations need to be performed and when, and (2) what the SIMDRAM memory objects are and when to transpose them.

Table 2.1 shows the CPU ISA extensions that the SIMDRAM framework exposes to the programmer. There are three types of instructions: (1) SIMDRAM object initialization instructions, (2) instructions to perform different SIMDRAM operations, and (3) predication instructions. We discuss `bbop_trsp_init`, our only SIMDRAM object initialization instruction, in Section 2.4.1. The CPU ISA extensions for performing SIMDRAM operations can be further divided into two categories: (1) operations with one input operand (e.g., bitcount, ReLU), and (2) operations with two input operands (e.g., addition, division, equal, maximum). SIMDRAM uses an array-based computation model, and `src` (i.e., `src` in 1-input operations and `src_1, src_2` in 2-input operations) and `dst` in these instructions represent source and destination *arrays*. `bbop_op` represents the opcode of the SIMDRAM operation, while `size` and `n` represent the number of elements in the source and destination arrays, and the number of bits in each array element, respectively. To enable predication, SIMDRAM uses the `bbop_if_else` instruction in which, in addition to two source and one destination arrays, `select` represents the predicate array (i.e., the predicate, or mask, bits).

Listing 2.1 shows how SIMDRAM's CPU ISA extensions can be used to perform in-DRAM computation, with an example code that performs element-wise addition or subtraction of

Table 2.1: SIMDRAM ISA extensions.

Type	ISA Format
Initialization	`bbop_trsp_init address, size, n`
1-Input Operation	`bbop_op dst, src, size, n`
2-Input Operation	`bbop_op dst, src_1, src_2, size, n`
Predication	`bbop_if_else dst, src_1, src_2, select, size, n`

two arrays (`A` and `B`) depending on the comparison of each element of `A` to the corresponding element of a third array (`pred`). Listing 2.1a shows the original C code for the computation, while Listing 2.1b shows the equivalent code using SIMDRAM operations. The lines that perform the same operations are highlighted using the same colors in both C code and SIMDRAM code. The if-then-else condition in C code is performed in SIMDRAM using a predication instruction (i.e., `bbop_if_else` on line 16 in Listing 2.1b). SIMDRAM treats the if-then-else condition as a multiplexer. Accordingly, `bbop_if_else` takes two source arrays and a predicate array as inputs, where the predicate is used to choose which source array should be selected as the output at the corresponding index. To this end, we first allocate two arrays to hold the addition and subtraction results (i.e., arrays `D` and `E` on line 10 in Listing 2.1b), and then populate them using `bbop_add` and `bbop_sub` (lines 13 and 14 in Listing 2.1b), respectively. We then allocate the predicate array (i.e., array `F` on line 11 in Listing 2.1b) and populate it using `bbop_greater` (line 15 in Listing 2.1b). The addition, subtraction, and predicate arrays form the three inputs (arrays `D`, `E`, `F`) to the `bbop_if_else` instruction (line 16 in Listing 2.1b), which stores the outcome of the predicated execution to the destination array (i.e., array `C` in Listing 2.1b).

In this work, we assume that the programmer manually rewrites the code to use SIMDRAM operations. We follow this approach when evaluating real-world applications in Section 2.6.3. We envision two programming models for SIMDRAM. In the first programming model, SIMDRAM operations are encapsulated within userspace library routines to ease programmability. With this approach, the programmer can optimize the SIMDRAM-based code to make the most out of the underlying in-DRAM computing mechanism. In the second programming model, SIMDRAM operations are transparently inserted within the application's binary using compiler assistance. Since SIMDRAM is a SIMD-like compute engine, we expect that the compiler can generate SIMDRAM code without programmer intervention in at least two ways. First, it can leverage auto-vectorization routines already present in modern compilers [99, 242] to generate SIMDRAM code, by setting the width of the SIMD lanes equivalent to a DRAM row. For example, in LLVM [208], the width of the SIMD units can be defined using the `"-force-vector-width"` flag [242]. A SIMDRAM-based compiler back-end can convert the LLVM intermediate representation instructions into *bbop* instructions. Second, the compiler can compose groups of existing SIMD instructions generated by the compiler (e.g., AVX2 instructions [95]) into blocks that match the size of a DRAM row, and then convert such instructions into a single SIMDRAM operation. Prior

```c
int size = 65536;
int elm_size = sizeof(uint8_t);
uint8_t *A, *B, *C = (uint8_t*)
    malloc(size*elm_size);
uint8_t *pred = (uint8_t*)malloc(
    size*elm_size);
...
for(int i = 0; i < size; ++i) {
    bool cond = A[i] > pred[i];
    if (cond)
        C[i] = A[i] + B[i];
    else
        C[i] = A[i] - B[i];
}
```

(a) C code for vector add/sub with predicated execution

```c
int size = 65536;
int elm_size = sizeof(uint8_t);
uint8_t *A, *B, *C = (uint8_t*)
    malloc(size*elm_size);

bbop_trsp_init(A,size,elm_size);
bbop_trsp_init(B,size,elm_size);
bbop_trsp_init(C,size,elm_size);
uint8_t *pred = (uint8_t*)malloc(
    size*elm_size);
// D, E, F store intermediate data
uint8_t *D, *E = (uint8_t*)malloc(
    size*elm_size);
bool *F = (bool*)malloc(size*
    sizeof(bool));
...
bbop_add(D, A, B, size, elm_size);
bbop_sub(E, A, B, size, elm_size);
bbop_greater(F, A, pred, size,
    elm_size);
bbop_if_else(C, D, E, F, size,
    elm_size);
```

(b) Equivalent code using SIMDRAM operations

Listing 2.1: Example code using SIMDRAM instructions.

work [6] uses a similar approach for 3D-stacked PIM. We leave the design of a compiler for SIMDRAM for future work.

SIMDRAM instructions can be implemented by extending the ISA of the host CPU. This is possible since there is enough unused opcode space to support the extra opcodes that SIMDRAM requires. To illustrate, prior works [244, 245] show that there are 389 unused operation codes considering only the AVX and SSE extensions for the x86 ISA. Extending the instruction set is a common approach to interface a CPU with PIM architectures [7, 338].

2.4.3 Handling Page Faults, Address Translation, Coherence, and Interrupts

SIMDRAM handles four key system mechanisms as follows:

- *Page Faults:* We assume that the pages that are touched during in-DRAM computation are already present and pinned in DRAM. In case the required data is not present in DRAM, we rely on the conventional page fault handling mechanism to bring the required pages into DRAM.
- *Address Translation:* Virtual memory and address translation are challenging for many PIM architectures [8, 113, 309]. SIMDRAM is relieved of such challenge as it operates directly on physical addresses. When the CPU issues a SIMDRAM instruction, the instruction's virtual memory addresses are translated into their corresponding physical

addresses using the same translation lookaside buffer (TLB) lookup mechanisms used by regular load/store operations.

- *Coherence:* Input arrays to SIMDRAM may be generated or modified by the CPU, and the data updates may reside only in the cache (e.g., because the updates have not yet been written back to DRAM). To ensure that SIMDRAM does not operate on stale data, programmers are responsible for flushing cache lines [23, 159] modified by the CPU. SIMDRAM can leverage coherence optimizations tailored to PIM to improve overall performance [53, 54].
- *Interrupts:* Two cases where an interrupt could affect the execution of a SIMDRAM operation are (1) on an application context switch, and (2) on a page fault. In case of a context switch, the control unit's context needs to be saved and then restored later when the application resumes execution. We do not expect to encounter a page fault during the execution of a SIMDRAM operation since, as previously mentioned, pages touched by SIMDRAM operations are expected to be loaded into and pinned in DRAM.

2.4.4 Handling Limited Subarray Size

SIMDRAM operates on data placed within the same subarray. However, a single subarray stores only several megabytes of data. For example, a subarray with 1024 rows and a row size of 8 kB can only store 8 MB of data. Therefore, SIMDRAM needs to use a mechanism that can efficiently move data within DRAM (e.g., across DRAM banks and subarrays). SIMDRAM can exploit (1) RowClone Pipelined Serial Mode (PSM) [336] to copy data between two banks by using the internal DRAM bus, or (2) Low-Cost Inter-Linked Subarrays (LISA) [66] to copy rows between two subarrays within the same bank. We evaluate the performance overheads of using both mechanisms in Section 2.6.6. Other mechanisms for fast in-DRAM data movement [345, 380] can also enhance SIMDRAM's capability.

2.4.5 Security Implications

SIMDRAM and other similar in-DRAM computation mechanisms that use dedicated DRAM rows to perform computation may increase vulnerability to RowHammer attacks [101, 187, 193, 269, 274]. We believe, and the literature suggests, that there should be robust and scalable solutions to RowHammer, orthogonally to our work (e.g., BlockHammer [390], PARA [192], TWiCe [220], Graphene [298]). Exploring RowHammer prevention and mitigation mechanisms in conjunction with SIMDRAM (or other PIM approaches) requires special attention and research, which we leave for future work.

2.4.6 SIMDRAM Limitations

We note three key limitations of the current version of the SIMDRAM framework:

- *Floating-Point Operations:* SIMDRAM supports only integer and fixed-point operations. Enabling floating-point operations in-DRAM while maintaining low area overheads is a challenge. For example, for floating-point addition, the IEEE 754 FP32 format [153] requires shifting the mantissa by the difference of the exponents of elements. Since each

bitline stores a data element in SIMDRAM, shifting the value stored in one bitline without compromising the values stored in other bitlines at low cost is currently infeasible.

- *Operations That Require Shuffling Data Across Bitlines:* Different from prior works (e.g., DRISA [228]), SIMDRAM does *not* add any extra circuitry to perform bit-shift operations. Instead, SIMDRAM stores data in a vertical layout and can perform explicit bit-shift operations (if needed) by orchestrating row copies. Even though this approach enables SIMDRAM to implement a large range of operations, it is not possible to perform shuffling and reduction operations *across bitlines* without the inclusion of dedicated bit-shifting circuitry. This is due to the lack of physical connections across bitlines, which can be solved by building a bit-shift engine near the sense amplifiers.
- *Synchronization Between Concurrent In-DRAM Operations:* SIMDRAM can be easily modified to enable concurrent execution of *distinct operations* across different subarrays in DRAM. However, this would require the implementation of software or hardware synchronization primitives to orchestrate the computation of a single task across different subarrays. Ideas that are similar to SynCron [116] can be beneficial.

2.5 Methodology

We implement SIMDRAM using the gem5 simulator [48] and compare it to a real multicore CPU (Intel Skylake [158]), a real high-end GPU (NVIDIA Titan V [285]), and a state-of-the-art processing-using-DRAM mechanism (Ambit [338]). In all our evaluations, the CPU code is optimized to leverage AVX-512 instructions [95]. Table 2.2 shows the system parameters we use in our evaluations. To measure CPU performance, we implement a set of timers in `sys/time.h` [370]. To measure CPU energy consumption, we use Intel RAPL [130]. To measure GPU performance, we implement a set of timers using the `cudaEvents` API [70]. We capture GPU kernel execution time that excludes data initialization/transfer time. To measure GPU energy consumption, we use the `nvml` API [284]. We report the average of five runs for each CPU/GPU data point, each with a warmup phase to avoid cold cache effects. We implement Ambit on gem5 and validate our implementation rigorously with the results reported in [338]. We use the same vertical data layout in our Ambit and SIMDRAM implementations, which enables us to (1) evaluate all 16 SIMDRAM operations in Ambit using their equivalent AND/OR/NOT-based implementations, and (2) highlight the benefits of Step 1 in the SIMDRAM framework (i.e., using an optimized MAJ/NOT-based implementation of the operations). Our synthetic throughput analysis (Section 2.6.1) uses 64M-element input arrays.

We evaluate three different configurations of SIMDRAM where 1 (*SIMDRAM:1*), 4 (*SIMDRAM:4*), and 16 (*SIMDRAM:16*) banks out of all the banks in one channel (16 banks in our evaluations) have SIMDRAM computation capability. In the SIMDRAM 1-bank configuration, our mechanism exploits 65536 (i.e., size of an 8 kB row buffer) SIMD lanes. Conventional DRAM architectures exploit bank-level parallelism (BLP) to maximize DRAM

Table 2.2: Evaluated system configurations.

Intel Skylake CPU [158]	x86 [159], 16 cores, 8-wide, out-of-order, 4 GHz; *L1 Data + Inst. Private Cache:* 32 kB, 8-way, 64 B line; *L2 Private Cache:* 256 kB, 4-way, 64 B line; *L3 Shared Cache:* 8 MB, 16-way, 64 B line; *Main Memory:* 32 GB DDR4-2400, 4 channels, 4 ranks
NVIDIA Titan V GPU [285]	6 graphics processing clusters, 5120 CUDA Cores; 80 streaming multiprocessors, 1.2 GHz base clock; *L2 Cache:* 4.5 MB L2 Cache; *Main Memory:* 12 GB HBM [166, 215]
Ambit [338] and SIMDRAM	gem5 system emulation; x86 [159], 1-core, out-of-order, 4 GHz; *L1 Data + Inst. Cache:* 32 kB, 8-way, 64 B line; *L2 Cache:* 256 kB, 4-way, 64 B line; *Memory Controller:* 8 kB row size, FR-FCFS [275, 404] scheduling *Main Memory:* DDR4-2400, 1 channel, 1 rank, 16 banks

throughput [194–196, 213, 276]. The memory controller can issue commands to different banks (one-per-cycle) on the same channel such that banks can operate in parallel. In SIMDRAM, banks in the same channel can operate in parallel, just like conventional banks. Therefore, to enable the required parallelism, SIMDRAM requires no more modifications. Accordingly, the number of available SIMD lanes, i.e., SIMDRAM's computation capability, increases by exploiting BLP in SIMDRAM configurations (i.e., the number of available SIMD lanes in the 16-bank configuration is 16×65536).[5]

2.6 Evaluation

We demonstrate the advantages of the SIMDRAM framework by evaluating: (1) SIMDRAM's throughput and energy consumption for a wide range of operations; (2) SIMDRAM's performance benefits on real-world applications; (3) SIMDRAM's performance and energy benefits over a closely-related processing-using-cache architecture [103]; and (4) the reliability of SIMDRAM operations. Finally, we evaluate three key overheads in SIMDRAM: in-DRAM data movement, data transposition, and area cost.

2.6.1 Throughput Analysis

Figure 2.9 (left) shows the normalized throughput of all 16 SIMDRAM operations (Section 2.3.4) compared to those on CPU, GPU, and Ambit (normalized to the multicore CPU throughput), for an element size of 32 bits. We provide the absolute throughput of the baseline CPU (in GOps/s) in each graph. We classify each operation based on how the latency of the operation scales with respect to element size n.[6] Class 1, 2, and 3 operations scale linearly, logarithmically, and quadratically with n, respectively. Figure 2.9 (right) shows how the average throughput across all operations of the same class scales relative to element size. We evaluate element sizes of 8, 16, 32, 64 bits. We normalize the figure to the average throughput on a CPU.

[5]SIMDRAM computation capability can be further increased by enabling and exploiting subarray-level parallelism in each bank [65, 66, 173, 195].

[6]Appendix C discusses the scalability of each operation.

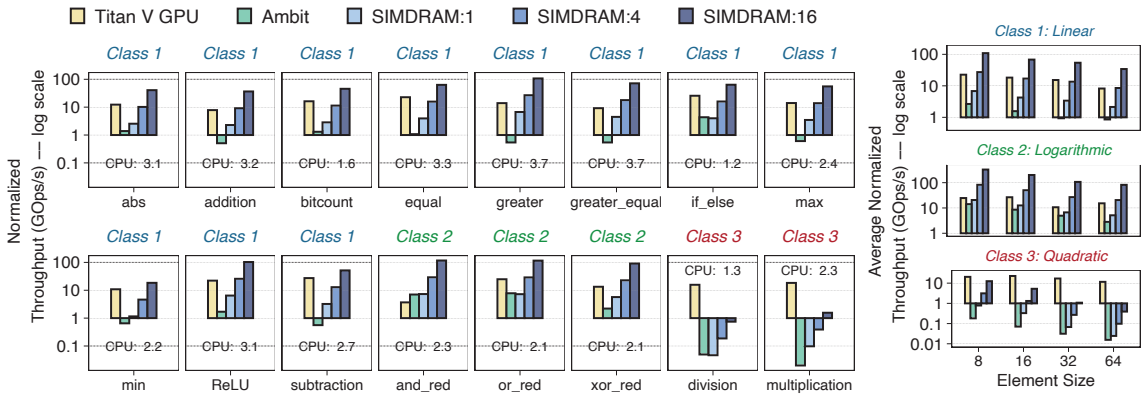


Figure 2.9: Normalized throughput of 16 operations. SIMDRAM:X uses X DRAM banks for computation.

We make four observations from Figure 2.9. First, we observe that SIMDRAM outperforms the three state-of-the-art baseline systems i.e., CPU/GPU/Ambit. Compared to CPU/GPU, SIMDRAM's throughput is 5.5×/0.4×, 22.0×/1.5×, and 88.0×/5.8× that of the CPU/GPU, averaged across all 16 SIMDRAM operations for 1, 4, and 16 banks, respectively. To ensure fairness, we compare Ambit, which uses a single DRAM bank in our evaluations, only against *SIMDRAM:1*.[7] Our evaluations show that *SIMDRAM:1* outperforms Ambit by 2.0×, averaged across all 16 SIMDRAM operations. Second, SIMDRAM outperforms the GPU baseline when we use more than four DRAM banks for *all* the linear and logarithmic operations. *SIMDRAM:16* provides 5.7× (9.3×) the throughput of the GPU across all linear (logarithmic) operations, on average. *SIMDRAM:16*'s throughput is 83× (189×) and 45.2× (19.9×) that of CPU and Ambit, respectively, averaged across all linear (logarithmic) operations. Third, we observe that both the multicore CPU baseline and GPU outperform *SIMDRAM:1*, *SIMDRAM:4*, and *SIMDRAM:16 only* for the division and multiplication operations. This is due to the quadratic nature of our bit-serial implementation of these two operations. Fourth, as expected, we observe a drop in the throughput for *all* operations with increasing element size, since the latency of each operation increases with element size. We conclude that SIMDRAM significantly outperforms all three state-of-the-art baselines for a wide range of operations.

2.6.2 Energy Analysis

We use CACTI [265] to evaluate SIMDRAM's energy consumption. Prior work [338] shows that each additional simultaneous row activation increases energy consumption by 22%. We use this observation in evaluating the energy consumption of SIMDRAM, which requires TRAs. Figure 2.10 compares the energy efficiency (Throughput per Watt) of SIMDRAM against the GPU and Ambit baselines, normalized to the CPU baseline. We provide the absolute Throughput per Watt of the baseline CPU in each graph. We make four observations.

[7] Ambit's throughput scales proportionally to bank count, just like SIMDRAM's.

First, SIMDRAM significantly increases energy efficiency for *all* operations over *all* three baselines. SIMDRAM's energy efficiency is 257×, 31×, and 2.6× that of CPU, GPU, and Ambit, respectively, averaged across all 16 operations. The energy savings in SIMDRAM directly result from (1) avoiding the costly off-chip round-trips to load/store data from/to memory, (2) exploiting the abundant memory bandwidth within the memory device, reducing execution time, and (3) reducing the number of TRAs required to compute a given operation by leveraging an optimized majority-based implementation of the operation. Second, similar to our results on throughput (Section 2.6.1), the energy efficiency of SIMDRAM reduces as element size increases. However, the energy efficiency of the CPU or GPU does not. This is because (1) for all SIMDRAM operations, the number of TRAs increases with element size; and (2) CPU and GPU can fully utilize their wider arithmetic units with larger (i.e., 32- and 64-bit) element sizes. Third, even though SIMDRAM multiplication and division operations scale poorly with element size, the SIMDRAM implementations of these operations are significantly more energy-efficient compared to the CPU and GPU baselines, making SIMDRAM a competitive candidate even for multiplication and division operations. Fourth, since *both* SIMDRAM's throughput and power consumption increase proportionally to the number of banks, the Throughput per Watt for SIMDRAM 1-, 4-, and 16-bank configurations is the same. We conclude that SIMDRAM is more energy-efficient than all three state-of-the-art baselines for a wide range of operations.

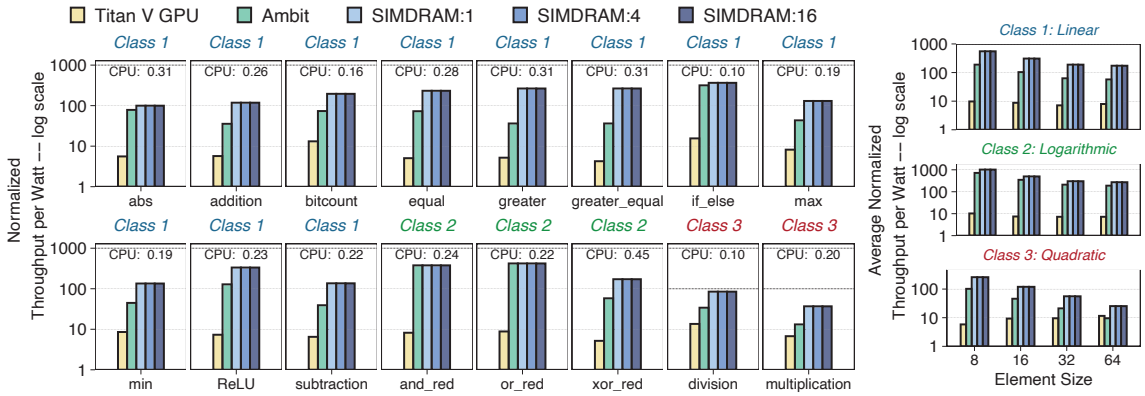


Figure 2.10: Normalized energy efficiency of 16 operations.

2.6.3 Effect on Real-World Kernels

We evaluate SIMDRAM with a set of kernels that represent the behavior of selected important real-world applications from different domains. The evaluated kernels come from databases (TPC-H query 1 [372], BitWeaving [233]), convolutional neural networks (LeNET-5 [209], VGG-13 [352], VGG-16 [352]), classification algorithms (k-nearest neighbors [222]), and image processing (brightness [119]). These kernels rely on many of the basic operations we evaluate in Section 2.6.1. We provide a brief description of each kernel and the SIMDRAM operations that they utilize in Appendix D.

Figure 2.11 shows the performance of SIMDRAM and our baseline configurations for each kernel, normalized to that of the multicore CPU. We make four observations. First, *SIMDRAM:16* greatly outperforms the CPU and GPU baselines, providing 21× and 2.1× the performance of the CPU and GPU, respectively, on average across all seven kernels. SIMDRAM has a maximum performance of 65× and 5.4× that of the CPU and GPU, respectively (for the BitWeaving kernel in both cases). Similarly, *SIMDRAM:1* provides 2.5× the performance of Ambit (which also uses a single bank for in-DRAM computation), on average across all seven kernels, with a maximum of 4.8× the performance of Ambit for the TPC-H kernel. Second, even with a single DRAM bank, SIMDRAM *always* outperforms the CPU baseline, providing 2.9× the performance of the CPU on average across all kernels. Third, *SIMDRAM:4* provides 2× and 1.1× the performance of the GPU baseline for the BitWeaving and brightness kernels, respectively. Fourth, despite GPU's higher multiplication throughput compared to SIMDRAM (Section 2.6.1), *SIMDRAM:16* outperforms the GPU baseline even for kernels that heavily rely on multiplication (Appendix D) (e.g., by 1.03× and 2.5× for kNN and TPC-H kernels, respectively). This speedup is a direct result of exploiting the high in-DRAM bandwidth in SIMDRAM to avoid the memory bottleneck in GPU caused by the large amounts of intermediate data generated in such kernels. We conclude that SIMDRAM is an effective and efficient substrate to accelerate many commonly-used real-world applications.

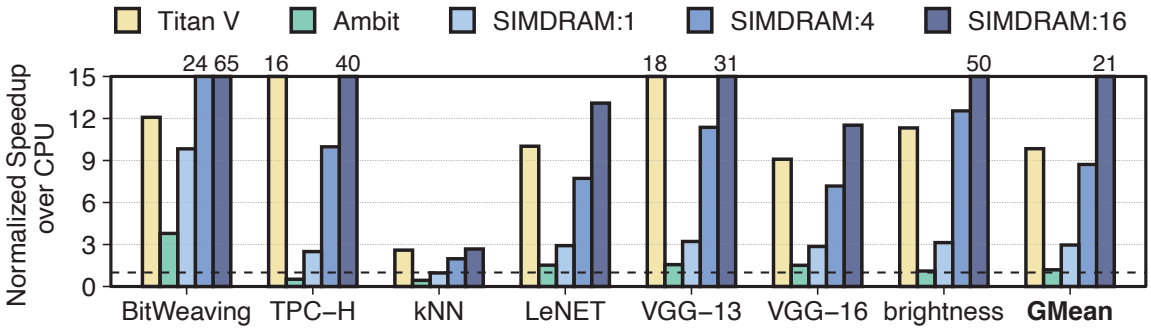


Figure 2.11: Normalized speedup of real-world kernels.

2.6.4 Comparison to DualityCache

We compare SIMDRAM to DualityCache [103], a closely-related processing-using-cache architecture. DualityCache is an in-cache computing framework that performs computation using discrete logic elements (e.g., logic gates, latches, muxes) that are added to the SRAM peripheral circuitry. In-cache computing approaches (such as DualityCache) need data to be brought into the cache first, which requires extra data movement (and even more if the working set of the application does not fit in the cache) compared to in-memory computing approaches (like SIMDRAM).

Figure 2.12 (top) compares the latency of SIMDRAM against DualityCache [103] for the subset of operations that *both* SIMDRAM and DualityCache support (i.e., addition, subtraction, multiplication, and division). In this experiment, we study three different

configurations. First, *DualityCache:Ideal* has *all* data required for DualityCache residing in the cache. Therefore, results for *DualityCache:Ideal* do *not* include the overhead of moving data from DRAM to the cache, making it an unrealistic configuration that needs the data to already reside and fit in the cache. Second, *DualityCache:Realistic* includes the overhead of data movement from DRAM to the cache. Both DualityCache configurations compute on an input array of 45 MB. Third, *SIMDRAM:16*. For all three configurations, we use the same cache size (35 MB) as the original DualityCache work [103] to provide a fair comparison. As shown in the figure, SIMDRAM greatly outperforms DualityCache when data movement is realistically taken into account. *SIMDRAM:16* outperforms *DualityCache:Realistic* for all four operations (by 52.9×, 52.4×, 1.8×, and 2.1× for addition, subtraction, multiplication, and division respectively, on average across all element sizes). SIMDRAM's performance improvement comes at a much lower area overhead compared to DualityCache. DualityCache (including its peripherals, transpose memory unit, controller, miss status holding registers, and crossbar network) has an area overhead of 3.5% in a high-end CPU, whereas SIMDRAM has an area overhead of only 0.2% (Section 2.6.8). As a result, SIMDRAM can actually fit a significantly higher number of SIMD lanes in a given area compared to DualityCache. Therefore, SIMDRAM's performance improvement per unit area would be much larger than that we observe in Figure 2.12. We conclude that SIMDRAM achieves higher performance at lower area cost over DualityCache, when we consider DRAM-to-cache data movement.

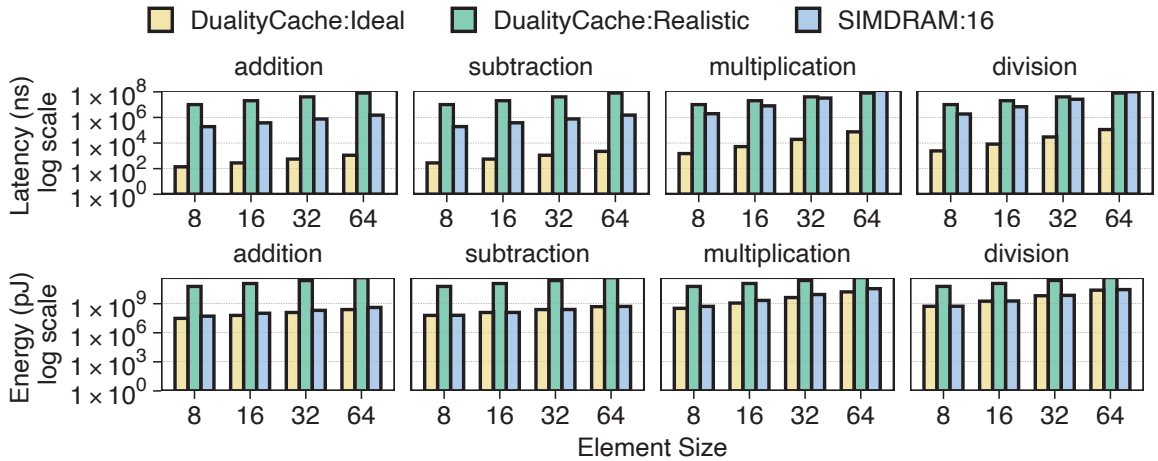


Figure 2.12: Latency and energy to execute 64M operations.

Figure 2.12 (bottom) shows the energy consumption of *DualityCache:Realistic*, *DualityCache:Ideal*, and *SIMDRAM:16* when performing 64M addition, subtraction, multiplication, and division operations. We make two observations. First, compared to *DualityCache:Ideal*, *SIMDRAM:16* increases average energy consumption by 60%. This is because while the energy per bit to perform computation in DRAM (13.3 nJ/bit [262, 379]) is smaller than the energy per bit to perform computation in the cache (60.1 nJ/bit [85]), the DualityCache implementation of each operation requires fewer iterations than its equivalent SIMDRAM implementation. Second, *SIMDRAM:16* reduces average energy by 600× over *Duality-*

Cache:Realistic because *DualityCache:Realistic* needs to load *all* input data from DRAM, incurring high energy overhead (a DRAM access consumes 650× the energy-per-bit of a DualityCache operation [85, 103]). In contrast, SIMDRAM operates on data that is already present in DRAM, eliminating any data movement overhead. We conclude that SIMDRAM is much more efficient than DualityCache, when cache-to-DRAM data movement is realistically considered.

2.6.5 Reliability

We use SPICE simulations to test the reliability of SIMDRAM for different technology nodes and varying amounts of process variation. At the core of SIMDRAM, there are two back-to-back triple-row activations (TRAs). Table 2.3 shows the characteristics of TRA and two back-to-back TRAs (TRAb2b) for the 45, 32, and 22 nm technology nodes. We compare these with the reliability of quintuple-row activations (QRAs), used by prior works [11, 19] to implement bit-serial addition. We use the reference 55 nm DRAM model from Rambus [316] and scale it based on the ITRS roadmap [163, 379] to model smaller technology nodes following the PTM transistor models [282]. The goal of our analysis is to understand the reliability trends for TRA and QRA operations with technology scaling. For each technology node and process variation amount, we run Monte-Carlo simulations for 10^4 iterations.

Table 2.3: Process variation's effect on TRA/QRA failure rates.

	Variation (%)	**± 0**	**± 5**	**± 10**	**± 20**
45 nm	TRA Failure (%)	0	0	0.02	3.01
	TRAb2b Failure (%)	0	0	0.04	5.93
	QRA Failure (%)	0	0	0.35	6.54
32 nm	TRA Failure (%)	0	0	0.35	3.90
	TRAb2b Failure (%)	0	0	0.69	7.64
	QRA Failure (%)	0	0.42	6.33	11.52
22 nm	TRA Failure (%)	0	0	0.42	4.50
	TRAb2b Failure (%)	0	0	0.84	8.83
	QRA Failure (%)	error	error	error	error

We make four observations. First, for all process variation ranges, TRA and TRAb2b perform more reliably than QRA. Specifically, TRA and TRAb2b perform without errors for 5% variation. Second, while moving from 45 nm to 32 nm, we observe that the error rate of QRA increases faster than than that of TRA, making QRA less reliable as the technology node size reduces. Third, for TRA and TRAb2b in 22 nm, we observe a similar trend of increased error rate while still having zero error rate for 5% process variation. In our simulations, QRA does not perform correctly in the projected 22 nm DRAM. For example, `MAJ(11100)` always leads to the incorrect outcome of `'0'`. This is because charge sharing between five capacitors in QRA does not lead to enough voltage on the bitline for the sense amplifier to pull up the bitline to the value `'1'`. We believe that proposals based on QRA require changes to the circuit elements (e.g., transistors in the sense amplifier) to enable correct operation in the 22 nm technology node. Fourth, a TRA can fail depending on the amount of manufacturing process variation. We observe that a TRA starts to fail when process variation is larger than

10%, for all technology nodes. Since SIMDRAM operations are executed within a DRAM module, it is quite challenging to leverage existing in-DRAM or in-memory-controller error correction mechanisms [249, 260, 299, 300]. The same problem exists for other processing-using-DRAM mechanisms [11, 19, 77, 82, 227, 228, 335, 336, 338, 340–342, 364, 389]. We conclude that the TRA operations SIMDRAM relies on are much more scalable and variation-tolerant than QRA operations some prior works rely on. We leave a study of reliability solutions for future work.

2.6.6 Data Movement Overhead

There may be cases where the output of a SIMDRAM operation that is used as an input to a subsequent operation does not reside in the same subarray as other inputs. For example, consider the computation $C = OP(A, B)$. If the output of the SIMDRAM operation OP is an input to a subsequent SIMDRAM operation, C needs to move to the same subarray as the other inputs of the subsequent operation, before the operation can start. Figure 2.13 shows the distribution of the worst-case latency overhead of moving the output of each of our 16 SIMDRAM operations with 8-, 16-, 32-, and 64-bit element sizes in *SIMDRAM:1* to a different subarray within the same bank, i.e., *intra-bank* (using LISA [66]) or a different bank, i.e., *inter-bank* (using RowClone PSM [336]). We make two observations. First, intra-bank data movement (Figure 2.13, left) results in only 0.39% latency overhead, averaged across all 16 SIMDRAM operations and four different element sizes (max. 1.52% for 8-bit reduction, min. 0.001% for 64-bit multiplication). Second, inter-bank data movement (Figure 2.13, right) results in 17.5% latency overhead, averaged across all 16 SIMDRAM operations and four different element sizes (max. 68.7% for 8-bit reduction, min. 0.03% for 64-bit multiplication). We observe that the latency overhead of moving data, as a fraction of the total computation latency *decreases* with element size, because the computation latency of each SIMDRAM operation increases with element size. We conclude that while efficient data movement is a challenge in processing-in-memory architectures that rely on moving and aligning operands, the performance overhead of data movement in SIMDRAM stays within an acceptable range even under worst-case assumptions.

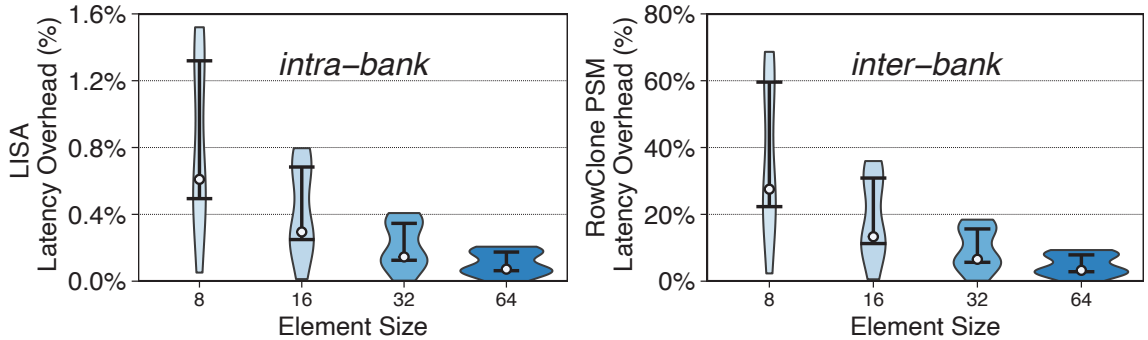


Figure 2.13: Latency overhead distribution of worst-case intra-bank (left) and inter-bank (right) data movement for *SIMDRAM:1*. Error bars depict the 25th and 75th percentiles.

2.6.7 Data Transposition Overhead

Transposition of the data in one subarray can overlap with in-DRAM computation in another subarray. As a result, if the data required for in-DRAM computation spans over multiple subarrays, only the transposition of the data in the first subarray is on the critical path of SIMDRAM execution. The data in each remaining subarray is then transposed simultaneously with the in-DRAM computation in the previous subarray.

To better understand the overhead of transposing data, we evaluate the worst-case latency of data transposition, which is when SIMDRAM's data initially resides in the cache in a horizontal layout. Before the computation of the SIMDRAM operation can start, this data needs to be transposed to a vertical layout and transferred to DRAM, incurring additional latency. Figure 2.14 shows this worst-case data transposition latency and the distribution of latency overhead of data transposition in *SIMDRAM:1* across all 16 SIMDRAM operations, as a function of element size. We make three observations. First, in *SIMDRAM:1* (*SIMDRAM:16*), data transposition incurs 7.1% (44.6%) latency overhead across all SIMDRAM operations (min. 0.03% (0.55%) for 64-bit multiplication, max. 38.9% (91.1%) for 8-bit AND-reduction and OR-reduction). As shown in Section 2.6.1, for all the evaluated element sizes, *SIMDRAM:1* (*SIMDRAM:16*) outperforms the CPU and GPU baselines by 5.5× and 0.4× (88.0× and 5.8×) on average across all 16 SIMDRAM operations, respectively. Even when we include the data transposition overhead, *SIMDRAM:1* (*SIMDRAM:16*) still outperforms both the CPU and GPU baselines by 4.0× and 0.24× (20.0× and 1.4×) on average across all 16 SIMDRAM operations. Our analysis for kernels that represent the behavior of real-world applications (Section 2.6.3) *already includes* the data transposition overhead. Second, the data transposition latency significantly increases with element size (by 9.7× from 8-bit elements to 64-bit elements). The number of cache lines that need to be transposed increases linearly with element size, which, in turn, increases the total transposition latency. Third, even though the transposition latency increases with element size, the transposition overhead as a fraction of the total latency *decreases* with element size, because the latency of each SIMDRAM operation also increases with element size. Since the transposition of data in each subarray is overlapped with the computation in another subarray, the increase in transposition latency is amortized over an even higher increase in the SIMDRAM operation latency. We conclude that SIMDRAM can efficiently perform in-DRAM computation even when worst-case data transposition overhead is taken into account.

2.6.8 Area Overhead

We use CACTI [265] to evaluate the area overhead of the primary components in the SIMDRAM design using a 22 nm technology node. SIMDRAM does not introduce any modifications to DRAM circuitry other than those proposed by Ambit, which has an area overhead of <1% in a commodity DRAM chip [338]. Therefore, SIMDRAM's area overhead

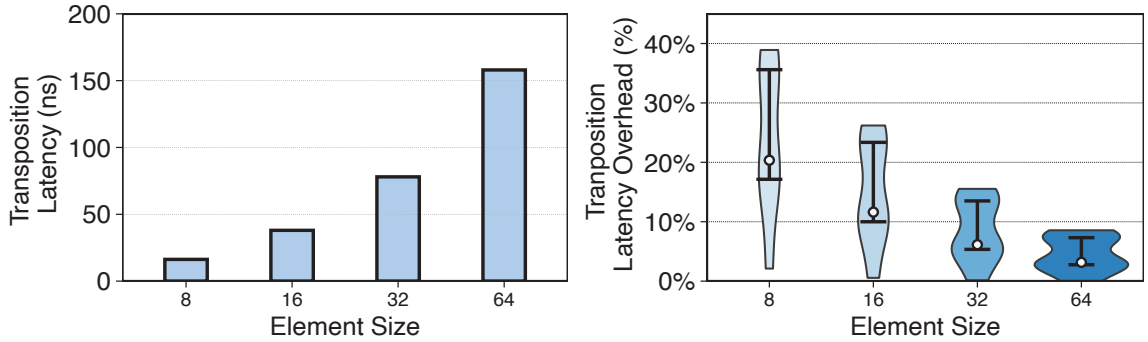


Figure 2.14: Worst-case latency (left) and worst-case latency overhead distribution (right) of data transposition in 16 SIMDRAM operations for *SIMDRAM:1*. Error bars depict the 25th and 75th percentiles, and a bubble depicts the 50th percentile.

over Ambit is only two structures in the memory controller: the control and transposition units.

Control Unit Area Overhead. The main components in the SIMDRAM control unit are the (1) *bbop* FIFO, (2) µProgram Scratchpad, (3) µOp Memory. We size the *bbop* FIFO and µProgram Scratchpad to 2 kB each. The size of the *bbop* FIFO is enough to hold up to 1024 *bbop* instructions, which we observe is more than enough for our real-world applications. The size of the µProgram Scratchpad is large enough to store the µPrograms for all 16 SIMDRAM operations that we evaluate in this work (16 µPrograms $\times$ 128 B max per µProgram). We use a 128 B scratchpad for the µOp Memory.[2] We estimate that the SIMDRAM control unit area is 0.04 mm^2.

Transposition Unit Area Overhead. The primary components in the transposition unit are (1) the Object Tracker and (2) two transposition buffers. We use an 8 kB fully-associative cache with a 64-bit cache line size for the Object Tracker. This is enough to store 1024 entries in the Object Tracker, where each entry holds the base physical address of a SIMDRAM object (19 bits), the total size of the allocated data (32 bits), and the size of each element in the object (6 bits). Each transposition buffer is 4 kB, to transpose up to a 64-bit SIMDRAM object (64-bit $\times$ 64 B). We estimate the transposition unit area is 0.06 mm^2. Considering the area of the control and transposition units, SIMDRAM has an area overhead of only 0.2% compared to the die area of an Intel Xeon E5-2697 v3 CPU [103]. We conclude that SIMDRAM has low area cost.

2.7 Related Work

To our knowledge, SIMDRAM is the first end-to-end framework that supports in-DRAM computation flexibly and transparently to the user. We highlight SIMDRAM's key contributions by contrasting it with state-of-the-art processing-in-memory designs.

Processing-near-Memory (PnM) within 3D-Stacked Memories. Many recent works (e.g., [7, 8, 14, 50–55, 80, 83, 92, 93, 109, 110, 116, 117, 125, 127, 147, 148, 184, 190, 223, 243, 278, 287, 288, 309, 326, 327, 331, 355, 397]) explore adding logic directly to the logic layer of 3D-stacked memories (e.g., High-Bandwidth Memory [166, 215], Hybrid Memory Cube [152]). The

implementation of SIMDRAM is considerably simpler, and relies on minimal modifications to commodity DRAM chips.

Processing-using-Memory (PuM). Prior works propose mechanisms wherein the memory arrays themselves perform various operations [4, 11, 19–22, 66, 71, 77, 82, 85, 94, 103, 108, 141, 157, 172, 188, 197, 227–229, 286, 333, 335, 336, 338, 340–343, 345, 347, 359, 360, 364, 380, 389]. SIMDRAM supports a much wider range of operations (compared to [11, 19, 77, 227, 228, 336, 338, 389]), at lower computational cost (compared to [338, 389]), at lower area overhead (compared to [228]), and with more reliable execution (compared to [11, 19]).

Processing-in-Cache. Recent works [4, 85, 103] propose in-SRAM accelerators that take advantage of the SRAM bitline structures to perform bit-serial computation in caches. SIMDRAM shares similarities with these approaches, but offers a significantly lower cost per bit by exploiting the high density and low cost of DRAM technology. We show the large performance and energy advantages of SIMDRAM compared to DualityCache [103] in Section 2.6.4.

Frameworks for PIM. Few prior works tackle the challenge of providing end-to-end support for PIM. We describe these frameworks and their limitations for in-DRAM computing. DualityCache [103] is an end-to-end framework for in-cache computing. DualityCache utilizes the CUDA/OpenAcc programming languages [70, 290] to generate code for an in-cache mechanism that executes a fixed set of operations in a single-instruction multiple-thread (SIMT) manner. Like SIMDRAM, DualityCache stores data in a vertical layout through the bitlines of the SRAM array. It treats each bitline as an independent execution thread and utilizes a crossbar network to allow inter-thread communication across bitlines. Despite its benefits, employing DualityCache in DRAM is not straightforward for two reasons. First, extending the DRAM subarray with the crossbar network utilized by DualityCache in SRAM to allow inter-thread communication would impose a prohibitive area overhead in DRAM (9× the DRAM subarray area). Second, as an in-cache computing solution, DualityCache does *not* account for the limitations of in-DRAM computing, i.e., DRAM operations that destroy input data, limited number of DRAM rows that are capable of processing-using-DRAM, and the need to avoid costly in-DRAM copies. We have already shown that SIMDRAM achieves higher performance at lower area overhead than DualityCache, when DRAM-to-cache data movement is realistically taken into account (Section 2.6.4).

Two prior works propose frameworks targeting ReRAM devices. Hyper-AP [396] is a framework for associative processing using ReRAM. Since Hyper-AP targets associative processing, the proposed framework is *fundamentally* different from SIMDRAM. IMP [102] is a framework for in-situ ReRAM operations. Like DualityCache, the IMP framework depends on particular structures of the ReRAM array (such as analog-to-digital/digital-to-analog converters) to perform computation and, thus, is not applicable to an in-DRAM substrate that performs bulk bitwise operations. Moreover, DualityCache, Hyper-AP, and IMP each have a rigid ISA that enables only a limited set of in-memory operations (DualityCache supports 16

in-memory operations, while both Hyper-AP and IMP support 12). In contrast, SIMDRAM is the first framework for PuM that is flexible, providing a methodology that allows new operations to be integrated and computed in memory as needed. In summary, SIMDRAM fills the gap for a flexible end-to-end framework that targets processing-using-DRAM.

2.8 Summary and Contributions

We introduce SIMDRAM, a massively-parallel general-purpose processing-using-DRAM framework that (1) enables the efficient implementation of a wide variety of operations in DRAM, in SIMD fashion, and (2) provides a flexible mechanism to support the implementation of arbitrary user-defined operations. SIMDRAM introduces a new three-step framework to enable efficient MAJ/NOT-based in-DRAM implementation for complex operations of different categories (e.g., arithmetic, relational, predication), and is applicable to a wide range of real-world applications. We design the hardware and ISA support for SIMDRAM framework to (1) address key system integration challenges, and (2) allow programmers to employ new SIMDRAM operations without hardware changes. We experimentally demonstrate that SIMDRAM provides significant performance and energy benefits over state-of-the-art CPU, GPU, and PuM systems. We hope that future work builds on our framework to further ease the adoption and improve the performance and efficiency of processing-using-DRAM architectures and applications.

In this chapter, we make the following key contributions:

- We propose the first framework, called SIMDRAM, to enable efficient computation of a flexible set and wide range of operations in a massively parallel SIMD substrate built via processing-using-DRAM.
- We demonstrate that SIMDRAM is a three-step framework to develop efficient and reliable MAJ/NOT-based implementations of a wide range of operations. We design this framework, and add hardware, programming, and ISA support, to (1) address key system integration challenges and (2) allow programmers to define and employ new SIMDRAM operations without hardware changes.
- We provide a detailed reference implementation of SIMDRAM, including required changes to applications, ISA, and hardware.
- We evaluate the reliability of SIMDRAM under different degrees of process variation and observe that it guarantees correct operation as the DRAM technology scales to smaller node sizes.

Chapter 3

The Virtual Block Interface

Virtual memory [38, 78, 79, 98, 182, 183] was originally designed for systems whose memory hierarchy fit a simple two-level model of small-but-fast main memory that can be directly accessed via CPU instructions and large-but-slow external storage accessed with the help of the operating system (OS). In such a configuration, the OS can easily abstract away the underlying memory architecture details and present applications with a unified view of memory.

However, continuing to efficiently support the conventional virtual memory framework requires significant effort due to (1) high memory demand and diverse memory requirements of modern applications, (2) emerging memory technologies (e.g., DRAM–NVM hybrid memories), and (3) diverse system architectures. The OS must now efficiently meet the wide range of application memory requirements that leverage the advantages offered by emerging memory architectures and new system designs while simultaneously hiding the complexity of the underlying memory and system architecture from the applications. Unfortunately, this is a difficult problem to tackle in a generalized manner. We describe three examples of challenges that arise when adapting conventional virtual memory frameworks to today's diverse system configurations.

Virtualized Environments. In a virtual machine, the guest OS performs virtual memory management on the emulated "physical memory" while the host OS performs a second round of memory management to map the emulated physical memory to the actual physical memory. This extra level of indirection results in three problems: (1) two-dimensional page walks [42, 105, 106, 256, 306, 323], where the number of memory accesses required to serve a TLB miss increases dramatically (e.g., up to 24 accesses in x86-64 with 4-level page tables); (2) performance loss in case of miscoordination between the guest and host OS mapping and allocation mechanisms (e.g., when the guest supports superpages, but the host does not); and (3) inefficiency in virtualizing increasingly complex physical memory architectures (e.g., hybrid memory systems) for the guest OS. These problems worsen with more page table levels [160], and in systems that support nested virtualization (i.e., a virtual machine running inside another) [91, 121].

Address Translation. In existing virtual memory frameworks, the OS manages virtual-to-physical address mapping. However, the hardware must be able to traverse these mappings to handle memory access operations (e.g., TLB lookups). This arrangement requires using *rigid* address-translation structures that are shared between and understood by both the hardware and the OS. Prior works show that many applications can benefit from flexible page tables, which cater to the application's actual memory footprint and access patterns [10, 34, 88, 169]. Unfortunately, enabling such flexibility in conventional virtual memory frameworks requires more complex address translation structures *every time* a new address translation approach is proposed. For example, a recent work [34] proposes using direct segments to accelerate big-memory applications. However, in order to support direct segments, the virtual memory contract needs to change to enable the OS to specify which regions of memory are directly mapped to physical memory. Despite the potential performance benefits, this approach is not easily scalable to today's increasingly diverse system architectures.

Memory Heterogeneity. Prior works propose many performance-enhancing techniques that require (1) dynamically *mapping* data to different physical memory regions according to application requirements (e.g., mapping frequently-accessed data to fast memory), and (2) *migrating* data when those requirements change (e.g., [64, 66, 81, 167, 186, 216, 218, 232, 246, 248, 311, 315, 317, 318, 358, 392, 400]). Efficiently implementing such functionality faces two challenges. First, a customized data mapping requires the OS to be aware of microarchitectural properties of the underlying memory. Second, even if this can be achieved, the OS has low visibility into rich fine-grained runtime memory behavior information (e.g., access pattern, memory bandwidth availability), especially at the main memory level. While hardware has access to such fine-grained information, informing the OS *frequently enough* such that it can react to changes in the memory behavior of an application in a *timely* manner is challenging [257, 317, 351, 374, 394].

A wide body of research (e.g., [1–3, 10, 17, 27, 28, 30, 32–34, 38, 42, 43, 45–47, 49, 59, 60, 69, 72, 78, 79, 84, 105–107, 111, 126, 128, 129, 134, 146, 149, 171, 174–176, 182, 200, 204–207, 230, 231, 251, 253, 254, 257, 259, 297, 304–306, 308, 309, 312, 313, 321–323, 329, 339, 344, 349, 351, 356, 367–369, 383–386, 391, 401]) proposes mechanisms to alleviate the overheads of conventional memory allocation and address translation by exploiting specific trends observed in modern systems (e.g., the behavior of emerging applications). Despite notable improvements, these solutions have two major shortcomings. First, these solutions mainly exploit specific system or workload characteristics and, thus, are applicable to a limited set of problems or applications. Second, each solution requires specialized and not necessarily compatible changes to both the OS and hardware. Therefore, implementing all of these proposals at the same time in a system is a daunting prospect.

Our goal in this work is to design *a general-purpose alternative virtual memory framework that naturally supports and better extracts performance from a wide variety of new system configurations, while still providing the key features of conventional virtual memory*

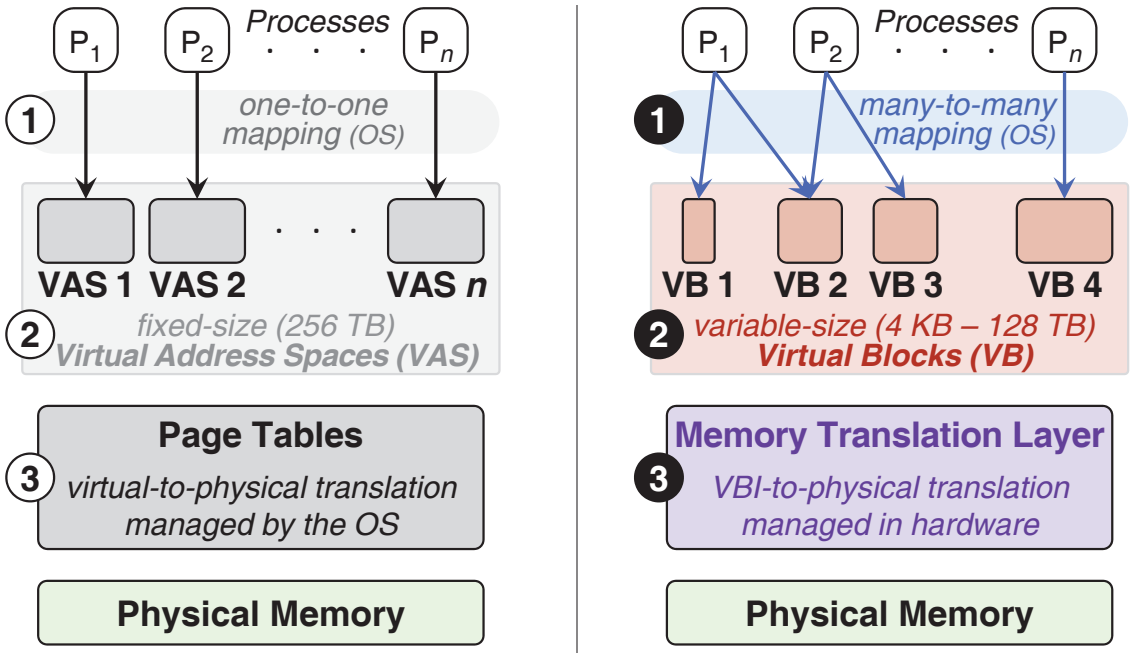
frameworks. To this end, we propose the Virtual Block Interface (VBI), an alternative approach to memory virtualization that is inspired by the logical block abstraction used by solid-state drives to hide the underlying device details from the rest of the system. In a similar way, we envision the memory controller as the primary provider of an abstract interface that hides the details of the underlying physical memory architecture, including the physical addresses of the memory locations.

VBI is based on three guiding principles. First, *programs should be allowed to choose the* size *of their virtual address space*, to mitigate translation overheads associated with very large virtual address spaces. Second, *address translation should be decoupled from memory protection*, since they are logically separate and need not be managed at the same granularity by the same structures. Third, *software should be allowed to communicate semantic information about application data to the hardware*, so that the hardware can more intelligently manage the underlying hardware resources.

VBI introduces a *globally-visible* address space called the *VBI Address Space*, that consists of a large set of *virtual blocks (VBs)* of different sizes. For any semantically meaningful unit of information (e.g., a data structure, a shared library), the program can choose a VB of appropriate size, and tag the VB with properties that describe the contents of the VB. **The key idea** of VBI is to delegate physical memory allocation and address translation to a hardware-based Memory Translation Layer (MTL) at the memory controller. This idea is enabled by the fact that the globally-visible VBI address space provides VBI with system-wide unique *VBI addresses* that can be *directly* used by on-chip caches without requiring address translation. In VBI, the OS no longer needs to manage address translation and memory allocation for the physical memory devices. Instead, the OS (1) retains full control over access protection by controlling which programs have access to which virtual blocks, and (2) uses VB properties to communicate the data's memory requirements (e.g., latency sensitivity) and characteristics (e.g., access pattern) to the memory controller.

Figure 3.1 illustrates the differences between virtual memory management in state-of-the-art production Intel x86-64 systems and in VBI. In x86-64 (Figure 3.1a), the OS manages a single private virtual address space (VAS) for each process (①), providing each process with a fixed-size 256 TB VAS irrespective of the actual memory requirements of the process (②). The OS uses a set of page tables, one per process, to define how each VAS maps to physical memory (③). In contrast, VBI (Figure 3.1b) makes *all* virtual blocks (VBs) visible to *all* processes, and the OS controls which processes can access which VBs (❶). Therefore, a process' total virtual address space is defined by which VBs are attached to it, i.e., by the process' actual memory needs (❷). In VBI, the MTL has full control over mapping of data from each VB to physical memory, invisibly to the system software (❸).

VBI seamlessly and efficiently supports important optimizations that improve overall system performance, including: (1) enabling benefits akin to using virtually-indexed virtually-tagged (VIVT) caches (e.g., reduced address translation overhead), (2) eliminating two-



(a) Intel x86-64 (b) Virtual Block Interface

Figure 3.1: Virtual memory management in x86-64 and in VBI.

dimensional page table walks in virtual machine environments, (3) delaying physical memory allocation until the first dirty last-level cache line eviction, and (4) flexibly supporting different virtual-to-physical address translation structures for different memory regions. Section 3.2.5 describes these optimizations in detail.

We evaluate VBI for two important and emerging use-cases. First, we demonstrate that VBI significantly reduces the address translation overhead both for *natively-running programs* and for programs running inside a virtual machine (*VM programs*). Quantitative evaluations using workloads from SPEC CPU 2006 [362], SPEC CPU 2017 [363], TailBench [135], and Graph 500 [122] show that a simplified version of VBI that maps VBs using 4 KB granularity only improves the performance of native programs by 2.18× and VM programs by 3.8×. Even when enabling support for large pages for *all data*, which significantly lowers translation overheads, VBI improves performance by 77% for native programs and 89% for VM programs. Second, we demonstrate that VBI significantly improves the performance of heterogeneous memory architectures by evaluating two heterogeneous memory systems (PCM–DRAM [317] and Tiered-Latency-DRAM [218]). We show that VBI, by intelligently mapping frequently-accessed data to the low-latency region of memory, improves overall performance of these two systems by 33% and 21% respectively, compared to systems that employ a heterogeneity-unaware data mapping scheme. Section 3.6 describes our methodology, results, and insights from these evaluations.

3.1 Design Principles

To minimize performance and complexity overheads of memory virtualization, our virtual memory framework is grounded on three key design principles.

Appropriately-Sized Virtual Address Spaces. The virtual memory framework should *allow each application to have control over the size of its virtual address space*. The majority of applications far underutilize the large virtual address space offered by modern architectures (e.g., 256 TB in Intel x86-64). Even demanding applications such as databases [75, 90, 264, 281, 291, 328] and caching servers [96, 283] are cognizant of the amount of available physical memory and of the size of virtual memory they need. Unfortunately, a larger virtual address space results in larger or deeper page tables (i.e., page tables with more levels). A larger page table increases TLB contention, while a deeper page table requires a greater number of page table accesses to retrieve the physical address for each TLB miss. In both cases, the address translation overhead increases. Therefore, allowing applications to choose an appropriately-sized virtual address space based on their actual needs, avoids the higher translation overheads associated with a larger address space.

Decoupling Address Translation from Access Protection. The virtual memory framework should *decouple address translation from access protection checks*, as the two have inherently different characteristics. While address translation is typically performed at page granularity, protection information is typically the same for an entire data structure, which can span multiple pages. Moreover, protection information is purely a function of the virtual address, and does not require address translation. However, existing systems store both translation and protection information for each virtual page as part of the page table. Decoupling address translation from protection checking can enable opportunities to remove address translation from the critical path of an access protection check, deferring the translation until physical memory *must* be accessed, thereby lowering the performance overheads of virtual memory.

Better Partitioning of Duties Between Software and Hardware. The virtual memory framework should *allow software to easily communicate semantic information about application data to hardware and allow hardware to manage the physical memory resources*. Different pieces of program data have different performance characteristics (latency, bandwidth, and parallelism), and have other inherent properties (e.g., compressibility, persistence) at the software level. As highlighted by recent work [375, 377], while software is aware of this semantic information, the hardware is privy to fine-grained dynamic runtime information (e.g., memory access behavior, phase changes, memory bandwidth availability) that can enable vastly more intelligent management of the underlying hardware resources (e.g., better data mapping, migration, and scheduling decisions). Therefore, conveying semantic information to the hardware (i.e., memory controller) that manages the physical memory resources can enable a host of new optimization opportunities.

3.2 Virtual Block Interface: Overview

Figure 3.2 shows an overview of VBI. There are three major aspects of the VBI design: (1) the VBI address space, (2) VBI access permissions, and (3) the Memory Translation

Layer. We first describe these aspects in detail (Section 3.2.1–Section 3.2.3). Next, we explain the implementation of key OS functionalities in VBI (Section 3.2.4). Finally, we discuss some of the key optimizations that VBI enables (Section 3.2.5).

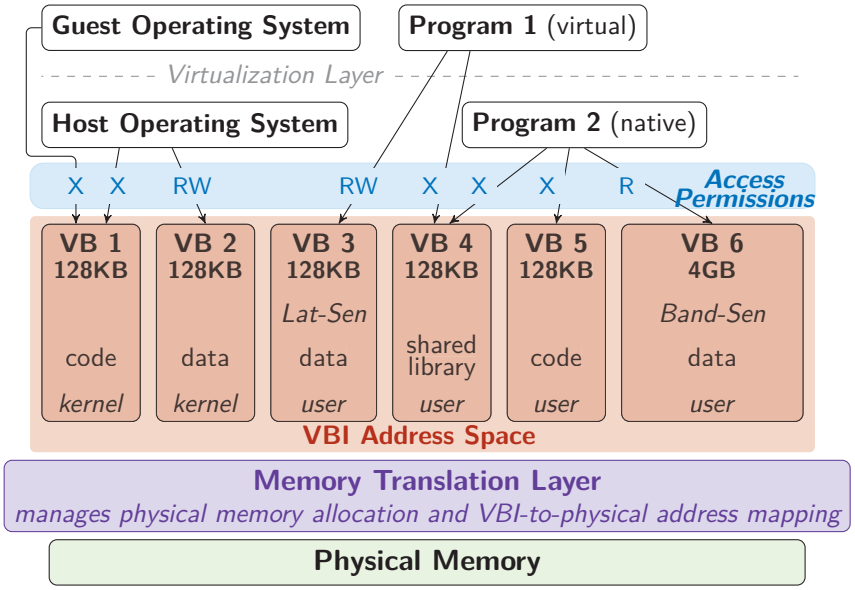


Figure 3.2: Overview of VBI. *Lat-Sen* and *Band-Sen* represent latency-sensitive and bandwidth-sensitive, respectively.

3.2.1 VBI Address Space

Unlike most existing architectures wherein each process has its own virtual address space, virtual memory in VBI is a single, globally-visible address space called the *VBI Address Space*. As shown in Figure 3.2, the VBI Address Space consists of a finite set of *Virtual Blocks* (VBs). Each VB is a contiguous region of VBI address space that does not overlap with any other VB. Each VB contains a semantically meaningful unit of information (e.g., a data structure, a shared library) and is associated with (1) a system-wide unique ID, (2) a specific size (chosen from a set of pre-determined size classes), and (3) a set of properties that specify the semantics of the content of the VB and its desired characteristics. For example, in the figure, `VB 1` indicates the VB with ID 1; its size is 128 KB, and it contains code that is accessible only to the kernel. On the other hand, `VB 6` is the VB with ID 6; its size is 4 GB, and it contains data that is bandwidth-sensitive. In contrast to conventional systems, where the mapping from the process' virtual-to-physical address space is stored in a per-process page table [161], VBI maintains the VBI-to-physical address mapping information of each VB in a *separate* translation structure. This approach enables VBI to flexibly tune the type of translation structure for each VB to the characteristics of the VB (as described in Section 3.4.2). VBI stores the above information and a pointer to the translation structure of each VB in a set of *VB Info Tables* (VITs; described in Section 3.3.5).

3.2.2 VBI Access Permissions

As the VBI Address Space is global, all VBs in the system are *visible* to all processes. However, a program can *access* data within a VB *only if* it is attached to the VB with appropriate permissions. In Figure 3.2, `Program 2` can only execute from `VB 4` or `VB 5`, only read from `VB 6`, and cannot access `VB 3` at all; `Program 1` and `Program 2` both share `VB 4`. For each process, VBI maintains information about the set of VBs attached to the process in an OS-managed per-process table called the *Client–VB Table* (CVT) (described in Section 3.3.1). VBI provides the OS with a set of instructions with which the OS can control which processes have what type of access permissions to which VBs. On each memory access, the processor checks the CVT to ensure that the program has the necessary permission to perform the access. With this approach, VBI *decouples protection checks from address translation*, which allows it to defer the address translation to the memory controller where the physical address is required to access main memory.

3.2.3 Memory Translation Layer

In VBI, to access a piece of data, a program must specify the ID of the VB that contains the data and the offset of the data within the VB. Since the ID of the VB is unique system-wide, the combination of the ID and offset points to the address of a specific byte of data in the VBI address space. We call this address the *VBI address*. As the VBI address space is globally visible, similar to the physical address in existing architectures, the VBI address points to a unique piece of data in the system. As a result, VBI uses the VBI address *directly* (i.e., without requiring address translation) to locate data within the on-chip caches without worrying about the complexity of homonyms and synonyms [57, 58, 164], which cannot exist in VBI (see Section 3.2.5). Address translation is required only when an access misses in all levels of on-chip caches.

To perform address translation, VBI uses the Memory Translation Layer (MTL). The MTL, implemented in the memory controller with an interface to the system software, manages both allocation of physical memory to VBs and VBI-to-physical address translation (relieving the OS of these duties). Memory-controller-based memory management enables a number of performance optimizations (e.g., avoiding 2D page walks in virtual machines, flexible address translation structures), which we describe in Section 3.2.5.

3.2.4 Implementing Key OS Functionalities

VBI allows the system to efficiently implement existing OS functionalities. In this section, we describe five key functionalities and how VBI enables them.

Physical Memory Capacity Management. In VBI, the MTL allocates physical memory for VBs as and when required. To handle situations when the MTL runs out of physical memory, VBI provides two system calls that allow the MTL to move data from physical memory to the backing store and vice versa. The MTL maintains information about swapped-out data as part of the VB's translation structures.

Data Protection. The goal of data protection is to prevent a malicious program from accessing kernel data or private data of other programs. In VBI, the OS ensures such protection by appropriately setting the permissions with which each process can access different VBs. Before each memory access, the CPU checks if the executing thread has appropriate access permissions to the corresponding VB (Section 3.3.2).

Inter-Process Data Sharing (True Sharing). When two processes share data (e.g., via pipes), both processes have a coherent view of the shared memory, i.e., modifications made by one process should be visible to the other process. In VBI, the OS supports such *true* sharing by granting both processes permission to access the VB containing the shared data.

Data Deduplication (Copy-on-Write Sharing). In most modern systems, the OS reduces redundancy in physical memory by mapping virtual pages containing the *same* data to the same physical page. On a write to one of the virtual pages, the OS copies the data to a new physical page, and remaps the written virtual page to the new physical page before performing the write. In VBI, the MTL performs data deduplication when a VB is cloned by sharing both translation structures and data pages between the two VBs (Section 3.3.4), and using the copy-on-write mechanism to ensure consistency.

Memory-Mapped Files. To support memory-mapped files, existing systems map a region of the virtual address space to a file in storage, and loads/stores to that region are used to access/update the file content. VBI naturally supports memory-mapped files as the OS simply associates the file to a VB of appropriate size. An offset within the VB maps to the same offset within the file. The MTL uses the same system calls used to manage physical memory capacity (described under *Physical Memory Capacity Management* above) to move data between the VB in memory and the file in storage.

3.2.5 Optimizations Supported by VBI

In this section, we describe four key optimizations that the VBI design enables.

Virtually-Indexed Virtually-Tagged Caches. Using fully-virtual (i.e., VIVT) caches enables the system to delay address translation and reduce accesses to translation structures such as the TLBs. However, most modern architectures do not support VIVT caches due to two main reasons. First, handling homonyms (i.e., where the same virtual address maps to multiple physical addresses) and synonyms (i.e., where multiple virtual addresses map to the same physical address) introduces complexity to the system [57, 58, 164]. Second, although address translation is not required to access VIVT caches, the access permission check required prior to the cache access still necessitates accessing the TLB and can induce a page table walk on a TLB miss. This is due to the fact that the protection bits are stored as part of the page table entry for each page in current systems. VBI avoids both of these problems.

First, VBI addresses are unique system-wide, eliminating the possibility of homonyms. Furthermore, since VBs do not overlap, each VBI address appears in *at most one* VB, avoiding the possibility of synonyms. In case of true sharing (Section 3.2.4), different processes are attached to the same VB. Therefore, the VBI address that each process uses to access the shared region refers to the *same* VB. In case of copy-on-write sharing, where the MTL may map two VBI addresses to the same physical memory for deduplication, the MTL creates a new copy of the data before any write to either address. Thus, neither form of sharing can lead to synonyms. As a result, by using VBI addresses directly to access on-chip caches, VBI achieves benefits akin to VIVT caches without the complexity of dealing with synonyms and homonyms. Additionally, since the VBI address acts as a system-wide single point of reference for the data that it refers to, all coherence-related requests can use VBI addresses without introducing any ambiguity.

Second, VBI decouples protection checks from address translation, by storing protection and address translation information in *separate* sets of tables and delegating access permission management to the OS, avoiding the need to access translation structures for protection purposes (as done in existing systems).

Avoiding 2D Page Walks in Virtual Machines. In VBI, once a process inside a VM attaches itself to a VB (with the help of the host and guest OSes), any memory access from the VM directly uses a VBI address. As described in Section 3.2.3, this address is directly used to address the on-chip caches. In case of an LLC miss, the MTL translates the VBI address to physical address. As a result, unlike existing systems, address translation for a VM under VBI is no different from that for a host, enabling significant performance improvements. We expect these benefits to further increase in systems supporting nested virtualization [91, 121]. Section 3.5.1 discusses the implementation of VBI in virtualized environments.

Delayed Physical Memory Allocation. As VBI uses VBI addresses to access all on-chip caches, it is no longer necessary for a cache line to be backed by physical memory *before* it can be accessed. This enables the opportunity to delay physical memory allocation for a VB (or a region of a VB) until a dirty cache line from the VB is evicted from the last-level cache. Delayed allocation has three benefits. First, the allocation process is removed from the critical path of execution, as cache line evictions are not on the critical path. Second, for VBs that never leave the cache during the lifetime of the VB (likely more common with growing cache sizes in modern hardware), VBI avoids physical memory allocation altogether. Third, when using delayed physical memory allocation, for an access to a region with no physical memory allocated yet, VBI simply returns a zero cache line, thereby avoiding *both* address translation and a main memory access, which improves performance. Section 3.4.1 describes the implementation of delayed physical memory allocation in VBI.

Flexible Address Translation Structures. A recent work [10] shows that different data structures benefit from different types of address translation structures depending on their data layout and access patterns. However, since in conventional virtual memory, the hardware needs to read the OS-managed page tables to perform page table walks, the structure of the page table needs to be understood by both the hardware and OS, thereby limiting the flexibility of the page table structure. In contrast, in VBI, the MTL is the *only* component that manages and accesses translation structures. Therefore, the constraint of sharing address translation structures with the OS is relaxed, providing VBI with more flexibility in employing different types of translation structures in the MTL. Accordingly, VBI maintains a separate translation structure for each VB, and can tune it to suit the properties of the VB (e.g., multi-level tables for large VBs or those with many sparsely-allocated regions, and single-level tables for small VBs or those with many large contiguously-allocated regions). This optimization reduces the number of memory accesses necessary to serve a TLB miss.

3.3 VBI: Detailed Design

In this section, we present the detailed design and a reference implementation of the Virtual Block Interface. We describe (1) the components architecturally exposed by VBI to the rest of the system (Section 3.3.1), (2) the life-cycle of allocated memory (Section 3.3.2), (3) the interactions between the processor, OS, and the process in VBI (Section 3.3.4), and (4) the operation of the Memory Translation Layer in detail (Section 3.3.5).

3.3.1 Architectural Components

VBI exposes two architectural components to the rest of the system that form the contract between hardware and software: (1) virtual blocks, and (2) *memory clients*.

Virtual Blocks (VBs)

The VBI address space in VBI is characterized by three parameters: (1) the size of the address space, which is determined by the bit width of the processor's address bus (64 in our implementation); (2) the number of VB size classes (8 in our implementation); and (3) the list of size classes (4 KB, 128 KB, 4 MB, 128 MB, 4 GB, 128 GB, 4 TB, and 128 TB). Each size class in VBI is associated with an ID (`SizeID`), and each VB is assigned an ID *within its size class* (`VBID`). Every VB is identified system-wide by its *VBI unique ID* (`VBUID`), which is the concatenation of `SizeID` and `VBID`. As shown in Figure 3.3, VBI constructs a *VBI address* using two components: (1) `VBUID`, and (2) the offset of the addressed data within the VB. In our implementation, `SizeID` uses three bits to represent each of our eight possible size classes. The remaining address bits are split between `VBID` and the offset. The precise number of bits required for the offset is determined by the size of the VB, and the remaining bits are used for `VBID`. For example, the 4 KB size class in our implementation uses 12 bits for the offset, leaving 49 bits for `VBID`, i.e., 2^{49} VBs of size 4 KB. In contrast, the 128 TB size class uses 47 bits for the offset, leaving 14 bits for `VBID`, i.e., 2^{14} VBs of size 128 TB.

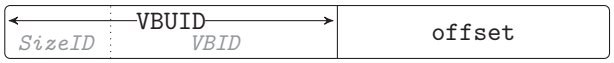


Figure 3.3: Components of a VBI address.

As Section 3.2 describes, VBI associates each VB with a set of flags that characterize the contents of the VB (e.g, `code`, `read-only`, `kernel`, `compressible`, `persistent`). In addition to these flags, software may also provide hints to describe the memory behavior of the data that the VB contains (e.g., latency sensitivity, bandwidth sensitivity, compressibility, error tolerance). Prior work extensively studies a set of useful properties [249,375,377,395]. Software specifies these properties via a bitvector that is defined as part of the ISA specification. VBI maintains the flags and the software-provided hints as a *property bitvector*.

For each VB in the system, VBI stores (1) an *enable* bit to describe whether the VB is currently assigned to any process, (2) the property bitvector, (3) the number of processes attached to the VB (i.e., a reference count), (4) the type of VBI-to-physical address translation structure being used for the VB, and (5) a pointer to the VB's address translation structure. All of this information is stored as an entry in the VB Info Tables (Section 3.3.5).

Memory Clients

Similar to address space identifiers [5] in existing architectures, VBI introduces the notion of *memory client* to communicate the concept of a process in VBI. A memory client refers to any entity that needs to allocate and use memory, such as the OS itself, and any process running on the system (natively or inside a virtual machine). In order to track the permissions with which a client can access different VBs, each client in VBI is assigned a unique ID to identify the client system-wide. During execution, VBI tags each core with the client ID of the process currently running on it.

As Section 3.2 discusses, the set of VBs that a client can access and their associated permissions are stored in a per-client table called the *Client–VB Table* (CVT). Each entry in the CVT contains (1) a valid bit, (2) `VBUID` of the VB, and (3) a three-bit field representing the read-write-execute permissions (RWX) with which the client can access that VB. For each memory access, the processor checks the CVT to ensure that the client has appropriate access to the VB. The OS implicitly manages the CVTs using the following two new instructions:

```
attach CID, VBUID, RWX
```

```
detach CID, VBUID
```

The `attach` instruction adds an entry for VB `VBUID` in the CVT of client `CID` with the specified `RWX` permissions (either by replacing an invalid entry in the CVT, or being inserted at the end of the CVT). This instruction returns the index of the CVT entry to the OS and increments the reference count of the VB (stored in the VIT entry of the VB; see Section 3.3.5). The `detach` instruction resets the valid bit of the entry corresponding to VB `VBUID` in the CVT of client `CID` and decrements the reference count of the VB.

The processor maintains the location and size of the CVT for each client in a reserved region of physical memory. As clients are visible to both the hardware and the software, the

number of clients is an architectural parameter determined at design time and exposed to the OS. In our implementation, we use 16-bit client IDs (supporting 2^{16} clients).

3.3.2 Life Cycle of Allocated Memory

In this section, we describe the phases in the life cycle of dynamically-allocated memory: memory allocation, address specification, data access, and deallocation. Figure 3.4 shows this flow in detail, including the hardware components that aid VBI in efficiently executing memory operations. In Section 3.3.4, we discuss how VBI manages code, shared libraries, static data, and the life cycle of an entire process.

When a program needs to allocate memory for a new data structure, it first requests a new VB from the OS. For this purpose, we introduce a new system call, `request_vb`. The program invokes `request_vb` with two parameters: (1) the *expected* size of the data structure, and (2) a bitvector of the desired properties for the data structure (1a in Figure 3.4).

In response, the OS first scans the VB Info Table to identify the smallest free VB that can accommodate the data structure. The OS then uses the `enable_vb` instruction (1b) to inform the MTL that the VB is now enabled. The `enable_vb` instruction takes the `VBUID` of the VB to be enabled along with the properties bitvector as arguments. Upon executing this instruction, the MTL updates the entry for the VB in the VB Info Table to reflect that it is now enabled with the appropriate properties (1c).

```
enable_vb VBUID, props
```

Dynamic Memory Allocation

After enabling the VB, the OS uses the `attach` instruction (2a) to add the VB to the CVT of the calling process and increment the VB's reference count in its VIT entry (2b; Section 3.3.1). The OS then returns the index of the newly-added CVT entry as the return value of the `request_vb` system call (stored as `index` in the application code example of Figure 3.4). This `index` serves as a pointer to the VB. As we discuss in Section 3.3.2, the program uses this index to specify virtual addresses to the processor.

After the VB is attached to the process, the process can access any location within the VB with the appropriate permissions. It can also dynamically manage memory inside the VB using modified versions of `malloc` and `free` that take the CVT entry `index` as an additional argument (3). During execution, it is possible that the process runs out of memory within a VB (e.g., due to an incorrect estimate of the expected size of the data structure). In such a case, VBI allows automatic promotion of the allocated data to a VB of a larger size class. Section 3.3.4 discusses VB promotion in detail.

Address Specification

In order to access data inside a VB, the process generates a two-part virtual address in the format of {`CVT index, offset`}. The `CVT index` specifies the CVT entry that points to the corresponding VB, and the `offset` is the location of the data inside the VB. Accessing the data indirectly through the `CVT index` as opposed to direcly using the VBI address

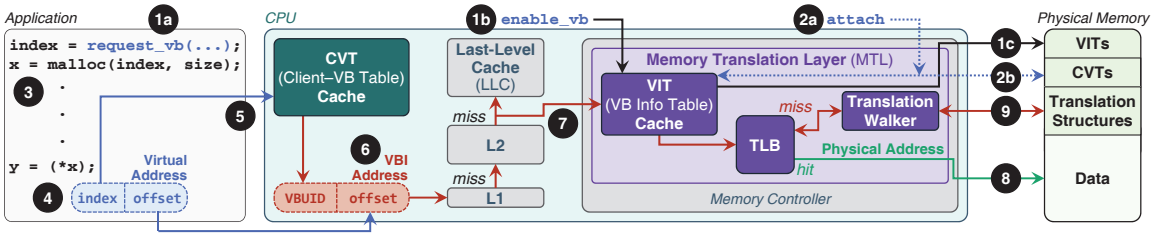


Figure 3.4: Reference microarchitectural implementation of the Virtual Block Interface.

allows VBI to not require relocatable code and maintain the validity of the pointers (i.e., virtual addresses) within a VB when migrating/copying the content of a VB to another VB. With CVT indirection, VBI can seamlessly migrate/copy VBs by just updating the `VBUID` of the corresponding CVT entry with the `VBUID` of the new VB.

Operation of a Memory Load

Figure 3.4 shows the execution of the memory load instruction triggered by the code `y = (*x)`, where the pointer `x` contains the virtual address consisting of (1) the index of the corresponding VB in the process' CVT, and (2) the offset within the VB (❹ in Figure 3.4). When performing a load operation, the CPU first checks whether `index` is within the range of the client's CVT. Next, the CPU needs to fetch the corresponding CVT entry in order to perform the permissions check. The CPU uses a per-process small direct-mapped CVT cache to speed up accesses to the client's recently-accessed CVT entries (Section 3.3.3). Therefore, the CPU looks up the corresponding CVT cache entry using `index` as the key (❺), and checks if (1) the client has permission to read from the VB, and (2) `offset` is smaller than the size of the VB. If either of these checks fail, the CPU raises an exception. If the access is allowed, the CPU constructs the VBI address by concatenating the `VBUID` stored in the CVT entry with `offset` (❻). The processor directly uses the generated VBI address to access the on-chip caches. If the data is present in any of the on-chip caches, it is returned to the CPU, thereby completing the load operation.

VBI performs address translation in parallel with the cache lookup in order to minimize the address translation overhead on the critical path of the access. Accordingly, when an access misses in the L2 cache, the processor requests the MTL to perform the VBI-to-physical address translation. To this end, MTL fetches the pointer to the VB's translation structure from the VBI Info Table (VIT) entry associated with the VB. VBI uses a VIT cache to speed up accesses to recently-accessed VIT entries (❼). In order to facilitate the VBI-to-physical address translation, MTL employs a translation lookaside buffer (TLB). On a TLB hit, the memory controller accesses the cache line using the physical address in the corresponding TLB entry (❽). On a TLB miss, the MTL performs the address translation by traversing the VB's translation structure (❾), and inserts the mapping information into the TLB once the physical address is obtained. Next, the memory controller fetches the corresponding cache line from main memory and returns it to the processor. The processor inserts the

cache line into the on-chip caches using the VBI address, and returns the cache line to the CPU to complete the load. Section 3.3.5 describes the operation of the MTL in detail.

Memory Deallocation

The program can deallocate the memory allocated inside a VB using `free` (Section 3.3.2). When a process terminates, the OS traverses the CVT of the process and detaches all of the VBs attached to the process using the `detach` instruction. For each VB whose reference count (stored as part of VIT entry of the VB; see Section 3.3.5) drops to zero, the OS informs VBI that the VB is no longer in use via the `disable_vb` instruction.

```
disable_vb VBUID
```

In response to the `disable_vb` instruction, the MTL destroys all state associated with VB `VBUID`. To avoid stale data in the cache, all of the VB's cache lines are invalidated *before* the `VBUID` is reused for another memory allocation. Because there are a large number of VBs in each size class, it is likely that the disabled `VBUID` does not need to be reused immediately, and the cache cleanup can be performed lazily in the background.

3.3.3 CVT Cache

For every memory operation, the CPU must check if the operation is permitted by accessing the information in the corresponding CVT entry. To exploit locality in the CVT, VBI uses a per-core *CVT cache* to store recently-accessed entries in the client's CVT. The CVT cache is similar to the TLB in existing processors. However, unlike a TLB that caches virtual-to-physical address mappings of page-sized memory regions, the CVT cache maintains information at the VB granularity, and only for VBs that can be accessed by the program. While programs may typically access hundreds or thousands of pages, our evaluations show that most programs only need a few *tens* of VBs to subsume all their data. With the exception of *GemsFDTD* (which allocates 195 VBs),[1] all applications use fewer than 48 VBs. Therefore, the processor can achieve a near-100% hit rate even with a 64-entry *direct-mapped* CVT cache, which is faster and more efficient than the large set-associative TLBs employed by modern processors.

3.3.4 Processor, OS, and Process Interactions

VBI handles basic process lifetime operations similar to current systems. This section describes in detail how these operations work with VBI.

System Booting. When the system is booted, the processor initializes the data structures relevant to VBI (e.g., pointers to VIT tables) with the help of the MTL (discussed in Section 3.3.5). An initial ROM program runs as a privileged client, copies the bootloader code from bootable storage to a newly enabled VB, and jumps to the bootloader's entry

[1] *GemsFDTD* performs computations in the time domain on 3D grids. It involves multiple execution timesteps, each of which allocates new 3D grids to store the computation output. Multiple allocations are also needed during the post-processing Fourier transformation performed in *GemsFDTD*.

point. This process initiates the usual sequence of chain loading until the OS is finally loaded into a VB. The OS reads the parameters of VBI, namely, the number of bits of virtual address, the number and sizes of the virtual block size classes, and the maximum number of memory clients supported by the system, to initialize the OS-level memory management subsystem.

Process Creation. When a binary is executed, the OS creates a new process by associating it with one of the available client IDs. For each section of the binary (e.g., code, static data), the OS (1) enables the smallest VB that can fit the contents of the section and associates the VB with the appropriate properties using the `enable_vb` instruction, (2) attaches itself to the VB with write permissions using the `attach` instruction, (3) copies the contents from the application binary into the VB, and (4) detaches itself from the VB using the `detach` instruction. The OS then attaches the client to the newly enabled VBs and jumps to program's entry point.

Shared Libraries. The OS loads the executable code of each shared library into a separate VB. While a shared library can dynamically allocate data using the `request_vb` system call, any static per-process data associated with the library should be loaded into a separate VB for each process that uses the library. In existing systems, access to static data is typically performed using PC-relative addressing. VBI provides an analogous memory addressing mode that we call *CVT-relative addressing*. In this addressing mode, the CVT index of a memory reference is specified relative to the CVT index of the VB containing the reference. Specifically, in shared libraries, all references to static data use +1 CVT-relative addressing, i.e., the CVT index of the data is one more than the CVT index of the code. After process creation, the OS iterates over the list of shared libraries requested by the process. For each shared library, the OS attaches the client to the VB containing the corresponding library code and ensures that the subsequent CVT entry is allocated to the VB containing the static data associated with the shared library. This solution avoids the need to perform load-time relocation for each data reference in the executable code, although VBI can use relocations in the same manner as current systems, if required.

Process Destruction. When a process terminates, the OS deallocates all VBs for the process using the mechanism described in Section 3.3.2, and then frees the client ID for reuse.

Process Forking. When a process forks, all of its memory state must be replicated for the newly created process. In VBI, forking entails creating copies of all the private VBs attached to a process. To reduce the overhead of this operation, VBI introduces the following instruction:

```
clone_vb SVBUID, DVBUID
```

`clone_vb` instructs VBI to make the destination VB `DVBUID` a clone of the source VB `SVBUID`. To efficiently implement `clone_vb`, the MTL marks all translation structures and physical pages of the VB as copy-on-write, and lazily copies the relevant regions if they receive a write operation.[2]

When forking a process, the OS first copies all CVT entries of the parent to the CVT of the child so that the child VBs have the same CVT index as the parent VBs. This maintains the validity of the pointers in the child VBs after cloning. Next, for each CVT entry corresponding to a private VB (shared VBs are already enabled), the OS (1) enables a new VB of the same size class and executes the `clone_vb` instruction, and (2) updates the `VBUID` in the CVT entry to point to the newly enabled clone. The fork returns after all the `clone_vb` operations are completed.

VB Promotion. As described in Section 3.3.2, when a program runs out of memory for a data structure within the assigned VB, the OS can automatically promote the data structure to a VB of higher size class. To perform such a promotion, the OS first suspends the program. It enables a new VB of the higher size class, and executes the `promote_vb` instruction.

```
promote_vb SVBUID, LVBUID
```

In response to this instruction, VBI first flushes all dirty cache lines from the smaller VB with the unique ID of `SVBUID`. This operation can be sped up using structures like the Dirty Block Index [334]. VBI then copies all the translation information from the smaller VB appropriately to the larger VB with the unique ID of `LVBUID`. After this operation, in effect, the early portion of the larger VB is mapped to the same region in the physical memory as the smaller VB. The remaining portions of the larger VB are unallocated and can be used by the program to expand its data structures and allocate more memory using `malloc`. VBI updates the entry in the program's CVT that points to `SVBUID` to now point to `LVBUID`.

3.3.5 Memory Translation Layer

The Memory Translation Layer (MTL) centers around the VB Info Tables (VITs), which store the metadata associated with each VB. In this section, we discuss (1) the design of the VITs, (2) the two main responsibilities of the MTL; memory allocation and address translation, and (3) the hardware complexity of the MTL.

VB Info Table (VIT)

As Section 3.3.1 briefly describes, MTL uses a set of VB Info Tables (VITs) to maintain information about VBs. Specifically, for each VB in the system, a VB Info Table stores an entry that consists of (1) an *enable* bit, which indicates if the VB is currently assigned to a process; (2) `props`, a bitvector that describes the VB properties; (3) the number of processes attached to the VB (i.e., a reference count); (4) the type of VBI-to-physical address

[2]The actual physical copy can be accelerated using in-DRAM copy mechanisms such as RowClone [336], LISA [66], and NoM [345].

translation structure being used for the VB; and (5) a pointer to the translation structure. For ease of access, the MTL maintains a *separate* VIT for each size class. The ID of a VB within its size class (VBID) is used as an index into the corresponding VIT. When a VB is enabled (using `enable_vb`), the MTL finds the corresponding VIT and entry using the `SizeID` and `VBID`, respectively (both extracted from `VBUID`). MTL then sets the *enabled* bit of the entry and updates `props`. The reference counter of the VB is also set to 0, indicating that no process is attached to this VB. The type and pointer of the translation structure of the VB are updated in its VIT entry at the time of physical memory allocation (as we discuss in Section 3.4.2). Since a VIT contains entries for the VBs of only a single size class, the number of entries in each VIT equals the number of VBs that the associated size class supports (Section 3.3.1). However, VBI limits the size of each VB Info Table by storing entries only up to the currently-enabled VB with the largest `VBID` in the size class associated with that VB Info Table. The OS ensures that the table does not become prohibitively large by reusing previously-disabled VBs for subsequent requests (Section 3.3.2).

Base Memory Allocation and Address Translation

Our *base* memory allocation algorithm allocates physical memory at 4 KB granularity. Similar to x86-64 [161], Our *base* address translation mechanism stores VBI-to-physical address translation information in multi-level tables. However, unlike the 4-level page tables in x86-64, VBI uses tables with varying number of levels according to the size of the VB. For example, a 4 KB VB does not require a translation structure (i.e., can be direct-mapped) since 4 KB is the minimum granularity of meomry allocation. On the other hand, a 128 KB VB requires a one-level table for translating address to 4 KB regions. As a result, smaller VBs require fewer memory accesses to serve a TLB miss. For each VB, the VIT stores a pointer to the address of the root of the multi-level table (or the base physical address of the directly mapped VBs).

MTL Hardware Complexity

We envision the MTL as software running on a programmable low-power core within the memory controller. While conventional OSes are responsible for memory allocation, virtual-to-physical mapping, and memory protection, the MTL does not need to deal with protection, so we expect the MTL code to be simpler than typical OS memory management software. As a result, the complexity of the MTL hardware is similar to that of prior proposals such as Pinnacle [25] (commercially available) and Page Overlays [344], which perform memory allocation and remapping in the memory controller. While both Pinnacle and Page Overlays are hardware solutions, VBI provides flexibility by making the MTL programmable, thereby allowing software updates for different memory management policies (e.g., address translation, mapping, migration, scheduling). Our goal in this work is to understand the potential of hardware-based memory allocation and address translation.

3.4 Allocation and Translation Optimizations

The MTL employs three techniques to optimize the base memory allocation and address translation described in Section 3.3.5. We explain these techniques in the following subsections.

3.4.1 Delayed Physical Memory Allocation

As described in Section 3.2.5, VBI delays physical memory allocation for a VB (or a region of a VB) until a dirty cache line from that VB (or a region of the VB) is evicted from the last-level cache (LLC). This optimization is enabled by the fact that VBI uses VBI address directly to access *all* on-chip caches. Therefore, a cache line does *not* need to be backed by a physical memory mapping in order to be accessed.

In this approach, when a VB is enabled, VBI does not immediately allocate physical memory to the VB. On an LLC miss to the VB, VBI checks the status of the VB in its corresponding VIT entry. If there is no physical memory backing the data, VBI does one of two things. (1) If the VB corresponds to a memory-mapped file or if the required data was allocated before but swapped out to a backing store, then VBI allocates physical memory for the region, interrupts the OS to copy the relevant data from storage into the allocated memory, and then returns the relevant cache line to the processor. (2) If this is the first time the cache line is being accessed from memory, VBI simply returns a zeroed cache line without allocating physical memory to the VB.

On a dirty cache line writeback from the LLC, if physical memory is yet to be allocated for the region that the cache line maps to, VBI first allocates physical memory for the region, and then performs the writeback. VBI allocates only the region of the VB containing the evicted cache line. As Section 3.3.5 describes, our base memory allocation mechanism allocates physical memory at a 4 KB granularity. Therefore, the region allocated for the evicted cache line is 4 KB. Section 3.4.3 describes an optimization that eagerly *reserves* a larger amount of physical memory for a VB during allocation, to reduce the overall translation overhead.

3.4.2 Flexible Address Translation Structures

For each VB, VBI chooses one of three types of address translation structures, depending on the needs of the VB and the physical memory availability. The first type *directly* maps the VB to physical memory when enough contiguous memory is available. With this mapping, a single TLB entry is sufficient to maintain the translation for the entire VB. The second type uses a single-level table, where the VB is divided into equal-sized blocks of one of the supported size classes. Each entry in the table maintains the mapping for the corresponding block. This mapping exploits the fact that a majority of the data structures are densely allocated inside their respective VBs. With a single-level table, the mapping for any region of the VB can be retrieved with a single memory access. The third type, suitable for sparsely-allocated VBs, is our base address translation mechanism (described in Section 3.3.5), which uses multi-level page tables where the table depth is chosen based on the size of the VB.

In our evaluation, we implement a flexible mechanism that statically chooses a translation structure type based on the size of the VB. Each 4 KB VB is directly mapped. 128 KB and 4 MB VBs use a single-level table. VBs of a larger size class use a multi-level table with as many levels as necessary to map the VB using 4 KB pages.[3] The *early reservation* optimization (described in Section 3.4.3) improves upon this static policy by dynamically choosing a translation structure type from the three types mentioned above based on the available contiguous physical memory. While we evaluate table-based translation structures in this work, VBI can be easily extended to support other structures (e.g., customized per-application translation structures as proposed in DVMT [10]).

Similar to x86-64, VBI uses multiple types of TLBs to cache mappings of different granularity. The type of translation structure used for a VB is stored in the VIT and is cached in the on-chip VIT Cache. This information enables VBI to access the right type of TLB. For a fair comparison, our evaluations use the same TLB type and size for all baselines and variants of VBI.

3.4.3 Early Reservation of Physical Memory

VBI can perform early reservation of the physical memory for a VB. To this end, VBI reserves (but does not allocate) physical memory for the entire VB at the time of memory allocation, and treats the VB as *directly mapped* by serving future memory allocation requests for that VB from that contiguous reserved region. This optimization is inspired by prior work on super-page management [280], which reserves a larger contiguous region of memory than the requested size, and upgrades the allocated pages to larger super-pages when enough contiguous pages are allocated in that region.

For VBI's early reservation optimization, at the time of the *first* physical memory allocation request for a VB, the MTL checks if there is enough contiguous free space in physical memory to fit the entire VB. If so, it allocates the requested memory from that contiguous space, and marks the remaining free blocks in that contiguous space as reserved for that specific VB. In order to reduce internal fragmentation when free physical memory is running low, physical blocks reserved for a VB may be used by another VB when no unreserved blocks are available. As a result, the MTL uses a three-level priority when allocating physical blocks: (1) free blocks reserved for the VB that is demanding allocation, (2) unreserved free blocks, and (3) free blocks reserved for other VBs. A VB is considered directly mapped as long as all its allocated memory is mapped to a single contiguous region of memory, thereby requiring just a single TLB entry for the entire VB. If there is not enough contiguous physical memory available to fit the entire VB, the early reservation mechanism allocates the VB sparsely by reserving blocks of the largest size class that can be allocated contiguously.

[3]For fair comparison with conventional virtual memory, our evaluations use a 4 KB granularity to map VBs to physical memory. However, VBI can flexibly map VBs at the granularity of any available size class.

With the early reservation approach, memory allocation is performed at a different granularity than mapping, which enables VBI to benefit from larger mapping granularities and thereby minimize the address translation latency, while eliminating memory allocation for regions that may never be accessed. To support the early reservation mechanism, VBI uses the Buddy algorithm [199, 348] to manage free and reserved regions of different size classes.

3.5 VBI in Other System Architectures

VBI is designed to easily and efficiently function in various system designs. We describe the implementation of VBI in two important examples of modern system architectures: virtualized environments and multi-node systems.

3.5.1 Supporting Virtual Machines

VBI implements address space isolation between virtual machines (VMs) by *partitioning* the global VBI address space among multiple VMs and the host OS. To this end, VBI reserves a few bits in the VBI address for the `VM ID`. Figure 3.5 shows how VBI implements this for a system supporting 31 virtual machines (ID 0 is reserved for the host). In the VBI address, the 5 bits following the size class bits are used to denote the `VM ID`. For every new virtual machine in the system, the host OS assigns a `VM ID` to be used by the guest OS while assigning virtual blocks to processes inside the virtual machine. VBI partitions client IDs using a similar approach. With address space division between VMs, a guest VM is unaware that it is virtualized, and it can allocate/deallocate/access VBs *without* having to coordinate with the host OS. Sharing VBs across multiple VMs is possible, but requires explicit coordination with the host OS.

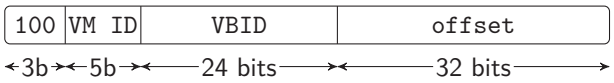


Figure 3.5: Partitioning the VBI address space among virtual machines, using the 4 GB size class (`100`) as an example.

3.5.2 Supporting Multi-Node Systems

There are many ways to implement VBI in multi-node systems. Our initial approach provides each node with its own MTL. VBI equally partitions VBs of each size class among the MTLs, with the higher order bits of `VBID` indicating the VB's home MTL. The home MTL of a VB is the only MTL that manages the VB's physical memory allocation and address translation. When allocating a VB to a process, the OS attempts to ensure that the VB's home MTL is in the same node as the core executing the process. During phase changes, the OS can seamlessly migrate data from a VB hosted by one MTL to a VB hosted by another MTL. We leave the evaluation of this approach and exploration of other ways of integrating VBI with multi-node systems to future work.

3.6 Evaluation

We evaluate VBI for two concrete use cases. First, we evaluate how VBI reduces address translation overheads in native and virtualized environments (Section 3.6.2 and Section 3.6.2, respectively). Second, we evaluate the benefits that VBI offers in harnessing the full potential of two main memory architectures that are tightly dependent on the data mapping: (1) a hybrid PCM–DRAM memory architecture; and (2) TL-DRAM [218], a heterogeneous-latency DRAM architecture (Section 3.6.3).

3.6.1 Methodology

For our evaluations, we use a heavily-customized version of Ramulator [196] to faithfully model all components of the memory subsystem (including TLBs, page tables, the page table walker, and the page walk cache), as well as the functionality of memory management calls (e.g., `malloc`, `realloc`, `free`). We have released this modified version of Ramulator [325]. Table 3.1 summarizes the main simulation parameters. Our workloads consist of benchmarks from SPECspeed 2017 [363], SPEC CPU 2006 [362], TailBench [135], and Graph 500 [122]. We identify representative code regions for the SPEC benchmarks using SimPoint [303]. For TailBench applications, we skip the first five billion instructions. For Graph 500, we mark the region of interest directly in the source code. We use an Intel Pintool [247] to collect traces of the representative regions of each of our benchmarks. For our evaluations, we first warm up the system with 100 million instructions, and then run the benchmark for 1 billion instructions.

CPU	4-wide issue, OOO, 128-entry ROB
L1 Cache	32 KB, 8-way associative, 4 cycles
L2 Cache	256 KB, 8-way associative, 8 cycles
L3 Cache	8 MB (2 MB per-core), 16-way associative, 31 cycles
L1 DTLB	4 KB pages: 64-entry, fully associative 2 MB pages: 32-entry, fully associative
L2 DTLB	4 KB and 2 MB pages: 512-entry, 4-way associative
Page Walk Cache	32-entry, fully associative
DRAM	DDR3-1600, 1 channel, 1 rank/channel 8 banks/rank, open-page policy
DRAM Timing [263]	tRCD=5cy, tRP=5cy, tRRDact=3cy, tRRDpre=3cy
PCM	PCM-800, 1 channel, 1 rank/channel, 8 banks/rank
PCM Timing [211]	tRCD=22cy, tRP=60cy, tRRDact=2cy, tRRDpre=11cy

Table 3.1: Simulation configuration.

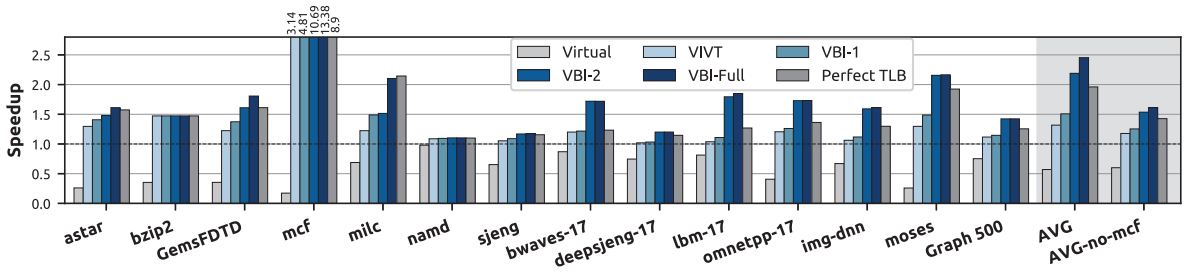


Figure 3.6: Performance of systems with 4KB pages (normalized to Native).

3.6.2 Use Case 1: Address Translation

We evaluate the performance of seven baseline systems to compare with VBI: (1) **Native**: applications run natively on an x86-64 system with only 4 KB pages; (2) **Native-2M**: Native but with only 2 MB pages; (3) **Virtual**: applications run inside a virtual machine with only 4 KB pages; (4) **Virtual-2M**: Virtual but with only 2 MB pages;[4] (5) **Perfect TLB**: an unrealistic version of Native with no L1 TLB misses (i.e., no address translation overhead); (6) **VIVT**: Native with VIVT on-chip caches; and (7) **Enigma-HW-2M**: applications run natively in a system with Enigma [398]. Enigma uses a system-wide unique intermediate address space to defer address translation until data must be retrieved from physical memory. A centralized translation cache (CTC) at the memory controller performs intermediate-to-physical address translation. However, unlike VBI, Enigma asks the OS to perform the translation on a CTC miss, and to explicitly manage address mapping. Therefore, Enigma's benefits do not seamlessly extend to programs running inside a virtual machine. We evaluate Enigma with a 16K-entry centralized translation cache (CTC) that we enhance with hardware-managed page walks and 2 MB pages.

We evaluate the performance of three VBI systems: (1) **VBI-1**: inherently virtual caches (Section 3.2.5) along with our *flexible translation mechanism* that maps VBs using a 4 KB granularity (Section 3.3.5) , (2) **VBI-2**: VBI-1 with *delayed physical memory allocation* (allocates the 4 KB region of the VB that the dirty cache line evicted from the last-level cache belongs to). (Section 3.4.1), and (3) **VBI-Full**: VBI-2 with *early reservation* (Section 3.4.3). VBI-1 and VBI-2 manage memory at 4 KB granularity, while VBI-Full uses early reservation to support all of the size classes listed in Section 3.3.1 for VB allocation, providing similar benefits to large page support and direct mapping. We first present results comparing VBI-1 and VBI-2 with Native, Virtual, VIVT, and Perfect TLB (Section 3.6.2). We then present results comparing VBI-Full with Native-2M, Enigma-HW-2M, and Perfect TLB (Section 3.6.2).

Results with 4 KB Pages

Figure 3.6 plots the performance of Virtual, VIVT, VBI-1, VBI-2, and Perfect TLB normalized to the performance of Native, for a single-core system. We also show VBI-Full as a reference that shows the full potentials of VBI which VBI-1 and VBI-2 do not enable. *mcf* has an overwhelmingly high number of TLB misses. Consequently, mechanisms that reduce TLB misses greatly improve *mcf*'s performance, to the point of skewing the average significantly. Therefore, the figure also presents the average speedup without *mcf*. We draw five observations from the figure.

First, VBI-1 outperforms Native by 50%, averaged across all benchmarks (25% without *mcf*). This performance gain is a direct result of (1) inherently virtual on-chip caches in VBI that reduce the number of address translation requests, and (2) fewer levels of address

[4]We augment this system with a 2D page walk cache, which is shown to improve the performance of guest workloads [42].

translation for smaller VBs, which reduces the number of translation-related memory accesses (i.e., page walks).

Second, Perfect TLB serves as an upper bound for the performance benefits of VBI-1. However, by employing flexible translation structures, VBI-1 bridges the performance gap between Native and Perfect TLB by 52%, on average.

Third, when accessing regions for which no physical memory is allocated yet, VBI-2 avoids *both* the memory requests themselves and any translation-related memory accesses for those requests. Therefore, VBI-2 enables benefits over and beyond solely reducing the number of page walks, as it further improves the overall performance by reducing the number of memory requests accessing the main memory as well. Consequently, for many memory-intensive applications, VBI-2 outperforms Perfect TLB. Compared to Perfect TLB, VBI-2 reduces the total number of DRAM accesses (including the translation-related memory accesses) by 62%, averaged across applications that outperform Perfect TLB, and by 46% across all applications. Overall, VBI-2 outperforms Native by an average of 118% (53% without *mcf*).

Fourth, by performing address translations *only for and in parallel with* LLC accesses, VIVT outperforms Native by 31% on average (17% without *mcf*). This performance gain is due to reducing the number of translation requests and therefore decreasing the number of TLB misses using VIVT caches. However, VBI-1 and VBI-2 gain an extra 19% and 87% performance on average, respectively, over VIVT. These improvements highlight VBI's ability to improve performance beyond only employing VIVT caches.

Finally, our results indicate that due to considerably higher translation overhead, Virtual significantly slows down applications compared to Native (44% on average). As described in Section 3.2.5, once an application running inside a virtual machine is attached to its VBs, VBI incurs no additional translation overhead compared to running natively. As a result, in virtualized environments that use only 4K pages, VBI-1 and VBI-2 achieve an average performance of 2.6× and 3.8×, respectively, compared to Virtual.

We conclude that even when mapping and allocating VBs using 4 KB granularity only, both VBI-1 and VBI-2 provide large benefits over a wide range of baseline systems, due to their effective optimizations to reduce address translation and memory allocation overheads. VBI-Full further improves performance by mapping VBs using larger granularities (as we elaborate in Section 3.6.2).

Results with Large Pages

Figure 3.7 plots the performance of Virtual-2M, Enigma-HW-2M, VBI-Full, and Perfect TLB normalized to the performance of Native-2M. We enhance the original design of Enigma [398] by replacing the OS system call handler for address translation on a CTC miss with a completely hardware-managed address translation, similar to VBI. For legibility, the figure shows results for only a subset of the applications. However, the chosen applications capture the behavior of all the applications, and the average (and average without *mcf*) is calculated across all evaluated applications. We draw three observations from the figure.

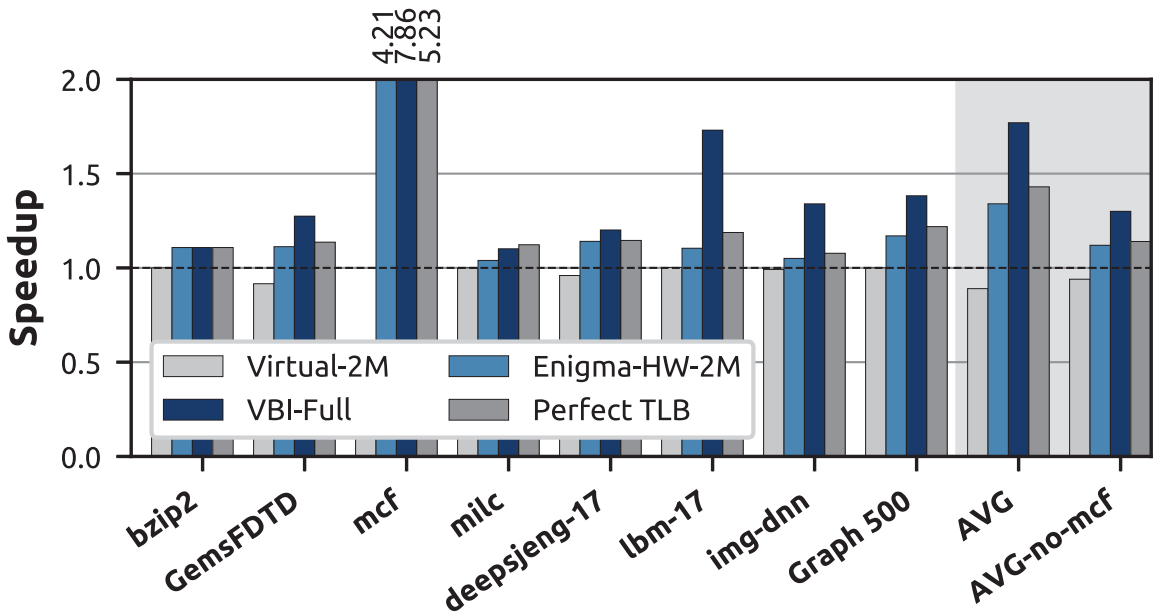


Figure 3.7: Performance with large pages (norm. to Native-2M).

First, managing memory at 2 MB granularity improves the performance of applications compared to managing memory at 4 KB granularity. This is because (1) the larger page size lowers the average TLB miss count (e.g., 66% lower for Native-2M compared to Native), and (2) requires fewer page table accesses on average to serve TLB misses (e.g., 73% fewer for Native-2M compared to Native).

Second, Enigma-HW-2M improves overall performance for programs running *natively* on the system by 34% compared to Native-2M, averaged across all benchmarks (including mcf). The performance gain is a direct result of (1) the very large CTC (16K entries), which reduces the number of translation-related memory accesses by 89% on average compared to Native-2M; and (2) our hardware-managed address translation enhancement, which removes the costly system calls on each page walk request.

Third, VBI-Full, with all three of our optimizations in Section 3.4, maps most VBs using direct mapping, thereby significantly reducing the number of TLB misses and translation-related memory accesses compared to Native-2M (on average by 79% and 99%, respectively). In addition, VBI-Full retains the benefits of VBI-2, which reduces the number of *overall* DRAM accesses. VBI-Full reduces the *total* number of DRAM accesses (including translation-related memory accesses) by 56% on average compared to Perfect TLB. Consequently, VBI-Full outperforms all four comparison points including Perfect TLB. Specifically, VBI-Full improves performance by 77% compared to Native-2M, 43% compared to Enigma-HW-2M and 89% compared to Virtual-2M.

We conclude that by employing all of the optimizations that it enables, VBI significantly outperforms all of our baselines in both native and virtualized environments.

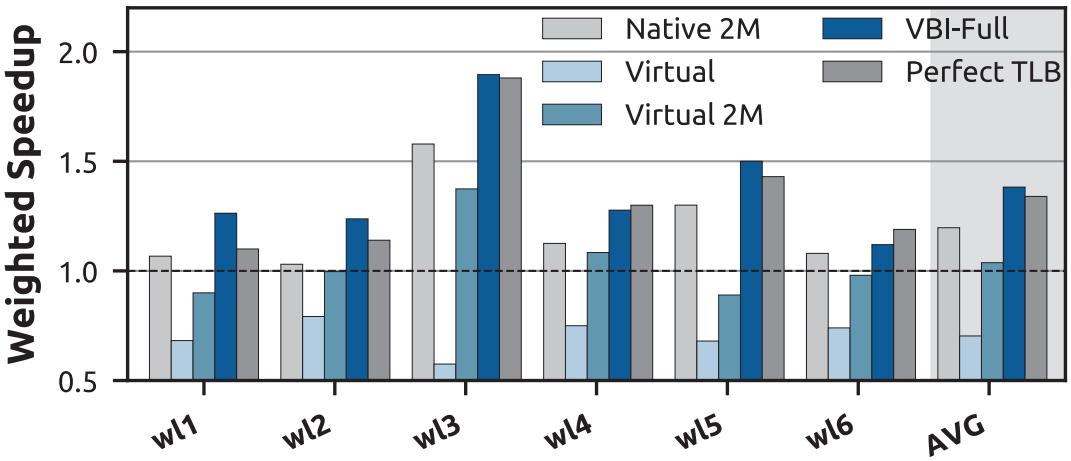


Figure 3.8: Multiprogrammed workload performance (normalized to Native).

Multicore Evaluation

Figure 3.8 compares the weighted speedup of VBI-Full against four baselines in a quad-core system. We examine six different workload bundles, listed in Table 3.2, which consist of the applications studied in our single-core evaluations. From the figure, we make two observations. First, averaged across all bundles, VBI-Full improves performance by 38% and 18%, compared to Native and Native-2M, respectively. Second, VBI-Full outperforms Virtual and Virtual-2M by an average 67% and 34%, respectively. We conclude that the benefits of VBI persist even in the presence of higher memory load in multicore systems.

3.6.3 Use Case 2: Memory Heterogeneity

As mentioned earlier, extracting the best performance from heterogeneous-latency DRAM architectures [64, 66, 186, 216, 218, 246, 248, 336, 358] and hybrid memory architectures [81, 167, 232, 311, 315, 317, 318, 392, 394, 400] critically depends on mapping data to the memory that suits the data requirements, and migrating data as its requirements change. We quantitatively show the performance benefits of VBI in exploiting heterogeneity by evaluating (1) a PCM–DRAM hybrid memory [317]; and (2) TL-DRAM [218], a heterogeneous-latency DRAM architecture. We evaluate five systems: (1) VBI PCM–DRAM and (2) VBI TL-DRAM, in which VBI maps and migrates frequently-accessed VBs to the low-latency memory (the fast memory region in the case of TL-DRAM); (3) Hotness-Unaware PCM–DRAM and (4) Hotness-Unaware TL-DRAM, where the mapping mechanism is unaware of the hotness (i.e., the access frequency) of the data and therefore do not necessarily map the frequently-accessed data to the fast region; and (5) IDEAL in each plot refers to an unrealistic perfect mapping mechanism, which uses oracle knowledge to always map frequently-accessed data to the fast portion of memory.

wl1	deepsjeng, omnetpp, bwaves, lbm	**w**l4	milc, namd, GemsFDTD, bzip2
wl2	graph500, astar, img-dnn, moses	**w**l5	bzip2, GemsFDTD, sjeng, mcf
wl3	mcf, GemsFDTD, astar, milc	**w**l6	namd, bzip2, astar, sjeng

Table 3.2: Multiprogrammed workload bundles.

Figures 3.9 and 3.10 show the speedup obtained by VBI-enabled mapping over the hotness-unaware mapping in a PCM–DRAM hybrid memory and in TL-DRAM, respectively. We draw three observations from the figures. First, for PCM–DRAM, VBI PCM–DRAM improves performance by 33% on average compared to the Hotness-Unaware PCM–DRAM, by accurately mapping the frequently-accessed data structures to the low-latency DRAM. Second, by mapping frequently-accessed data to the fast DRAM regions, VBI TL-DRAM takes better advantage of the benefits of TL-DRAM, with a performance improvement of 21% on average compared to Hotness-Unaware TL-DRAM. Third, VBI TL-DRAM performs only 5.3% slower than IDEAL, which is the upper bound of performance achieved by mapping hot data to the fast regions of DRAM.

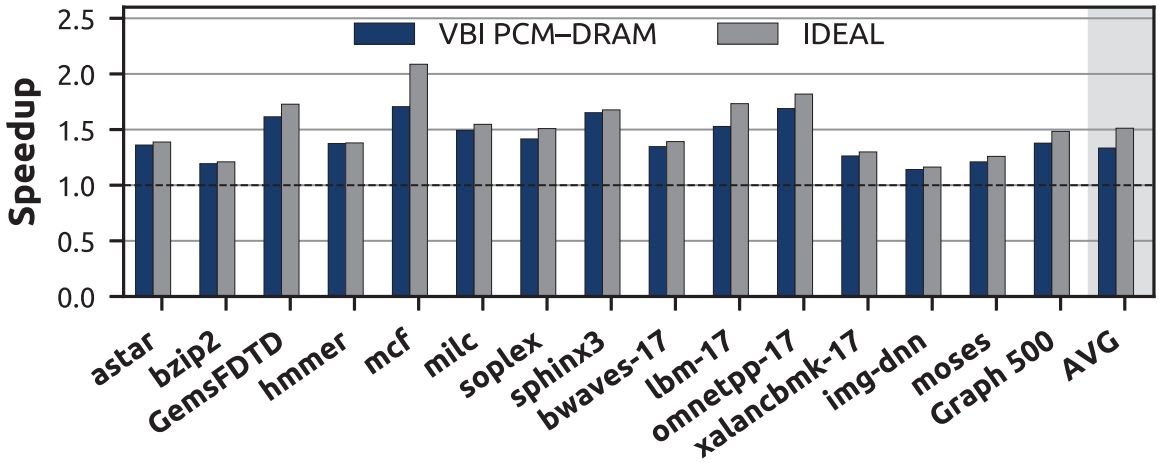


Figure 3.9: Performance of VBI PCM-DRAM (normalized to data-hotness-unaware mapping).

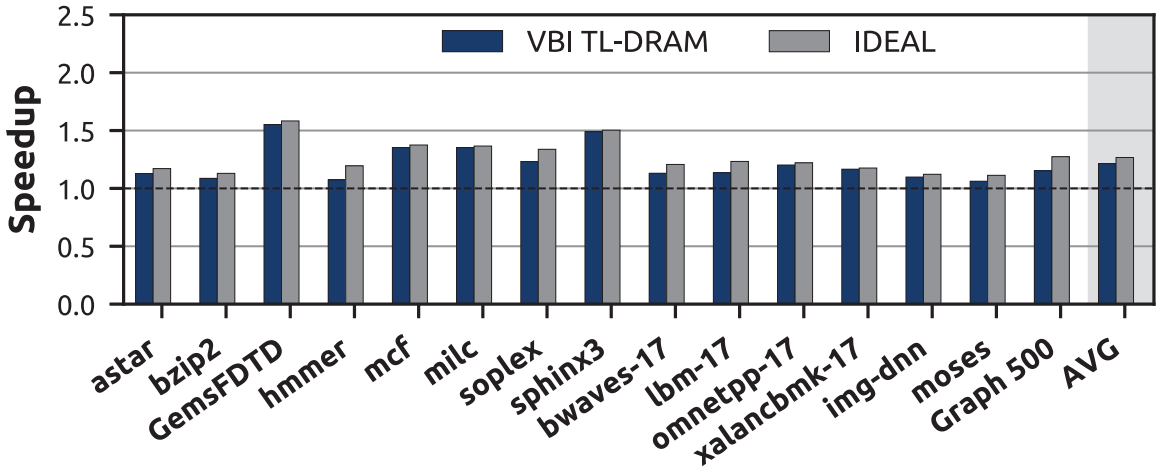


Figure 3.10: Performance of VBI TL-DRAM (normalized to data-hotness-unaware mapping).

We conclude that VBI is effective for enabling efficient data mapping and migration in heterogeneous memory systems.

3.7 Related Work

To our knowledge, VBI is the first virtual memory framework to fully delegate physical memory allocation and address translation to the hardware. This section compares VBI with other virtual memory designs and related works.

Virtual Memory in Modern Architectures. Modern virtual memory architectures, such as those employed as part of modern instruction set architectures [24,143,161,162], have evolved into sophisticated systems. These architectures have support for features such as large pages, multi-level page tables, hardware-managed TLBs, and variable-size memory segments, but require significant system software support to enable these features and to manage memory. While system software support provides some flexibility to adapt to new ideas, it must communicate with hardware through a rigid contract. Such rigid hardware/software communication introduces costly overheads for many applications (e.g., high overheads with fixed-size per-application virtual address spaces, for applications that only need a small fraction of the space) and prevents the easy adoption of significantly different virtual memory architectures or ideas that depend on large changes to the existing virtual memory framework. VBI is a completely different framework from existing virtual memory architectures. It supports the functionalities of existing virtual memory architectures, but can do much more by reducing translation overheads, inherently and seamlessly supporting virtual caches, and avoiding unnecessary physical memory allocation. These benefits come from enabling completely hardware-managed physical memory allocation and address translation, which no other virtual memory architecture does (including, for example, Multics [37,38,74]).

Several memory management frameworks [27,28,224,308,312,376] are designed to minimize the virtual memory overhead in GPUs. Unlike VBI, these works provide optimizations *within the existing virtual memory design*, so their benefits are constrained to the design of conventional virtual memory.

OS Support for Virtual Memory. There has been extensive work on how address spaces should be mapped to execution contexts [236]. Unix-like OSes provide a rigid one-to-one mapping between virtual address spaces and processes [255,320]. SpaceJMP [86] proposes a design in which processes can jump from one virtual address space to another in order to access larger amounts of physical memory. Single address space OSes rely on system-software-based mechanisms to expose a single global address space to processes, to facilitate efficient data sharing between processes [67,68,142]. VBI makes use of a similar concept as single address space OSes with its single globally-visible VBI address space. However, while existing single address space OS designs expose the single address space to processes, VBI does not do so, and instead has processes operate on CVT-relative virtual addresses. This allows VBI to enjoy the same advantages as single address space OSes (e.g., synonym-/homonym-free VIVT caches), while providing further benefits (e.g., non-fixed addresses for shared libraries, hardware-based memory management). Additionally, VBI

naturally supports single address space sharing between the host OS and guest OSes in virtualized environments.

User-Space Memory Management. Several OS designs propose user-space techniques to provide an application with more control over memory management [10, 36, 87, 88, 133, 169, 198, 330, 382]. For example, the exokernel OS architecture [88, 169] allows applications to manage their own memory and provides memory protection via capabilities, thereby minimizing OS involvement. Do-It-Yourself Virtual Memory Translation (DVMT) [10] decouples memory translation from protection in the OS, and allows applications to handle their virtual-to-physical memory translation. These solutions (1) increase application complexity and add non-trivial programmer burden to directly manage hardware resources, and (2) do not expose the rich runtime information available in the hardware to memory managers. In contrast to these works, which continue to rely on software for physical memory management, VBI does not use *any* part of the software stack for physical memory management. By partitioning the duties differently between software and hardware, and, importantly, performing physical memory management in the memory controller, VBI provides similar flexibility benefits as user-space memory management without introducing additional programmer burden.

Reducing Address Translation Overhead. Several studies have characterized the overhead of virtual-to-physical address translation in modern systems, which occurs primarily due to growing physical memory sizes, inflexible memory mappings, and virtualization [34, 46, 146, 161, 176, 256]. Prior works try to ameliorate the address translation issue by: (1) increasing the TLB reach to address a larger physical address space [27, 72, 174, 297, 304, 305, 323, 367]; (2) using TLB speculation to speed up address translation [30, 33, 296, 307]; (3) introducing and optimizing page walk caches to store intermediate page table addresses [32, 42, 43, 89]; (4) adding sharing and coherence between caching structures to share relevant address translation updates [41, 45, 89, 177, 204, 305, 321, 391]; (5) allocating and using large contiguous regions of memory such as superpages [27, 34, 105–107, 129, 306]; (6) improving memory virtualization with large, contiguous memory allocations and better paging structures [27, 105, 106, 306, 307, 323]; (7) prioritizing page walk data throughout the memory hierarchy [28]; and (8) reducing the overheads associated with address translation and maintaining TLB consistency in shared-memory multiprocessors [1, 49, 368, 369, 385]. While all of these works can mitigate the translation overhead, they build on top of the existing rigid virtual memory framework and do not address the underlying overheads inherent to the existing rigid framework and to software-based memory management. Additionally, several prior works propose mechanisms for fine grain protection domains and sub-page sharing (e.g., [123, 371]), for example by exploiting the contiguity of fine grained permission rights across larger address ranges [383, 384]. However, these techniques do not address the other issues and overheads associated with the conventional virtual memory frameworks such as high address translation overhead. Unlike these works, VBI is a completely new framework for virtual

memory, which eliminates several underlying sources of address translation overhead and enables many other benefits (e.g., efficient memory management in virtual machines, easy extensibility to heterogeneous memory systems). VBI can be combined with some of the above proposals to further optimize address translation.

3.8 Summary and Contributions

We introduce the Virtual Block Interface (VBI), a new virtual memory framework to address the challenges in adapting conventional virtual memory to increasingly diverse system configurations and workloads. The key idea of VBI is to delegate memory management to hardware in the memory controller. The memory-controller-based memory management in VBI leads to many benefits not easily attainable in existing virtual memory, such as inherently virtual caches, avoiding 2D page walks in virtual machines, and delayed physical memory allocation. We experimentally show that VBI (1) reduces the overheads of address translation by reducing the number of translation requests and associated memory accesses, and (2) increases the effectiveness of managing heterogeneous main memory architectures. We conclude that VBI is a promising new virtual memory framework that can enable several important optimizations and increased design flexibility for virtual memory. We believe and hope that VBI will open up a new direction and many opportunities for future work in novel virtual memory frameworks.

In this chapter, we make the following key contributions:

- We propose the first virtual memory framework that relieves the OS of explicit physical memory management and delegates this duty to the hardware, i.e., the memory controller. propose VBI, a new virtual memory framework that efficiently enables memory-controller-based memory management by exposing a purely virtual memory interface to applications, the OS, and the hardware caches. VBI naturally and seamlessly supports several optimizations (e.g., low-cost page walks in virtual machines, purely virtual caches, delayed physical memory allocation), and integrates well with a wide range of system designs.
- We provide a detailed reference implementation of VBI, including required changes to the user applications, system software, ISA, and hardware.
- We quantitatively evaluate VBI using two concrete use cases: (1) address translation improvements for native execution and virtual machines, and (2) two different heterogeneous memory architectures. Our evaluations show that VBI significantly improves performance in both use cases.

Chapter 4

Conclusions and Future Work

4.1 Conclusions

The goal of this thesis is to enable efficient data handling in modern computing systems via new frameworks that are developed based on a fundamental rethinking of the computing paradigm and key concepts and components in modern computing systems with the goal to make them data-centric and data-aware.

In this thesis, we demonstrate that the overall performance and efficiency of the system can improve significantly using (1) data-centric architectures that minimize data movement and compute data in or near where the data resides, and (2) data-aware frameworks that understand what can be done with and to each piece of data and makes use of different properties of data (e.g., compressibility, approximability, locality, sparsity, access semantics) to improve performance, efficiency and other metrics. We propose two novel frameworks that follow these fundamental guiding principles.

First, we propose SIMDRAM, an end-to-end processing-using-DRAM framework that follows the data-centric approach and provides the programming interface, the ISA, and the hardware support for: (1) efficiently computing complex operations inside DRAM chips, the predominant main memory technology, and (2) providing the ability to implement arbitrary operations as required. SIMDRAM achieves this using an in-DRAM massively-parallel SIMD substrate that requires minimal changes to the DRAM architecture. We show that SIMDRAM significantly improves the performance and the energy efficiency of the system when computing a wide variety of complex operations and commonly-used real-world applications. We conclude that SIMDRAM is a promising processing-using-memory framework that can enable significant performance and efficiency improvements in the system by efficiently computing complex and arbitrary operations in DRAM. We believe and hope that future work builds on our framework to further ease the adoption and improve the performance and efficiency of processing-using-DRAM architectures and applications.

Second, we introduce the Virtual Block Interface (VBI), an alternative virtual memory framework that follows the data-aware approach and (1) addresses the important challenges in adapting conventional virtual memory to increasingly large and diverse data demand in modern applications, (2) understands, conveys, and exploits the properties of different pieces of program data to enable more intelligent management of main memory, and (3) efficiently and flexibly supports increasingly diverse system configurations that are employed today to process the high data demand in modern applications. VBI achieves these while providing the key features of the conventional virtual memory frameworks. As two example use cases of the VBI framework, we show that (1) VBI significantly improves the overall system performance for both native execution and virtual machine environments, and (2) VBI significantly improves the effectiveness of heterogeneous main memory architectures. We conclude that VBI is a promising new virtual memory framework, that can enable several important optimizations, and increase the design flexibility for virtual memory to support efficient handling of data in modern computing systems. We believe and hope that VBI will open up a new direction and many opportunities for future work in novel virtual memory frameworks.

4.2 Future Work

This dissertation opens up many new research directions and opportunities. In this section, we discuss several major high-level future directions in which the ideas and approaches presented in this thesis can be extended to tackle other issues in modern computing systems regarding efficient handling of large amount of data in modern applications.

4.2.1 Data-Aware Memory Architectures

As we showed in Chapter 3, conveying the properties of different pieces of program data to hardware can enable significant performance optimizations. While conveying data properties is difficult to implement on top of conventional virtual memory, VBI (Chapter 3) is designed from the ground up to efficiently convey properties of program data to the hardware, including the memory. We believe that data-aware memories, i.e., memory architectures that understand and exploit the properties of the data to make intelligent utilization decisions, can significantly transform the computing landscape, by exploiting information previously unavailable to the memory in an easy and flexible manner. The native support for conveying data properties to the hardware in VBI can enable a wide range of research and development in this area, serving as a platform to demonstrate numerous data-aware hardware optimizations in main memory management.

4.2.2 Enabling Support for Designing New Unconventional Memory Subsystems

As we discuss in Chapter 3, conventional OS-based virtual memories are unable to efficiently adapt to and fully exploit today's diverse memory designs (e.g., hybrid memory systems), as the OS has poor visibility into the physical memory architecture and lacks the rich fine-grained information on runtime memory behavior that is essential in managing the memory resources. VBI (Chapter 3) provides significant flexibility in enabling efficient virtual memory support for emerging memory architectures, without breaking the interfaces that programmers are used to, and without requiring bespoke virtual memory architectures for each different system design. In this thesis, we showed how VBI can provide efficient support for heterogeneous memory systems, which are becoming widely available (e.g., systems with Optane memory modules side-by-side with DRAM). Given its flexibility and customization opportunities, we believe that VBI can serve as a foundation for systems that incorporate non-traditional memory subsystems, and opens up many new research opportunities in using emerging memory technologies (e.g., combining main memory and storage devices [259]) and designing new unconventional memory subsystems (e.g., potentially using neuromorphic hardware) that can further enhance the efficiency of handling the large amount of data in modern applications.

4.2.3 Virtual Memory Support for Processing-Using-Memory architectures

Processing-using-memory reduces/eliminates the need to move data from the main memory to the processor for computation. SIMDRAM (Chapter 2) eases the adoption of processing-using-DRAM architectures by enabling efficient implementation of complex operations and providing the ability to perform any arbitrary operations. However, full adoption of processing-using-memory solutions such as SIMDRAM requires support for some key virtual memory functionalities. More specifically, relying on the CPU to provide the processing-using-memory architectures with address translation, memory allocation, and potentially memory security can potentially nullify the very benefits of processing-using-memory approach which aims to reduce the interaction between the main memory and the CPU. Accordingly, designing efficient support in processing-using-memory architectures for critical virtual memory functionalities such as address translation, memory allocation, and security mechanisms is a promising research direction.

Chapter 5

Other Works of the Author

In addition to the works presented in this thesis, I have also contributed to several other research works done in collaboration with SAFARI Research Group members at CMU and ETH. In this section, I briefly overview these works.

Expressive Memory (XMem) [377]: Programs are traditionally conveyed to the hardware in the form of ISA instructions and a set of memory accesses to virtual addresses. This semantic gap leads to hardware treating all data as the same, thereby being unable to exploit data's semantics properties to employ more intelligent management or optimization policies. This work introduces a new cross-layer interface, called Expressive Memory (XMem), to communicate higher-level program semantics from the application to the system software and hardware architecture. By bridging the semantic gap, XMem provides two key benefits. First, it enables architectural/system-level techniques to leverage key program semantics that are challenging to predict or infer. Second, it improves the efficacy and portability of software optimizations by alleviating the need to tune code for specific hardware resources (e.g., cache space). This work was published in ISCA 2018 [377].

CoNDA [53, 54]: Recent advances in memory technology have enabled near-data accelerators (NDAs), which are located off-chip, close to main memory. The lack of an efficient communication mechanism between CPUs and accelerators creates a significant overhead to synchronize data updates between the two. Accordingly, enforcing coherence with the rest of the system, which is already a major challenge for accelerators, becomes more difficult for NDAs. This work introduces CoNDA, a coherence mechanism that lets an NDA optimistically execute an NDA kernel, under the assumption that the NDA has all necessary coherence permissions. This optimistic execution allows CoNDA to gather information on the memory accesses performed by the NDA and by the rest of the system. CoNDA then exploits this information to avoid performing unnecessary coherence requests, and as a result, reduce the data movement for coherence significantly. CoNDA was published

in ISCA 2019, and an earlier version of it, LazyPIM, in IEEE CAL [54].

Demystifying Complex Workload-DRAM Interactions [114, 115]: With the increasingly diversifying application behavior and the wide array of available DRAM types, it has become very difficult to identify the best DRAM type for a given workload. Much of this difficulty lies in the complex interaction between memory access latency, bandwidth, parallelism, energy consumption, and application memory access patterns. Importantly, changes made by DRAM vendors in new DRAM types can significantly affect the behavior of an application in ways that are often difficult to intuitively and easily understand. This work identifies important families of workloads, as well as widely used types of DRAM chips, and comprehensively analyze the combined DRAM–workload behavior. We provide an experimental study of the interaction between nine different DRAM types and 115 modern applications and multi-programmed workloads. Furthermore, we perform a rigorous experimental characterization of system performance and DRAM energy consumption, and introduce new metrics to capture the sophisticated interactions between memory access patterns and the underlying hardware. The trends identified from the characterization performed in this work can drive optimizations in both hardware and software design. This work was published in SIGMETRICS 2019 [114].

AirLift [324]: Genome sequencing is a technique that determines the DNA sequence of an organism. Modern genome sequencing machines [154–156, 293–295, 332] extract small random fragments of the original DNA sequence, known as reads [12–14, 190, 331, 387, 388]. To adapt an existing genomic study (i.e., read sets from many samples) to a new reference genome, we need to remap the reads (i.e., update a read's alignment location from the original (old) reference to another (new) reference). AirLift presents a methodology for quickly and comprehensively mapping a set of reads from one reference to another reference. AirLift is the first methodology and tool that leverages the similarity between two reference genomes to (1) substantially reduce the time to remap a read set from an old (i.e., previously mapped to) reference genome to a new reference genome, (2) comprehensively remap a read set, i.e., attempt to remap all reads in a read set, (3) provide accurate remapping results, i.e., provide alignments with error rates below a specified acceptable error rate, and (4) provide an end-to-end remapping solution on which downstream analysis (e.g., variant calling) can be immediately performed.

Bibliography

[1] Scott A Ritchie. TLB For Free: In-Cache Address Translation For A Multiprocessor. In *Technical rept.*, 1985.

[2] Reto Achermann, Chris Dalton, Paolo Faraboschi, Moritz Hoffmann, Dejan Milojicic, Geoffrey Ndu, Alexander Richardson, Timothy Roscoe, Adrian L. Shaw, and Robert N. M. Watson. Separating Translation from Protection in Address Spaces with Dynamic Remapping. In *HotOS*, 2017.

[3] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines. In *ASPLOS*, 2020.

[4] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. Compute Caches. In *HPCA*, 2017.

[5] Thomas Ahearn, Robert Capowski, Neal Christensen, Patrick Gannon, Arlin Lee, and John Liptay. Virtual Memory System, 1973.

[6] Hameeza Ahmed, Paulo C Santos, João P .C. Lima, Rafael F Moura, Marco A. Z. Alves, Antônio C. S. Beck, and Luigi Carro. A Compiler for Automatic Selection of Suitable Processing-in-Memory Instructions. In *DATE*, 2019.

[7] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. In *ISCA*, 2015.

[8] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture. In *ISCA*, 2015.

[9] Berkin Akin, Franz Franchetti, and James C Hoe. Data Reorganization in Memory Using 3D-Stacked DRAM. In *ISCA*, 2016.

[10] Hanna Alam, Tianhao Zhang, Mattan Erez, and Yoav Etsion. Do-It-Yourself Virtual Memory Translation. In *ISCA*, 2017.

[11] Mustafa F Ali, Akhilesh Jaiswal, and Kaushik Roy. In-Memory Low-Cost Bit-Serial Addition Using Commodity DRAM Technology. In *TCAS-I*, 2019.

[12] Mohammed Alser, Hasan Hassan, Akash Kumar, Onur Mutlu, and Can Alkan. Shouji: A Fast and Efficient Pre-Alignment Filter for Sequence Alignment. In *Bioinformatics Journal*, 2019.

[13] Mohammed Alser, Taha Shahroodi, Juan-Gomez Luna, Can Alkan, and Onur Mutlu. SneakySnake: A Fast and Accurate Universal Genome Pre-Alignment Filter for CPUs, GPUs, and FPGAs. In *Bioinformatics Journal*, 2020.

[14] Mohammed Alser, Bingol Zulal, Damla Senol Cali, Jeremie S Kim, Saugata Ghose, Can Alkan, and Onur Mutlu. Accelerating Genome Analysis: A Primer on an Ongoing Journey. In *IEEE MICRO*, 2020.

[15] Marco A. Z. Alves, Paulo C. Santos, Matthias Diener, and Luigi Carro. Opportunities and Challenges of Performing Vector Operations Inside the DRAM. In *Int. Symp. on Memory Systems*, MEMSYS.

[16] Luca Amaru, Pierre-Emmanuel Gaillardon, and Giovanni Micheli. Majority-Inverter Graph: A Novel Data-Structure and Algorithms for Efficient Logic Optimization. In *DAC*, 2014.

[17] Nadav Amit. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. In *USENIX*, 2017.

[18] Shaahin Angizi, Naima Ahmed Fahmi, Wei Zhang, and Deliang Fan. PIM-Assembler: A Processing-in-Memory Platform for Genome Assembly. In *DAC*, 2020.

[19] Shaahin Angizi and Deliang Fan. GraphiDe: A Graph Processing Accelerator Leveraging in-DRAM-Computing. In *GLSVLSI*, 2019.

[20] Shaahin Angizi and Deliang Fan. ReDRAM: A Reconfigurable Processing-in-DRAM Platform for Accelerating Bulk Bit-Wise Operations. In *ICCAD*, 2019.

[21] Shaahin Angizi, Zhezhi He, and Deliang Fan. DIMA: A Depthwise CNN In-Memory Accelerator. In *ICCAD*, 2018.

[22] Shaahin Angizi, Zhezhi He, Farhana Parveen, and Deliang Fan. IMCE: Energy-Efficient Bitwise In-Memory Convolution Engine for Deep Neural Network. In *ASP-DAC*, 2018.

[23] ARM Ltd. *Cortex-A8 Technical Reference Manual*, 2010.

[24] Arm Ltd. *Arm® Architecture Reference Manual: ARMv8, for ARMv8-A Architecture Profile*, 2013.

[25] S. Arramreddy, K. Mak, R. B. Tremaine, M. Wazlowski, T. B. Smith, and D. Har. Pinnacle: IBM MXT in a Memory Controller Chip. *IEEE Micro*, 2001.

[26] Hadi Asghari-Moghaddam, Young Hoon Son, Jung Ho Ahn, and Nam Sung Kim. Chameleon: Versatile and practical near-dram acceleration architecture for large memory systems. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press.

[27] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes. In *MICRO*, 2017.

[28] Rachata Ausavarungnirun, Vance Miller, Joshua Landgraf, Saugata Ghose, Jayneel Gandhi, Adwait Jog, Christopher J. Rossbach, and Onur Mutlu. MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency. In *ASPLOS*, 2018.

[29] Oreoluwatomiwa O Babarinsa and Stratos Idreos. JAFAR: Near-Data Processing for Databases. In *SIGMOD*, 2015.

[30] Kavita Bala, M. Frans Kaashoek, and William E. Weihl. Software Prefetching and Caching for Translation Lookaside Buffers. In *OSDI*, 1994.

[31] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H. Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. Near-data processing: Insights from a micro-46 workshop. In *IEEE Micro*.

[32] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation Caching: Skip, Don't Walk (the Page Table). In *ISCA*, 2010.

[33] Thomas W. Barr, Alan L. Cox, and Scott Rixner. SpecTLB: A Mechanism for Speculative Address Translation. In *ISCA*, 2011.

[34] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient Virtual Memory for Big Memory Servers. In *ISCA*, 2013.

[35] Kenneth E. Batcher. Bit-Serial Parallel Processing Systems. In *TC*, 1982.

[36] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *SOSP*, 2009.

[37] A. Bensoussan, Clingen C. T., and R. C. Daley. The Multics Virtual Memory. *SOSP*, 1969.

[38] A. Bensoussan, Clingen C. T., and R. C. Daley. The Multics Virtual Memory: Concepts and Design. *Communications of the ACM 15, 308-318*, 1972.

[39] Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Janda Kacper Kanellopoulos, Konstantinos, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana Stefan, Juan Gómez Luna, Marcin Copik, Lukas Kapp-Schwoerer, Salvatore Di Girolamo, Marek Konieczny, Onur Mutlu, and Torsten Hoefler. SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems. In *arXiv preprint arXiv:2104.07582*, 2021.

[40] Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Janda Kacper Kanellopoulos, Konstantinos, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana Stefan, Juan Gómez Luna, Marcin Copik, Lukas Kapp-Schwoerer, Salvatore Di Girolamo, Marek Konieczny, Onur Mutlu, and Torsten Hoefler. SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems. In *MICRO*, 2021.

[41] Srikant Bharadwaj, Guilherme Cox, Tushar Krishna, and Abhishek Bhattacharjee. Scalable Distributed Shared Last-Level TLBs Using Low-Latency Interconnects. In *MICRO*, 2018.

[42] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating Two-Dimensional Page Walks for Virtualized Systems. In *ASPLOS*, 2008.

[43] Abhishek Bhattacharjee. Large-Reach Memory Management Unit Caches. In *MICRO*, 2013.

[44] Abhishek Bhattacharjee. Translation-Triggered Prefetching. In *ASPLOS*, 2017.

[45] Abhishek Bhattacharjee, Daniel Lustig, and Margaret Martonosi. Shared Last-Level TLBs for Chip Multiprocessors. In *ISCA*, 2011.

[46] Abhishek Bhattacharjee and Margaret Martonosi. Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors. In *PACT*, 2009.

[47] Abhishek Bhattacharjee and Margaret Martonosi. Inter-Core Cooperative TLB Prefetchers for Chip Multiprocessors. In *ASPLOS*, 2010.

[48] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 Simulator. *Comput. Archit. News*, 2011.

[49] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill. Translation Lookaside Buffer Consistency: A Software Approach. In *ASPLOS*, 1989.

[50] Amirali Boroumand, Saugata Ghose, Berkin Akin, Ravi Narayanaswami, Geraldo F Oliveira, Xiaoyu Ma, Eric Shiu, and Onur Mutlu. Mitigating Edge Machine Learning Inference Bottlenecks: An Empirical Study on Accelerating Google Edge Models. In *arXiv preprint arXiv:2103.00768*, 2021.

[51] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. In *ASPLOS*, 2018.

[52] Amirali Boroumand, Saugata Ghose, Geraldo F Oliveira, and Onur Mutlu. Polynesia: Enabling Effective Hybrid Transactional/Analytical Databases with Specialized Hardware/Software Co-Design. In *arXiv preprint arXiv:2103.00798*, 2021.

[53] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungnirun, Kevin Hsieh, Nastaran Hajinazar, Krishna T Malladi, Hongzhong Zheng, and Onur Mutlu. CoNDA: Efficient Cache Coherence Support for Near-Data Accelerators. In *ISCA*, 2019.

[54] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Nastaran Hajinazar, Kevin Hsieh, Krishna Malladi, Hongzhong Zheng, and Onur Mutlu. LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory. *CAL*, 2017.

[55] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Nastaran Hajinazar, Kevin Hsieh, Krishna Malladi, Hongzhong Zheng, and Onur Mutlu. LazyPIM: An Efficient Cache Coherence Mechanism for Processing-in-Memory. *arXiv:1706.03162 [cs:AR]*, 2017.

[56] Benjamin C Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. Phase-change technology and the future of main memory. In *IEEE MICRO, 2010*.

[57] Michel Cekleov and Michel Dubois. Virtual-Address Caches Part 1: Problems and Solutions in Uniprocessors. *IEEE Micro*, 1997.

[58] Michel Cekleov and Michel Dubois. Virtual-Address Caches Part 2: Multiprocessor Issues. *IEEE Micro*, 1997.

[59] J. Morris Chang and Edward F. Gehringer. A High-Performance Memory Allocator for Object-Oriented Systems. *TC*, 1996.

[60] J. Morris Chang, Witawas Srisa-An, and C-TD Lo. Architectural Support for Dynamic Memory Management. In *ICCD*, 2000.

[61] Kevin Chang. *Understanding and Improving the Latency of DRAM-Based Memory Systems*. PhD thesis, Carnegie Mellon University, 2017.

[62] Kevin Chang, Rachata Ausavarungnirun, Chris Fallin, and Onur Mutlu. HAT: Heterogeneous Adaptive Throttling for On-Chip Networks. In *SBAC-PAD*, 2012.

[63] Kevin Chang, A. Giray Yaglikci, Saugata Ghose, Aditya Agrawal, Niladrish Chatterjee, Abhijith Kashyap, Donghyuk Lee, Mike O'Connor, Hasan Hassan, and Mutlu Onur. Understanding Reduced-Voltage Operation in Modern DRAM Devices: Experimental Characterization, Analysis, and Mechanisms. In *SIGMETRICS*, 2017.

[64] Kevin K. Chang, Abhijith Kashyap, Hasan Hassan, Saugata Ghose, Kevin Hsieh, Donghyuk Lee, Tianshi Li, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization. In *SIGMETRICS*, 2016.

[65] Kevin K. Chang, Donghyuk Lee, Zeshan Chishti, Alaa R Alameldeen, Chris Wilkerson, Yoongu Kim, and Onur Mutlu. Improving DRAM Performance by Parallelizing Refreshes with Accesses. In *HPCA*, 2014.

[66] Kevin K. Chang, Prashant J Nair, Donghyuk Lee, Saugata Ghose, Moinuddin K Qureshi, and Onur Mutlu. Low-Cost Inter-Linked Subarrays (LISA): Enabling Fast Inter-Subarray Data Movement in DRAM. In *HPCA*, 2016.

[67] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and Protection in a Single-Address-Space Operating System. *TOCS*, 1994.

[68] Jeffrey S. Chase, Henry M. Levy, Edward D. Lazowska, and Miche Baker-Harvey. Lightweight Shared Objects in a 64-bit Operating System. In *OOPSLA*, 1992.

[69] Licheng Chen, Yanan Wang, Zehan Cui, Yongbing Huang, Yungang Bao, and Mingyu Che. Scattered Superpage: A Case for Bridging the Gap Between Superpage and Page Coloring. In *ICCD*, 2013.

[70] John Cheng, Max Grossman, and Ty McKercher. *Professional CUDA C Programming*. John Wiley & Sons, 2014.

[71] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. In *ISCA*, 2016.

[72] Guilherme Cox and Abhishek Bhattacharjee. Efficient Address Translation for Architectures with Multiple Page Sizes. In *ASPLOS*, 2017.

[73] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, G. Sun, Yongpan Liu, Y. Wang, Yuan Xie, and H. Yang. GraphH: A Processing-in-Memory Architecture for Large-Scale Graph Processing. In *IEEE TCAD*, 2018.

[74] Robert C. Daley and Jack B. Dennis. Virtual Memory, Processes, and Sharing in MULTICS. *Communications of the ACM*, 1968.

[75] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *SOSP*, 2007.

[76] Li Deng. The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]. *IEEE Signal Processing Magazine*, 2012.

[77] Quan Deng, Lei Jiang, Youtao Zhang, Minxuan Zhang, and Jun Yang. DrAcc: A DRAM Based Accelerator for Accurate CNN Inference. In *DAC*, 2018.

[78] Peter J Denning. The Working Set Model for Program Behavior. In *Communications of the ACM*, 1968.

[79] Peter J. Denning. Virtual Memory. *CSUR*, 1970.

[80] Fabrice Devaux. The True Processing in Memory Accelerator. In *Hot Chips Symposium (HCS)*, 2019.

[81] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. PDRAM: A Hybrid PRAM and DRAM Main Memory System. In *DAC*, 2009.

[82] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. An Efficient and Scalable Semiconductor Architecture for Parallel Automata Processing. *TPDS*, 2014.

[83] Mario Drumond, Alexandros Daglis, Nooshin Mirzadeh, Dmitrii Ustiugov, Javier Picorel, Babak Falsafi, Boris Grot, and Dionisios Pnevmatikatos. The Mondrian Data Engine. In *ISCA*, 2017.

[84] Yu Du, Miao Zhou, Bruce R Childers, Daniel Mossé, and Rami Melhem. Supporting Superpages in Non-Contiguous Physical Memory. In *HPCA*, 2015.

[85] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaauw, and Reetuparna Das. Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks. In *ISCA*, 2018.

[86] Izzat El Hajj, Alexander Merritt, Gerd Zellweger, Dejan Milojicic, Reto Achermann, Paolo Faraboschi, Wen-mei Hwu, Timothy Roscoe, and Karsten Schwan. SpaceJMP: Programming with Multiple Virtual Address Spaces. In *ASPLOS*, 2016.

[87] D. R. Engler, S. K. Gupta, and M. F. Kaashoek. AVM: Application-Level Virtual Memory. In *HotOS*, 1995.

[88] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *SOSP*, 1995.

[89] Albert Esteve, Maria Engracia Gómez, and Antonio Robles. Exploiting Parallelization on Address Translation: Shared Page Walk Cache. In *OMHI*, 2014.

[90] Facebook, Inc. RocksDB: A Persistent Key–Value Store. `https://rocksdb.org/`.

[91] Joy Fan. Nested Virtualization in Azure. `https://azure.microsoft.com/en-us/blog/nested-virtualization-in-azure/`.

[92] Amin Farmahini-Farahani, Jung Ho Ahn, Katherine Morrow, and Nam Sung Kim. NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules. In *HPCA*, 2015.

[93] Ivan Fernandez, Ricardo Quislant, Eladio Gutiérrez, Oscar Plata, Christina Giannoula, Mohammed Alser, Juan Gómez-Luna, and Onur Mutlu. NATSA: A Near-Data Processing Accelerator for Time Series Analysis. In *ICCD*, 2020.

[94] João Dinis Ferreira, Gabriel Falcao, Juan Gómez-Luna, Mohammed Alser, Lois Orosa, Mohammad Sadrosadati, Jeremie S. Kim, Geraldo F. Oliveira, Taha Shahroodi, Anant Nori, and Onur Mutlu. pLUTo: In-DRAM Lookup Tables to Enable Massively Parallel General-Purpose Computation. In *arXiv:2104.07699 [cs.AR]*, 2021.

[95] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency, 2008. white paper.

[96] Brad Fitzpatrick. Distributed Caching with Memcached. *Linux J.*, 2004.

[97] James D Foley, Foley Dan Van, Andries Van Dam, Steven K Feiner, John F Hughes, Edward Angel, and J Hughes. *Computer Graphics: Principles and Practice*. 1996.

[98] John Fotheringham. Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store. *CACM*, 1961.

[99] Free Software Foundation. GNU Project: Auto-Vectorization in GCC. `https://gcc.gnu.org/projects/tree-ssa/vectorization.html`.

[100] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The Elements of Statistical Learning (2nd Edition)*. Springer-Verlag, 2008.

[101] Pietro Frigo, Emanuele Vannacci, Hasan Hassan, Victor van der Veen, Onur Mutlu, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TRRespass: Exploiting the Many Sides of Target Row Refresh. In *IEEE S&P*, 2020.

[102] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. In-Memory Data Parallel Processor. In *ASPLOS*, 2018.

[103] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. Duality Cache for Data Parallel Acceleration. In *ISCA*, 2019.

[104] Pierre-Emmanuel Gaillardon, Luca Amarú, Anne Siemon, Eike Linn, Rainer Waser, Anupam Chattopadhyay, and Giovanni De Micheli. The Programmable Logic-in-Memory (PLiM) Computer. In *DATE*, 2016.

[105] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Efficient Memory Virtualization: Reducing Dimensionality of Nested Page Walks. In *MICRO*, 2014.

[106] Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. Agile Paging: Exceeding the Best of Nested and Shadow Paging. In *ISCA*, 2016.

[107] Jayneel Gandhi, Vasileios Karakostas, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Unsal. Range Translations for Fast Virtual Memory. *IEEE Micro*, 2016.

[108] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. ComputeDRAM: In-Memory Compute Using Off-the-Shelf DRAMs. In *MICRO*, 2019.

[109] Mingyu Gao and Christos Kozyrakis. HRL: Efficient and Flexible Reconfigurable Logic for Near-Data Processing. In *HPCA*, 2016.

[110] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. TETRIS: Scalable and Efficient Neural Network Acceleration with 3D Memory. In *ASPLOS*, 2017.

[111] Simon Gerber, Gerd Zellweger, Reto Achermann, Kornilios Kourtis, Timothy Roscoe, and Dejan Milojicic. Not Your Parents' Physical Address Space. In *HotOS*, 2015.

[112] Saugata Ghose, Amirali Boroumand, Jeremie S Kim, Juan Gómez-Luna, and Onur Mutlu. Processing-in-Memory: A Workload-Driven Perspective. *IBM JRD*, 2019.

[113] Saugata Ghose, Kevin Hsieh, Amirali Boroumand, Rachata Ausavarungnirun, and Onur Mutlu. The Processing-in-Memory Paradigm: Mechanisms to Enable Adoption. In *Beyond-CMOS Technologies for Next Generation Computer Design*. Springer, 2019. preprint available at arXiv:1802.00320 [cs.AR].

[114] Saugata Ghose, Tianshi Li, Nastaran Hajinazar, Damla Senol Cali, and Onur Mutlu. Demystifying complex workload-DRAM interactions: An experimental study. In *SIGMETRICS*, 2019.

[115] Saugata Ghose, Tianshi Li, Nastaran Hajinazar, Damla Senol Cali, and Onur Mutlu. Understanding the Interactions of Workloads and DRAM Types: A Comprehensive Experimental Study. In *arXiv preprint arXiv:1902.07609*, 2019.

[116] Christina Giannoula, Nandita Vijaykumar, Nikela Papadopoulou, Vasileios Karakostas, Ivan Fernandez, Juan Gómez-Luna, Lois Orosa, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. SynCron: Efficient Synchronization Support for Near-Data-Processing Architectures. In *HPCA*, 2021.

[117] Maya Gokhale, Bill Holmes, and Ken Iobst. Processing in Memory: The Terasys Massively Parallel PIM Array. *Computer*, 1995.

[118] Maya Gokhale, Scott Lloyd, and Chris Hajas. Near Memory Data Structure Rearrangement. In *MEMSYS*, 2015.

[119] Rafael C Gonzalez and Richard E Woods. *Digital Image Processing*. Addison-Wesley, 2 edition, 2002.

[120] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[121] Google, Inc. Compute Engine: Enabling Nested Virtualization for VM Instances. `https://cloud.google.com/compute/docs/instances/enable-nested-virtualization-vm-instances`.

[122] Graph 500. Graph 500 Large-Scale Benchmarks. `http://www.graph500.org/`.

[123] Joseph L. Greathouse, Hongyi Xin, Yixin Luo, and Todd Austin. A Case for Unlimited Watchpoints. In *ASPLOS*, 2012.

[124] Peng Gu, Shuangchen Li, Dylan Stow, Russell Barnes, Liu Liu, Yuan Xie, and Eren Kursun. Leveraging 3D Technologies for Hardware Security: Opportunities and Challenges. In *GLSVLSI*, 2016.

[125] Peng Gu, Xinfeng Xie, Yufei Ding, Guoyang Chen, Weifeng Zhang, Dimin Niu, and Yuan Xie. iPIM: Programmable In-Memory Image Processing Accelerator Using Near-Bank Architecture. In *ISCA*, 2020.

[126] Harsh Gugale, N. Gulur, Yashwant Marathe, and L. John. ATTC (@C): Addressable-TLB based Translation Coherence. In *PACT*, 2020.

[127] Qi Guo, Nikolaos Alachiotis, Berkin Akin, Fazle Sadi, Guanglin Xu, Tze Meng Low, Larry Pileggi, James C. Hoe, and Franz Franchetti. 3D-Stacked Memory-Side Acceleration: Accelerator and System Design. In *WoNDP*, 2014.

[128] M. Gupta, V. Sridharan, D. Roberts, A. Prodromou, A. Venkat, D. Tullsen, and R. Gupta. Reliability-Aware Data Placement for Heterogeneous Memory Architecture. In *HPCA*, 2018.

[129] Faruk Guvenilir and Yale N. Patt. Tailored Page Sizes. In *ISCA*, 2020.

[130] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring Energy Consumption for Short Code Paths Using RAPL. *SIGMETRICS*, 2012.

[131] Nastaran Hajinazar, Geraldo F. Oliveira, Sven Gregorio, Joao Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gomez-Luna, and Onur Mutlu. SIMDRAM: A Framework for Bit-Serial SIMD Processing using DRAM. In *ASPLOS*, 2021.

[132] Nastaran Hajinazar, Pratyush Patel, Minesh Patel, Konstantinos Kanellopoulos, Saugata Ghose, Rachata Ausavarungnirun, Geraldo F. Oliveira, Jonathan Appavoo, Vivek Seshadri, and Onur Mutlu. The Virtual Block Interface: A Flexible Alternative to the Conventional Virtual Memory Framework. In *ISCA 2020*.

[133] Steven M Hand. Self-Paging in the Nemesis Operating System. In *OSDI*, 1999.

[134] Swapnil Haria, Mark D. Hill, and Michael M. Swift. Devirtualizing Memory in Heterogeneous Systems. In *ASPLOS*, 2018.

[135] Harshad Kasture and Daniel Sanchez. TailBench Benchmark Suite. `http://tailbench.csail.mit.edu/`.

[136] Milad Hashemi, Khubaib, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. Accelerating Dependent Cache Misses with an Enhanced Memory Controller. In *ISCA*, 2016.

[137] Milad Hashemi, Onur Mutlu, and Yale N Patt. Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press.

[138] Hasan Hassan, Minesh Patel, Jeremie S. Kim, A. Giray Yaglikci, Nandita Vijaykumar, Nika Mansourighiasi, Saugata Ghose, , and Onur Mutlu. CROW: A Low-Cost Substrate for Improving DRAM Performance, Energy Efficiency, and Reliability. In *ISCA*, 2019.

[139] Hasan Hassan, Gennady Pekhimenko, Nandita Vijaykumar, Vivek Seshadri, Donghyuk Lee, Oguz Ergin, and Onur Mutlu. ChargeCache: Reducing DRAM Latency by Exploiting Row Access Locality. In *HPCA*, 2016.

[140] Hasan Hassan, Nandita Vijaykumar, Samira Khan, Saugata Ghose, Kevin Chang, Gennady Pekhimenko, Donghyuk Lee, Oguz Ergin, and Onur Mutlu. SoftMC: A Flexible and Practical Open-Source Infrastructure for Enabling Experimental DRAM Studies. In *HPCA*, 2017.

[141] Zhezhi He, Li Yang, Shaahin Angizi, Adnan Siraj Rakin, and Deliang Fan. Sparse BD-Net: A Multiplication-Less DNN with Sparse Binarized Depth-Wise Separable Convolution. *JETC*, 2020.

[142] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi Single-Address-Space Operating System. *SPRE*, 1998.

[143] Hewlett-Packard Company. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual, Third Edition*, 1994.

[144] W Daniel Hillis and Lewis W Tucker. The CM-5 Connection Machine: A Scalable Supercomputer. *CACM*, 1993.

[145] William Daniel Hillis. *The Connection Machine*. PhD thesis, Massachusetts Inst. of Technology, 1988.

[146] Peter Hornyack, Luis Ceze, Steve Gribble, Dan Ports, and Hank Levy. A Study of Virtual Memory Usage and Implications for Large Memory. Technical report, Univ. of Washington, 2013.

[147] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. Transparent Offloading and Mapping (TOM) Enabling Programmer-Transparent Near-Data Processing in GPU Systems. In *ISCA*, 2016.

[148] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K. Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. Accelerating Pointer Chasing in 3D-Stacked Memory: Challenges, Mechanisms, Evaluation. In *ICCD*, 2016.

[149] Jian Huang, Anirudh Badam, Moinuddin K Qureshi, and Karsten Schwan. Unified Address Translation for Memory-Mapped SSDs with FlashMap. In *ISCA*, 2015.

[150] Yu Huang, Long Zheng, Pengcheng Yao, Jieshan Zhao, Xiaofei Liao, Hai Jin, and Jingling Xue. A Heterogeneous PIM Hardware-Software Co-Design for Energy-Efficient Graph Processing. In *IPDPS*, 2020.

[151] Wenqin Huangfu, Xueqi Li, Shuangchen Li, Xing Hu, Peng Gu, and Yuan Xie. MEDAL: Scalable DIMM Based Near Data Processing Accelerator for DNA Seeding Algorithm. In *Int. Symp. on Memory Systems*, MICRO.

[152] Hybrid Memory Cube Consortium. Hybrid Memory Cube Specification Rev. 2.1, 2014.

[153] IEEE. IEEE Standard for Floating-Point Arithmetic. Standard 754-2019, 2019.

[154] Illumina, Inc. Miseq system. https://www.illumina.com/systems/ sequencing-platforms/miseq.html.

[155] Illumina, Inc. Nextseq 2000 system. https://www.illumina.com/ systems/sequencing-platforms/nextseq-1000-2000.html.

[156] Illumina, Inc. Novaseq 6000 system. https://www.illumina.com/ systems/sequencing-platforms/novaseq.html.

[157] Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. FloatPIM: In-Memory Acceleration of Deep Neural Network Training with High Precision. In *ISCA*, 2019.

[158] Intel Corp. 6th Generation Intel Core Processor Family Datasheet. `http://www.intel.com/content/www/us/en/processors/core/`.

[159] Intel Corp. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Vol. 3*, 2016.

[160] Intel Corp. 5-Level Paging and 5-Level EPT. white paper, 2017.

[161] Intel Corp. Intel® 64 and IA-32 Architectures Software Developer's Manual, Vol. 3: System Programming Guide. 2019.

[162] International Business Machines Corp. *PowerPC® Microprocessor Family: The Programming Environments Manual for 32 and 64-bit Microprocessors*, 2005.

[163] International Technology Roadmap for Semiconductors. ITRS Reports. `http://www.itrs2.net/itrs-reports.html`, 2015.

[164] Bruce Jacob and Trevor Mudge. Virtual Memory in Contemporary Microprocessors. *IEEE Micro*, 1998.

[165] JEDEC. High Bandwidth Memory (HBM) DRAM. `https://www.jedec.org/standards-documents/docs/jesd235a`.

[166] JEDEC Solid State Technology Assn. JESD235C: High Bandwidth Memory (HBM) DRAM, January 2020.

[167] Xiaowei Jiang, Niti Madan, Li Zhao, Mike Upton, Ravishankar R. Iyer, Srihari Makineni, Donald Newell, Yan Solihin, and Rajeev Balasubramonian. CHOP: Adaptive Filter-Based DRAM Caching for CMP Server Platforms. In *HPCA*, 2010.

[168] Moinuddin K Qureshi, Dae-Hyun Kim, Samira Khan, Prashant J Nair, and Onur Mutlu. AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015.

[169] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héector M. Brice no, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *SOSP*, 1997.

[170] Brewster A Kahle and W Daniel Hillis. The Connection Machine Model CM-1 Architecture. Technical report, Thinking Machines Corp., 1989.

[171] Gokul B. Kandiraju and Anand Sivasubramaniam. Going the Distance for TLB Prefetching: An Application-driven Study. In *ISCA*, 2002.

[172] Mingu Kang, Min-Sun Keel, Naresh R. Shanbhag, Sean Eilert, and Ken Curewitz. An Energy-Efficient VLSI Architecture for Pattern Recognition via Deep Embedding of Computation in SRAM. In *ICASSP 2014*.

[173] Uksong Kang, Hak-Soo Yu, Churoo Park, Hongzhong Zheng, John Halbert, Kuljit Bains, S Jang, and Joo Sun Choi. Co-Architecting Controllers and DRAM to Enhance DRAM Process Scaling. In *The Memory Forum*, 2014.

[174] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman Ünsal. Redundant Memory Mappings for Fast Access to Large Memories. In *ISCA*, 2015.

[175] Vasileios Karakostas, Jayneel Gandhi, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Unsal. Energy-Efficient Address Translation. In *HPCA*, 2016.

[176] Vasileios Karakostas, Osman S Unsal, Mario Nemirovsky, Adrian Cristal, and Michael Swift. Performance Analysis of the Memory Management Unit Under Scale-Out Workloads. In *IISWC*, 2014.

[177] Stefanos Kaxiras and Alberto Ros. A New Perspective for Efficient Virtual-Cache Coherence. In *ISCA*, 2013.

[178] Samira Khan, Donghyuk Lee, Yoongu Kim, Alaa R. Alameldeen, Chris Wilkerson, and Onur Mutlu. The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study. In *SIGMETRICS*, 2014.

[179] Samira Khan, Chris Wilkerson, Donghyuk Lee, Alaa R. Alameldeen, Donghyuk Lee, and Onur Mutlu. A Case for Memory Content-Based Detection and Mitigation of Data-Dependent Failures in DRAM. In *IEEE CAL*, 2017.

[180] Samira Khan, Chris Wilkerson, Zhe Wang, Alaa R. Alameldeen, Donghyuk Lee, and Onur Mutlu. Detecting and Mitigating Data-dependent DRAM Failures by Exploiting Current Memory Content. In *MICRO*, 2017.

[181] Samira M. Khan, Donghyuk Lee, and Onur Mutlu. PARBOR: An Efficient System-Level Technique to Detect Data-Dependent Failures in DRAM. In *DSN*, 2016.

[182] T. Kilburn, D. J. Howarth, R. B. Payne, and F. H. Sumner. The Manchester University Atlas Operating System Part I: Internal Organization. In *The Computer Journal, Volume 4, Issue 3*, 1961.

[183] Tom Kilburn, Dai BG Edwards, Michael J Lanigan, and Frank H Sumner. One-Level Storage System. *IRE Trans. Electronic Computers*, 1962.

[184] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory. In *ISCA*, 2016.

[185] Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, and Kevin Hsieh. Toward Standardized Near-Data Processing With Unrestricted Data Placement for GPUs. In *SC*, 2017.

[186] Jeremie Kim, Minesh Patel, Hasan Hassan, and Onur Mutlu. Solar-DRAM: Reducing DRAM Access Latency by Exploiting the Variation in Local Bitlines. In *ICCD*, 2018.

[187] Jeremie Kim, Minesh Patel, A. Giray Yağlıkçı, Hasan Hassan, Roknoddin Azizi, Lois Orosa, and Onur Mutlu. Revisiting RowHammer: An Experimental Analysis of Modern DRAM Devices and Mitigation Techniques. In *ISCA*, 2020.

[188] Jeremie S Kim, Minesh Patel, Hasan Hassan, Lois Orosa, and Onur Mutlu. D-RaNGe: Using Commodity DRAM Devices to Generate True Random Numbers with Low Latency and High Throughput. In *HPCA*, 2019.

[189] Jeremie S Kim, Damla Senol Cali, Hongyi Xin, Donghyuk Lee, Saugata Ghose, Mohammed Alser, Hasan Hassan, Oguz Ergin, Can Alkan, and Onur Mutlu. GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies. In *arXiv:1708.04329 [q-bio.GN]*, 2017.

[190] Jeremie S Kim, Damla Senol Cali, Hongyi Xin, Donghyuk Lee, Saugata Ghose, Mohammed Alser, Hasan Hassan, Oguz Ergin, Can Alkan, and Onur Mutlu. GRIM-Filter: Fast Seed Location Filtering in DNA Read Mapping Using Processing-in-Memory Technologies. In *BMC Genomics*, 2018.

[191] Yoongu Kim. *Architectural Techniques to Enhance DRAM Scaling.* PhD thesis, Carnegie Mellon University, 2015.

[192] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, je Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. RowHammer: Reliability Analysis and Security Implications. In *arXiv:1603.00747*, 2015.

[193] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *ISCA*, 2014.

[194] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *MICRO*, 2010.

[195] Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu. A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM. In *ISCA*, 2012.

[196] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A Fast and Extensible DRAM Simulator. *CAL*, 2015.

[197] Jeremie Kim s., Minesh Patel, Hasan Hassan, , and Onur Mutlu. The DRAM Latency PUF: Quickly Evaluating Physical Unclonable Functions by Exploiting the Latency-Reliability Tradeoff in Modern DRAM Devices. In *HPCA*, 2018.

[198] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *SOSP*, 2009.

[199] Kenneth C. Knowlton. A Fast Storage Allocator. *CACM*, 1965.

[200] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the Walkers: Accelerating Index Traversals for In-memory Databases. In *MICRO*, 2013.

[201] Skanda Koppula, Lois Orosa, A. Giray Yaglikci, Roknoddin Azizi, Taha Shahroodi, Konstantinos Kanellopoulos, and Onur Mutlu. EDEN: Enabling Energy-Efficient, High-Performance Deep Neural Network Inference Using Approximate DRAM. In *MICRO*, 2019.

[202] Alex Krizhevsky. Convolutional Deep Belief Networks on CIFAR-10. `https://www.cs.toronto.edu/~kriz/conv-cifar10-aug2010.pdf`, 2010.

[203] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet Classification With Deep Convolutional Neural Networks. In *Advances in neural information processing systems*.

[204] Mohan Kumar Kumar, Steffen Maass, Sanidhya Kashyap, Ján Veselỳ, Zi Yan, Taesoo Kim, Abhishek Bhattacharjee, and Tushar Krishna. Latr: Lazy Translation Coherence. In *ASPLOS*, 2018.

[205] Andreas Kurth, Pirmin Vogel, Andrea Marongiu, and Luca Benini. Scalable and Efficient Virtual Memory Sharing in Heterogeneous SoCs with TLB Prefetching and MMU-Aware DMA Engine. In *ICCD*, 2018.

[206] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. Coordinated and Efficient Huge Page Management with Ingens. In *OSDI*, 2016.

[207] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J Rossbach, and Emmett Witchel. Ingens: Huge Page Support for the OS and Hypervisor. *OSR*, 2017.

[208] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, 2004.

[209] Yann LeCun, L. Bottou, Y. Bengio, and P. Haffner. LeNet-5, Convolutional Neural Networks. `http://yann.lecun.com/exdb/lenet`, 2015.

[210] Yann Lecun, L. D. Jackel, Leon Bottou, Corinna Cartes, John S. Denker, Harris Drucker, Urs Müller, Eduard Säckinger, Patrice Simard, Vladimir Vapnik, and et al. Learning Algorithms For Classification: A Comparison On Handwritten Digit Recognition. In *Neural Networks: The Statistical Mechanics Perspective*, 1995.

[211] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *ISCA*, 2009.

[212] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Phase Change Memory Architecture and the Quest for Scalability. In *Communications of the ACM*, 2010.

[213] Chang Joo Lee, Veynu Narasiman, Onur Mutlu, and Yale N Patt. Improving Memory Bank-Level Parallelism in the Presence of Prefetching. In *MICRO*, 2009.

[214] Donghyuk Lee. *Reducing DRAM Latency at Low Cost by Exploiting Heterogeneity*. PhD thesis, Carnegie Mellon University, 2015.

[215] Donghyuk Lee, Saugata Ghose, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. Simultaneous Multi-Layer Access: Improving 3D-Stacked Memory Bandwidth at Low Cost. *ACM TACO*, 2016.

[216] Donghyuk Lee, Samira Khan, Lavanya Subramanian, Saugata Ghose, Rachata Ausavarungnirun, Gennady Pekhimenko, Vivek Seshadri, and Onur Mutlu. Design-Induced Latency Variation in Modern DRAM Chips: Characterization, Analysis, and Latency Reduction Mechanisms. In *SIGMETRICS*, 2017.

[217] Donghyuk Lee, Yoongu Kim, Gennady Pekhimenko, Samira Khan, Vivek Seshadri, Kevin K. Chang, and Onur Mutlu. Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case. In *HPCA*, 2015.

[218] Donghyuk Lee, Yoongu Kim, Vivek Seshadri, Jamie Liu, Lavanya Subramanian, and Onur Mutlu. Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture. In *HPCA*, 2013.

[219] Donghyuk Lee, Lavanya Subramanian, Rachata Ausavarungnirun, Jongmoo Choi, and Onur Mutlu. Decoupled Direct Memory Access: Isolating CPU and IO Traffic by Leveraging a Dual-Data-Port DRAM. In *PACT*, 2015.

[220] Eojin Lee, Ingab Kang, Sukhan Lee, G Edward Suh, and Jung Ho Ahn. TWiCe: Preventing Row-Hammering by Exploiting Time Window Counters. In *ISCA*, 2019.

[221] Joo Hwan Lee, Jaewoong Sim, and Hyesoon Kim. BSSync: Processing Near Memory for Machine Learning Workloads with Bounded Staleness Consistency Models. In *PACT*, 2015.

[222] Yuchun Lee. Handwritten Digit Recognition Using k-Nearest-Neighbor, Radial-Basis Function, and Backpropagation Neural Networks. *Neural Computation*, 1991.

[223] Marzieh Lenjani, Patricia Gonzalez, Elaheh Sadredini, Shuangchen Li, Yuan Xie, Ameen Akel, Sean Eilert, Mircea R Stan, and Kevin Skadron. Fulcrum: A Simplified Control and Access Mechanism Toward Flexible and Practical In-Situ Accelerators. In *HPCA*, 2020.

[224] Chen Li, Rachata Ausavarungnirun, Christopher J. Rossbach, Youtao Zhang, Onur Mutlu, Yang Guo, and Jun Yang. A Framework for Memory Oversubscription Management in Graphics Processing Units. In *ASPLOS*, 2019.

[225] Dong Li, Zizhong Chen, Panruo Wu, and Jeffrey S. Vetter. Rethinking Algorithm-Based Fault Tolerance With a Cooperative Software-Hardware Approach. In *SC*.

[226] Dong Li, Jeffrey S. Vetter, and Weikuan Yu. Classifying Soft Error Vulnerabilities in Extreme-Scale Scientific Applications Using a Binary Instrumentation Tool. In *SC*.

[227] Shuangchen Li, Alvin Oliver Glova, Xing Hu, Peng Gu, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. SCOPE: A Stochastic Computing Engine for DRAM-Based In-Situ Accelerator. In *MICRO*, 2018.

[228] Shuangchen Li, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. DRISA: A DRAM-Based Reconfigurable In-Situ Accelerator. In *MICRO*, 2017.

[229] Shuangchen Li, Cong Xu, Qiaosha Zou, Jishen Zhao, Yu Lu, and Yuan Xie. Pinatubo: A Processing-in-Memory Architecture for Bulk Bitwise Operations in Emerging Non-Volatile Memories. In *DAC*, 2016.

[230] Wentong Li, Saraju Mohanty, and Krishna Kavi. A Page-Based Hybrid (Software-Hardware) Dynamic Memory Allocator. *CAL*, 2006.

[231] Wentong Li, Mehran Rezaei, Krishna Kavi, Afrin Naz, and Philip Sweany. Feasibility of Decoupling Memory Management from the Execution Pipeline. *J. Syst. Archit.*, 2007.

[232] Yang Li, Saugata Ghose, Jongmoo Choi, Jin Sun, Hui Wang, and Onur Mutlu. Utility-Based Hybrid Memory Management. In *CLUSTER*, 2017.

[233] Yinan Li and Jignesh M Patel. BitWeaving: Fast Scans for Main Memory Data Processing. In *SIGMOD*, 2013.

[234] Yong Li, Rami Melhem, and Alex K. Jones. PS-TLB: Leveraging Page Classification Information for Fast, Scalable and Efficient Translation for Future CMPs. In *TACO*, 2013.

[235] Xiaofan Lin, Cong Zhao, and Wei Pan. Towards Accurate Binary Convolutional Neural Network. In *NIPS*, 2017.

[236] Anders Lindstrom, John Rosenberg, and Alan Dearle. The Grand Unified Theory of Address Spaces. In *HotOS*, 1995.

[237] Jamie Liu, Ben Jaiyen, Yoongu Kim, Chris Wilkerson, and Onur Mutlu. An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms. In *ISCA*, 2013.

[238] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. RAIDR: Retention-Aware Intelligent DRAM Refresh. In *ISCA*, 2012.

[239] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. Concurrent Data Structures for Near-Memory Computing. In *SPAA*, 2017.

[240] Scott Lloyd and Maya Gokhale. In-Memory Data Rearrangement for Irregular, Data-Intensive Computing. In *Computer*, 2015.

[241] Scott Lloyd and Maya Gokhale. Design Space Exploration of Near Memory Accelerators. In *MEMSYS*, 2018.

[242] LLVM Project. Auto-Vectorization in LLVM — LLVM 12 Documentation. `https://llvm.org/docs/Vectorizers.html`.

[243] Gabriel H Loh, Nuwan Jayasena, M Oskin, Mark Nutter, David Roberts, Mitesh Meswani, Dong Ping Zhang, and Mike Ignatowski. A Processing in Memory Taxonomy and a Case for Studying Fixed-Function PIM. In *WoNDP*, 2013.

[244] Bruno Lopes, Rafael Auler, Rodolfo Azevedo, and Edson Borin. ISA Aging: A X86 Case Study. In *WIVOSCA*, 2013.

[245] Bruno Cardoso Lopes, Rafael Auler, Luiz Ramos, Edson Borin, and Rodolfo Azevedo. SHRINK: Reducing the ISA Complexity via Instruction Recycling. In *ISCA*, 2015.

[246] Shih-Lien Lu, Ying-Chen Lin, and Chia-Lin Yang. Improving DRAM Latency with Dynamic Asymmetric Subarray. In *MICRO*, 2015.

[247] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, 2005.

[248] Haocong Luo, Taha Shahroodi, Hasan Hassan, Minesh Patel, A. Giray Yağlıkçı, Jisung Park, Lois Orosa, and Onur Mutlu. CLR-DRAM: A Low-Cost DRAM Architecture Enabling Dynamic Capacity-Latency Trade-Off. In *ISCA*, 2020.

[249] Yixin Luo, Sriram Govindan, Bikash Sharma, Mark Santaniello, Justin Meza, Aman Kansal, Jie Liu, Kushagra Vaid Khessib, Badriddine, and Onur Mutlu. Characterizing Application Memory Error Vulnerability to Optimize Data Center Cost via Heterogeneous-Reliability Memory. In *DSN*, 2014.

[250] Daniel Lustig, Abhishek Bhattacharjee, and Margaret Martonosi. TLB Improvements for Chip Multiprocessors: Inter-Core Cooperative Prefetchers and Shared Last-Level TLBs. In *ACM Transactions on Architecture and Code Optimization*, 2013.

[251] Zhulin Ma, Yujuan Tan, H. Jiang, Zhichao Yan, Duo Liu, Xianzhang Chen, Q. Zhuge, E. Sha, and Chengliang Wang. Unified-TP: A Unified TLB and Page Table Cache Structure for Efficient Address Translation. In *ICCD*, 2020.

[252] Steffen Maass, Mohan Kumar Kumar, Taesoo Kim, Tushar Krishna, and Abhishek Bhattacharjee. ecoTLB: Eventually Consistent TLBs. In *ACM Transactions on Architecture and Code Optimization*, 2020.

[253] Artemiy Margaritov, Dmitrii , Amna Shahab, and Boris Grot. PTEMagnet: Fine-Grained Physical Memory Reservation for Faster Page Walks in Public Clouds. In *ASPLOS*, 2021.

[254] Artemiy Margaritov, Dmitrii Ustiugov, Edouard Bugnion, and Boris Grot. Prefetched Address Translation. In *MICRO*, 2019.

[255] Marshall Kirk McKusick, George Neville-Neil, and Robert N.M. Watson. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley Professional, 2014.

[256] Timothy Merrifield and H. Reza Taheri. Performance Implications of Extended Page Tables on Virtualized x86 Processors. In *VEE*, 2016.

[257] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh. Heterogeneous Memory Architectures: A HW/SW Approach for Mixing Die-Stacked and Off-Package Memories. In *HPCA*, 2015.

[258] Justin Meza, Jichuan Chang, HanBin Yoon, Onur Mutlu, and Parthasarathy Ranganathan. Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management. In *CAL*, 2012.

[259] Justin Meza, Yixin Luo, Samira Khan, Jishen Zhao, Yuan Xie, and Onur Mutlu. A Case for Efficient Hardware/Software Cooperative Management of Storage and Memory. In *WEED*, 2013.

[260] Justin Meza, Qiang Wu, Sanjeev Kumar, and Onur Mutlu. Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field. In *DSN*, 2015.

[261] Micron. Hybrid Memory Cube Second Generation. `http://investors.micron.com/releasedetail.cfm?ReleaseID=828028`.

[262] Micron Technology, Inc. Calculating Memory System Power for DDR3. *Technical Note TN-41-01*, 2015.

[263] Micron Technology, Inc. *2Gb: x4, x8, x16 DDR3 SDRAM Data Sheet*, 2016.

[264] MonetDB B.V. MonetDB Column Store. `https://www.monetdb.org/`.

[265] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. CACTI 6.0: A Tool to Model Large Caches. Technical Report HPL-2009-85, HP Laboratories, 2009.

[266] Onur Mutlu. Memory Scaling: A Systems Architecture Perspective. In *International Memory Workshop (IMW)*, 2013.

[267] Onur Mutlu. Main Memory Scaling: Challenges and Solution Directions. In *More than Moore Technologies for Next Generation Computer Design, pp. 127-153, Springer*, 2015.

[268] Onur Mutlu. Memory Scaling: A Systems Architecture Perspective. In *IMW*, 2015.

[269] Onur Mutlu. The RowHammer Problem and Other Issues We May Face as Memory Becomes Denser. In *DATE*, 2017.

[270] Onur Mutlu. Enabling Computation with Minimal Data Movement: Changing the Computing Paradigm for High Efficiency. In *DAC*, 2019.

[271] Onur Mutlu. Intelligent Architectures for Intelligent Computing Systems. In *DATE*, 2021.

[272] Onur Mutlu, Saugata Ghose, Juan Gomez-Luna, and Rachata Ausavarungnirun. Processing Data Where It Makes Sense: Enabling In-Memory Computation. *MICPRO*, 2019.

[273] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. A Modern Primer on Processing in Memory. In *Emerging Computing: From Devices to Systems - Looking Beyond Moore and Von Neumann*. Springer, 2021.

[274] Onur Mutlu and Jeremie S Kim. RowHammer: A Retrospective. *TCAD*, 2019.

[275] Onur Mutlu and Thomas Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO*, 2007.

[276] Onur Mutlu and Thomas Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems. In *ISCA*, 2008.

[277] Onur Mutlu and Lavanya Subramanian. Research problems and opportunities in memory systems. In *SUPERFRI*.

[278] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In *HPCA*, 2017.

[279] R. Nair, S. Antão, C. Bertolli, P. Bose, J. Brunheroto, Tong Chen, Chen-Yong Cher, Carlos H. A. Costa, J. Doi, C. Evangelinos, B. Fleischer, T. Fox, Diego S. Gallo, Leopold Grinberg, John A. Gunnels, A. Jacob, P. Jacob, H. Jacobson, T. Karkhanis, C. Kim, J. Moreno, John K. O'Brien, M. Ohmacht, Yoonho Park, D. Prener, B. Rosenburg, K. D. Ryu, Olivier Sallenave, M. Serrano, P. Siegl, K. Sugavanam, and Zehra Sura. Active Memory Cube: A processing-in-memory architecture for exascale systems. In *IBM JRD*, 2015.

[280] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, Transparent Operating System Support for Superpages. In *OSDI*, 2002.

[281] Neo4j, Inc. Neo4j Graph Platform. `https://neo4j.com/`.

[282] NIMO Group, Arizona State Univ. Predictive Technology Model. `http://ptm.asu.edu/`, 2012.

[283] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *NSDI*, 2013.

[284] NVIDIA Corp. NVIDIA Management Library (NVML). https://developer.nvidia.com/nvidia-management-library-nvml.

[285] NVIDIA Corp. NVIDIA Titan V. https://www.nvidia.com/en-us/titan/titan-v/.

[286] Ataberk Olgun, Minesh Patel, A. Giray Yağlıkçı, Haocong Luo, Jeremie S. Kim, Nisa Bostancı, Nandita Vijaykumar, Oğuz Ergin, and Onur Mutlu. QUAC-TRNG: High-Throughput True Random Number Generation Using Quadruple Row Activation in Commodity DRAM Chips. In *ISCA*, 2021.

[287] Geraldo F. Oliveira, Saugata Ghose, Juan Gómez-Luna, Amirali Boroumand, Alexis Savery, Sonny Rao, Gwendal Grignou, Rahul Thakur, Eric Shiu, and Onur Mutlu. Extending Memory Capacity in Consumer Devices with Emerging Non-Volatile Memory: An Experimental Study. In *SIGMETRICS*, 2021.

[288] Geraldo F Oliveira, Paulo C Santos, Marco A. Z. Alves, and Luigi Carro. NIM: An HMC-Based Machine for Neuron Computation. In *ARC*, 2017.

[289] Geraldo Francisco Oliveira, Juan Gómez-Luna, Lois Orosa, Saugata Ghose, Nandita Vijaykumar, Ivan Fernandez, Mohammad Sadrosadati, and Onur Mutlu. A New Methodology and Open-Source Benchmark Suite for Evaluating Data Movement Bottlenecks: A Near-Data Processing Case Study. In *SIGMETRICS*, 2021.

[290] OpenACC Organization. The OpenACC®Application Programming Interface, Version 3.1, 2020.

[291] Oracle Corp. TimesTen In-Memory Database. https://www.oracle.com/database/technologies/related/timesten.html.

[292] Lois Orosa, Yaohua Wang, Mohammad Sadrosadati, Jeremie S. Kim, Minesh Patel, Ivan Puddu, Haocong Luo, Kaveh Razavi, Juan Gomez-Luna, Hasan Hassan, Nika Mansouri-Ghiasi, Saugata Ghose, and Onur Mutlu. CODIC: A Low-Cost Substrate for Enabling Custom In-DRAM Functionalities and Optimizations. In *ISCA*, 2021.

[293] Oxford Nanopore Technologies Ltd. Gridion. https://nanoporetech. com/products/-gridion.

[294] Oxford Nanopore Technologies Ltd. Minion. https://nanoporetech. com/products/minion.

[295] Oxford Nanopore Technologies Ltd. Promethion. https://nanoporetech.com/products/promethion.

[296] Misel-Myrto Papadopoulou, Xin Tong, André Seznec, and Andreas Moshovos. Prediction-Based Superpage-Friendly TLB Designs. In *HPCA*, 2015.

[297] Chang Hyun Park, Taekyung Heo, Jungi Jeong, and Jaehyuk Huh. Hybrid TLB Coalescing: Improving TLB Translation Coverage Under Diverse Fragmented Memory Allocations. In *ISCA*, 2017.

[298] Yeonhong Park, Woosuk Kwon, Eojin Lee, Tae Jun Ham, Jung Ho Ahn, and Jae W Lee. Graphene: Strong yet Lightweight Row Hammer Protection. In *MICRO*, 2020.

[299] Minesh Patel, Jeremie Kim, Hasan Hassan, and Onur Mutlu. Understanding and Modeling On-Die Error Correction in Modern DRAM: An Experimental Study Using Real Devices. In *DSN*, 2019.

[300] Minesh Patel, Jeremie Kim, Taha Shahroodi, Hasan Hassan, and Onur Mutlu. Bit-Exact ECC Recovery (BEER): Determining DRAM On-Die ECC Functions by Exploiting DRAM Data Retention Characteristics. In *MICRO*, 2020.

[301] Minesh Patel, Jeremie S Kim, and Onur Mutlu. The Reach Profiler (REAPER): Enabling the Mitigation of DRAM Retention Failures via Profiling at Aggressive Conditions. In *ISCA*, 2017.

[302] Ashutosh Pattnaik, Xulong Tang, Adwait Jog, Onur Kayiran, Asit K Mishra, Mahmut T Kandemir, Onur Mutlu, and Chita R Das. Scheduling Techniques for GPU Architectures with Processing-in-Memory Capabilities. In *PACT*, 2016.

[303] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using SimPoint for Accurate and Efficient Simulation. In *SIGMETRICS*, 2003.

[304] Binh Pham, Abhishek Bhattacharjee, Yasuko Eckert, and Gabriel H Loh. Increasing TLB Reach by Exploiting Clustering in Page Translations. In *HPCA*, 2014.

[305] Binh Pham, Viswanathan Vaidyanathan, Aamer Jaleel, and Abhishek Bhattacharjee. CoLT: Coalesced Large-Reach TLBs. In *MICRO*, 2012.

[306] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. Large Pages and Lightweight Memory Management in Virtualized Environments: Can You Have It Both Ways? In *MICRO*, 2015.

[307] Binh Pham, Jan Vesely, Gabriel H Loh, and Abhishek Bhattacharjee. Using TLB Speculation to Overcome Page Splintering in Virtual Machines. Technical Report DCS-TR-713, Rutgers Univ., 2015.

[308] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *ASPLOS*, 2014.

[309] Javier Picorel, Djordje Jevdjic, and Babak Falsafi. Near-Memory Address Translation. In *PACT*, 2017.

[310] Massimiliano Poletto and Vivek Sarkar. Linear Scan Register Allocation. *TOPLAS*, 1999.

[311] B. Pourshirazi and Z. Zhu. Refree: A Refresh-Free Hybrid DRAM/PCM Main Memory System. In *IPDPS*, 2016.

[312] Jason Power, Mark D. Hill, and David A. Wood. Supporting x86-64 Address Translation for 100s of GPU Lanes. In *HPCA*, 2014.

[313] A. Prodromou, M. Meswani, N. Jayasena, G. Loh, and D. M. Tullsen. MemPod: A Clustered Architecture for Efficient and Scalable Migration in Flat Address Space Multi-Level Memories. In *HPCA*, 2017.

[314] Seth H Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. NDC: Analyzing the Impact of 3D-Stacked Memory+Logic Devices on MapReduce Workloads. In *ISPASS*, 2014.

[315] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable High Performance Main Memory System Using Phase-Change Memory Technology. In *ISCA*, 2009.

[316] Rambus Inc. Rambus Power Model. `https://www.rambus.com/energy/`.

[317] Luiz Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page Placement in Hybrid Memory Systems. In *ICS*, 2011.

[318] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Y-C Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, S-H Chen, H-L Lung, and C-H Lam. Phase-Change Random Access Memory: A Scalable Technology. *IBM Journal of Research and Development*, 2008.

[319] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In *ECCV*, 2016.

[320] D. M. Ritchie and Ken Thompson. The UNIX Time-Sharing System. *The Bell System Technical Journal*, 1978.

[321] Bogdan F. Romanescu, A. Lebeck, D. Sorin, and Anne Bracy. UNified Instruction/-Translation/Data (UNITD) Coherence: One Protocol to Rule Them All. In *HPCA*, 2010.

[322] J. H. Ryoo, M. R. Meswani, A. Prodromou, and L. K. John. SILC-FM: Subblocked InterLeaved Cache-Like Flat Memory Organization. In *HPCA*, 2017.

[323] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K. John. Rethinking TLB Designs in Virtualized Environments: A Very Large Part-of-Memory TLB. In *ISCA*, 2017.

[324] Jeremie S Kim, Can Firtina, Damla Senol Cali, Mohammed Alser, Nastaran Hajinazar, Can Alkan, and Onur Mutlu. AirLift: A Fast and Comprehensive Technique for Translating Alignments between Reference Genomes. In *arXiv preprint arXiv:1912.08735*, 2019.

[325] SAFARI Research Group. Ramulator-VBI — GitHub Repository. `https://github.com/CMU-SAFARI/Ramulator-VBI.git`.

[326] Paulo C Santos, Geraldo F Oliveira, João P Lima, Marco A. Z. Alves, Luigi Carro, and Antonio C. S. Beck. Processing in 3D Memories to Speed Up Operations on Complex Data Structures. In *DATE*, 2018.

[327] Paulo C Santos, Geraldo F Oliveira, Diego G Tomé, Marco A. Z. Alves, Eduardo C Almeida, and Luigi Carro. Operand Size Reconfiguration for Big Data Processing in Memory. In *DATE*, 2017.

[328] SAP SE. SAP HANA: In-Memory Data Platform. `https://www.sap.com/products/hana.html`.

[329] Ashley Saulsbury, Fredrik Dahlgren, and Per Stenström. Recency-based TLB preloading. In *ISCA*, 2000.

[330] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. EbbRT: A Framework for Building Per-Application Library Operating Systems. In *OSDI*, 2016.

[331] Damla Senol Cali, Gurpreet S Kalsi, Zülal Bingöl, Can Firtina, Lavanya Subramanian, Jeremie S Kim, Rachata Ausavarungnirun, Mohammed Alser, Juan Gómez-Luna, Amirali Boroumand, Anant Nori, Allison Scibisz, Sreenivas Subramoney, Can Alkan, Saugata Ghose, and Onur Mutlu. GenASM: A High-Performance, Low-Power Approximate String Matching Acceleration Framework for Genome Sequence Analysis. In *MICRO*, 2020.

[332] Damla Senol Cali, Jeremie Kim, Saugata Ghose, Can Alkan, and Onur Mutlu. Nanopore Sequencing Technology and Tools for Genome Assembly: Computational Analysis of the Current State, Bottlenecks and Future Directions. In *Briefings in Bioinformatics*, 2018.

[333] Vivek Seshadri. *Simple DRAM and Virtual Memory Abstractions to Enable Highly Efficient Memory Systems*. PhD thesis, Carnegie Mellon University, 2016.

[334] Vivek Seshadri, Abhishek Bhowmick, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. The Dirty-Block Index. In *ISCA*, 2014.

[335] Vivek Seshadri, Kevin Hsieh, Amirali Boroumabd, Donghyuk Lee, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd C Mowry. Fast Bulk Bitwise AND and OR in DRAM. *CAL*, 2015.

[336] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and T. C. Mowry. RowClone: Fast and Energy-Efficient in-DRAM Bulk Data Copy and Initialization. In *MICRO*, 2013.

[337] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd C Mowry. Buddy-RAM: Improving the Performance and Efficiency of Bulk Bitwise Operations Using DRAM. In *arXiv:1611.09988*, 2016.

[338] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd C Mowry. Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology. In *MICRO*, 2017.

[339] Vivek Seshadri, Thomas Mullins, Amirali Boroumand, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. Gather-Scatter DRAM: in-DRAM Address Translation to Improve the Spatial Locality of Non-unit Strided Accesses. In *MICRO*, 2015.

[340] Vivek Seshadri and Onur Mutlu. The Processing Using Memory Paradigm: In-DRAM Bulk Copy, Initialization, Bitwise AND and OR. arXiv:1610.09603 [cs.AR], 2016.

[341] Vivek Seshadri and Onur Mutlu. Simple Operations in Memory to Reduce Data Movement. In *Advances in Computers*, volume 106. 2017.

[342] Vivek Seshadri and Onur Mutlu. In-DRAM Bulk Bitwise Execution Engine. arXiv:1905.09822 [cs.AR], 2019.

[343] Vivek Seshadri and Onur Mutlu. In-DRAM Bulk Bitwise Execution Engine. In *Invited Book Chapter in Advances in Computers*, 2020.

[344] Vivek Seshadri, Gennady Pekhimenko, Olatunji Ruwase, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, Todd C. Mowry, and Trishul Chilimbi. Page Overlays: An Enhanced Virtual Memory Framework to Enable Fine-Grained Memory Management. In *ISCA*, 2015.

[345] Seyyed Hossein SeyyedAghaei Rezaei, Mehdi Modarressi, Rachata Ausavarungnirun, Mohammad Sadrosadati, Onur Mutlu, and Masoud Daneshtalab. NoM: Network-on-Memory for Inter-Bank Data Transfer in Highly-Banked Memories. *CAL*, 2020.

[346] Ofer Shacham, Omid Azizi, Megan Wachs, Wajahat Qadeer, Zain Asgar, Kyle Kelley, John P Stevenson, Stephen Richardson, Mark Horowitz, Benjamin Lee, Alex Solomatnikov, and Amin Firoozshahian. Rethinking digital design: Why design must change. In *MICRO*, 2010.

[347] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *ISCA*, 2016.

[348] Kenneth K. Shen and James L. Peterson. A Weighted Buddy Method for Dynamic Storage Allocation. *CACM*, 1974.

[349] Seunghee Shin, Guilherme Cox, Mark Oskin, Gabriel H. Loh, Yan Solihin, Abhishek Bhattacharjee, and Arkaprava Basu. Scheduling Page Table Walks for Irregular GPU Applications. In *ISCA*, 2018.

[350] William Shooman. Parallel Computing with Vertical Data. In *EJCC*, 1960.

[351] Jaewoong Sim, Alaa R. Alameldeenand, Zeshan Chishti, Chris Wilkerson, and Hyesoon Kim. Transparent Hardware Management of Stacked DRAM as Part of Memory. In *MICRO*, 2014.

[352] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:1409.1556 [cs.CV], 2014.

[353] Gagandeep Singh, Mohammed Alser, Damla Senol Cali, Dionysios Diamantopoulos, Juan Gómez-Luna, Henk Corporaal, and Onur Mutlu. FPGA-based Near-Memory Acceleration of Modern Data-Intensive Applications. In *IEEE MICRO*, 2021.

[354] Gagandeep Singh, Dionysios Diamantopoulos, Christoph Hagleitner, Juan Gómez-Luna, Sander Stuijk, Onur Mutlu, and Henk Corporaal. NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling. In *FPL*, 2020.

[355] Gagandeep Singh, Juan Gomez-Luna, Giovanni Mariani, Geraldo F. Oliveira, Stefano Corda, Sander Stujik, Onur Mutlu, and Henk Corporaal. NAPEL: Near-Memory Computing Application Performance Prediction via Ensemble Learning. In *DAC*, 2019.

[356] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism. In *ASPLOS*, 2020.

[357] Mathias Soeken, Saeideh Shirinzadeh, Pierre-Emmanuel Gaillardon, Luca Gaetano Amarú, Rolf Drechsler, and Giovanni De Micheli. An MIG-Based Compiler for Programmable Logic-in-Memory Architectures. In *DAC*, 2016.

[358] Young Hoon Son, O. Seongil, Yuhwan Ro, Jae W. Lee, and Jung Ho Ahn. Reducing Memory Access Latency with Asymmetric DRAM Bank Organizations. In *ISCA*, 2013.

[359] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning. In *HPCA*, 2017.

[360] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. GraphR: Accelerating Graph Processing Using ReRAM. In *HPCA*, 2018.

[361] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N Patt. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In *HPCA*, 2007.

[362] Standard Performance Evaluation Corp. SPEC CPU® 2006. `https://www.spec.org/cpu2006/`.

[363] Standard Performance Evaluation Corp. SPEC CPU® 2017 Benchmark Suite. `https://www.spec.org/cpu2017/`.

[364] Arun Subramaniyan and Reetuparna Das. Parallel Automata Processor. In *ISCA*, 2017.

[365] Zehra Sura, Arpith Jacob, Tong Chen, Bryan Rosenburg, Olivier Sallenave, Carlo Bertolli, Samuel Antao, Jose Brunheroto, Yoonho Park, Kevin O'Brien, and Ravi Nair. Data Access Optimization in a Processing-in-Memory System. In *CF*.

[366] Madhusudhan Talluri, M. Hill, and Y. Khalidi. A new Page Table for 64-bit Address Spaces. In *SOSP*, 1995.

[367] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB Performance of Superpages with Less Operating System Support. In *ASPLOS*, 1994.

[368] P.J. Teller. Translation-lookaside buffer consistency. In *Computer (Volume: 23, Issue: 6)*, 1990.

[369] P.J. Teller, R. Kenner, and M. Snir. TLB consistency on highly-parallel shared-memory multiprocessors. In *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences. Volume I: Architecture Track*, 1988.

[370] The Open Group. The Single UNIX Specification, Version 2. `https://pubs.opengroup.org/onlinepubs/7908799/xsh/systime.h.html`, 1997.

[371] Mohit Tiwari, Banit Agrawal, Shashidhar Mysore, Jonathan Valamehr, and Timothy Sherwood. A Small Cache of Large Ranges: Hardware Methods for Efficiently Searching, Storing, and Updating Big Dataflow Tags. In *MICRO*, 2008.

[372] Transaction Processing Performance Council. TPC-H. `http://www.tpc.org/tpch/`.

[373] Lewis W Tucker and George G Robertson. Architecture and Applications of the Connection Machine. *Computer*, 1988.

[374] Alexey Tumanov, Wise Joshua, and R. Ganger Gregory Mutlu Onur. Asymmetry-Aware Execution Placement on Manycore Chips. In *SFMA*, 2013.

[375] Nandita Vijaykumar, Eiman Ebrahimi, Kevin Hsieh, Phillip B Gibbons, and Onur Mutlu. The Locality Descriptor: A Holistic Cross-Layer Abstraction to Express Data Locality in GPUs. In *ISCA*, 2018.

[376] Nandita Vijaykumar, Kevin Hsieh, Gennady Pekhimenko, Samira Khan, Ashish Shrestha, Saugata Ghose, Adwait Jog, Phillip B. Gibbons, and Onur Mutlu. Zorua: A Holistic Approach to Resource Virtualization in GPUs. In *MICRO*, 2016.

[377] Nandita Vijaykumar, Abhilasha Jain, Diptesh Majumdar, Kevin Hsieh, Gennady Pekhimenko, Eiman Ebrahimi, Nastaran Hajinazar, Phillip B. Gibbons, and Onur Mutlu. A Case for Richer Cross-Layer Abstractions: Bridging the Semantic Gap with Expressive Memory. In *ISCA*, 2018.

[378] Nandita Vijaykumar, Gennady Pekhimenko, Adwait Jog, Abhishek Bhowmick, Rachata Ausavarungnirun, Chita Das, Mahmut Kandemir, Todd C. Mowry, and Onur Mutlu. A case for Core-Assisted Bottleneck Acceleration in GPUs: Enabling flexible data compression with assist warps. In *ISCA*, 2015.

[379] Thomas Vogelsang. Understanding the Energy Consumption of Dynamic Random Access Memories. In *MICRO*, 2010.

[380] Yaohua Wang, Lois Orosa, Xiangjun Peng, Yang Guo, Saugata Ghose, Minesh Patel, Jeremie S Kim, Juan Gómez Luna, Mohammad Sadrosadati, Nika Mansouri Ghiasi, and Onur Mutlu. FIGARO: Improving System Performance via Fine-Grained In-DRAM Data Relocation and Caching. In *MICRO*, 2020.

[381] Yaohua Wang, Arash Tavakkol, Lois Orosa, Saugata Ghose, Nika Mansouri Ghiasi, Minesh Patel, Jeremie Kim, Hasan Hassan, Mohammad Sadrosadati, and Onur Mutlu. Reducing DRAM Latency via Charge-Level-Aware Look-Ahead Partial Restoration. In *MICRO*, 2018.

[382] David Wentzlaff and Anant Agarwal. Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores. *OSR*, 2009.

[383] Emmett Witchel. *Mondriaan Memory Protection*. PhD thesis, Massachusetts Inst. of Technology, 2004.

[384] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian Memory Protection. In *ASPLOS*, 2002.

[385] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, and J. M. Pendleton. An In-Cache Address Translation Mechanism. In *ISCA*, 1986.

[386] Xingbo Wu, Fan Ni, and Song Jiang. Search Lookaside Buffer: Efficient Caching for Index Data Structures. In *SoCC*, 2017.

[387] Hongyi Xin, Donghyuk Lee, Farhad Hormozdiari, Samihan Yedkar, Onur Mutlu, and Can Alkan. Accelerating Read Mapping with FastHASH. In *BMC Genomics*, 2013.

[388] Hongyi Xin, Sunny Nahar, Richard Zhu, John Emmons, Gennady Pekhimenko, Carl Kingsford, and Onur Mutlu. Optimal Seed Solver: Optimizing Seed Selection in Read Mapping. In *Bioinformatics Journal*, 2015.

[389] Xin Xin, Youtao Zhang, and Jun Yang. ELP2IM: Efficient and Low Power Bitwise Operation Processing in DRAM. In *HPCA*, 2020.

[390] A Giray Yağlıkçı, Minesh Patel, Jeremie S. Kim, Roknoddin Azizibarzoki, Ataberk Olgun, Lois Orosa, Hasan Hassan, Jisung Park, Konstantinos Kanellopoullos, Taha Shahroodi, Saugata Ghose, and Onur Mutlu. BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows. In *HPCA*, 2021.

[391] Zi Yan, Ján Veselỳ, Guilherme Cox, and Abhishek Bhattacharjee. Hardware Translation Coherence for Virtualized Systems. In *ISCA*, 2017.

[392] HanBin Yoon, Justin Meza, Rachata Ausavarungnirun, Rachael A Harding, and Onur Mutlu. Row Buffer Locality Aware Caching Policies for Hybrid Memories. In *ICCD*, 2012.

[393] HanBin Yoon, Justin Meza, Naveen Muralimanohar, Norman P. Jouppi, and Onur Mutlu. Efficient Data Mapping and Buffering Techniques for Multi-Level Cell Phase-Change Memories. In *TACO*, 2014.

[394] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, Onur Mutlu, and Srinivas Devadas. Banshee: Bandwidth-Efficient DRAM Caching via Software/Hardware Cooperation. In *MICRO*, 2017.

[395] Zihao Yu, Bowen Huang, Jiuyue Ma, Ninghui Sun, and Yungang Bao. Labeled RISC-V: A New Perspective on Software-Defined Architecture. In *CARRV*, 2017.

[396] Yue Zha and Jing Li. Hyper-AP: Enhancing Associative Processing Through A Full-Stack Optimization. In *ISCA*, 2020.

[397] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L Greathouse, Lifan Xu, and Michael Ignatowski. TOP-PIM: Throughput-Oriented Programmable Processing in Memory. In *HPDC*, 2014.

[398] Lixin Zhang, Evan Speight, Ram Rajamony, and Jiang Lin. Enigma: Architectural and Operating System Support for Reducing the Impact of Address Translation. In *ICS*, 2010.

[399] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition. In *HPCA*, 2018.

[400] Wangyuan Zhang and Tao Li. Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures. In *PACT*, 2009.

[401] Tianhao Zheng, Haishan Zhu, and Mattan Erez. SIPT: Speculatively Indexed, Physically Tagged Caches. In *HPCA*, 2018.

[402] Q. Zhu, T. Graf, H. E. Sumbul, L. Pileggi, and F. Franchetti. Accelerating Sparse Matrix-Matrix Multiplication with 3D-Stacked Logic-in-Memory Hardware. In *HPEC*, 2013.

[403] Youwei Zhuo, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and aXuehai Qian. GraphQ: Scalable PIM-Based Graph Processing. In *MICRO*, 2019.

[404] W.K. Zuravleff and T. Robinson. Controller for a Synchronous DRAM That Maximizes Throughput by Allowing Memory Requests and Commands to Be Issued Out of Order. U.S. Patent No. 5,630,096, 1997.

Appendix A

AIG-to-MIG Conversion

The conversion from AND/OR/NOT representation of an operation to its MAJ/NOT representation relies on a set of transformation rules that are derived from the characteristics of the MAJ operation. Table A.1 lists the set of transformation rules that we use to synthesize a circuit for a desired operation with MAJ and NOT gates. We use full addition as a running example to describe the process of synthesizing a MAJ/NOT-based circuit, starting from an AND/OR/NOT representation of the circuit and using the transformation rules. We obtain MAJ/NOT-based circuits for other SIMDRAM operations following the same method. In a later step (section 2.3.2), we translate a MAJ/NOT-based circuit to sequences of `AAP`s/`AP`s operations.

Table A.1: MAJ/NOT transformation Rules [16].

Commutativity (C)	$M(x, y, z) = M(y, x, z) = M(z, y, x)$
Majority (M)	$\text{if}(x = y) : M(x, y, z) = x = y$ $\text{if}(x = \overline{y}) : M(x, y, z) = z$
Associativity (A)	$M(x, u, M(y, u, z)) = M(z, u, M(y, u, x))$
Distributivity (D)	$M(x, y, M(u, v, z)) = M(M(x, y, u), M(x, y, v), z)$
Inverter Propagation (I)	$\overline{M}(x, y, z) = M(\overline{x}, \overline{y}, \overline{z})$
Relevance (R)	$M(x, y, z) = M(x, y, \mathbf{z}_{x/\overline{y}})$
Complementary Associativity (CA)	$M(x, u, M(y, \overline{u}, z)) = M(x, u, M(y, x, z))$

Figure A.1a shows the optimized AND/OR/Inverter (i.e., AND/OR/NOT) Graph (AOIG) representation of a full addition (i.e., F = A + B + C_{in}). As shown in Figure A.1b, the naive way to transform the AOIG to a Majority/Inverter (i.e., MAJ/NOT) Graph (MIG) representation, is to replace every AND and OR primitive with a three-input MAJ primitive where the third input is 0 or 1, respectively. The resulting MIG is in fact Ambit's [338] representation of the full addition. While the AOIG in Figure A.1a is optimized for AND/OR/NOT operations, the resulting MIG in Figure A.1b can be further optimized by exploiting the transformation rules of the MAJ primitive (Table A.1, replicated from [16]). The MIG optimization is performed in two key steps: (1) node reduction, and (2) MIG reshaping.

Node reduction. In order to optimize the MIG in Figure A.1b, the first step is to reduce the number of MAJ nodes in the MIG. As shown in Table 1, rules **M** and **D** reduce the number of nodes in a MIG if applied from left to right (i.e., $\mathbf{M}_{L \rightarrow R}$) and from right to

left (i.e., $\mathbf{D}_{R\rightarrow L}$), respectively. $\mathbf{M}_{L\rightarrow R}$ replaces a MAJ node with a single value, and $\mathbf{D}_{R\rightarrow L}$ replaces three MAJ nodes with two MAJ nodes in the MIG. The node reduction step applies $\mathbf{M}_{L\rightarrow R}$ and $\mathbf{D}_{R\rightarrow L}$ as many times as possible to reduce the the number of MAJ operations in the MIG. We can see in Figure A.1b that none of the two rules are applicable in the particular case of the full addition MIG. Therefore, Fig A.1b remains unchanged after applying node reduction.

MIG reshaping. When no further node reduction is possible, we *reshape* the MIG in an effort to enable more node reduction opportunities by repeatedly using two sets of rules: (1) rules $\mathbf{M}_{R\rightarrow L}$, $\mathbf{D}_{L\rightarrow R}$, and $\mathbf{R}$ to temporarily inflate the MIG and create more node reduction opportunities with the help of the new nodes, and (2) rules $\mathbf{A}$ and $\mathbf{CA}$, to exchange variables between adjacent nodes. Note that in this step, rules $\mathbf{M}$ and $\mathbf{D}$ are applied in the reverse direction compared to the previous step (i.e., node reduction step) which results in increasing the number of nodes in the MIG. We now describe the MIG reshaping process for the full addition example (Figure A.1b). For simplicity, we first assume that the entire MIG is represented as function $\mathbf{F}$ that computes the full addition of the input operands $\mathbf{A}$ and $\mathbf{B}$. Then, we apply rule $\mathbf{M}_{R\rightarrow L}$ while introducing variable $\mathbf{X}$ to the MIG (as $F = M(F, x, \overline{x})$) without impacting the functionality of the MIG (Figure A.1c). We then apply the same rule again, and replace $\mathbf{X}$ with a new MAJ node while introducing variable $\mathbf{Y}$ (Figure A.1d). Next, by applying rule $\mathbf{D}_{L\rightarrow R}$, we introduce a new MAJ node and distribute the function $\mathbf{F}$ across the two MAJ nodes (Figure A.1e). Now, by applying rule $\mathbf{R}$ to the function $\mathbf{F}$ on the left, variable $\overline{\mathbf{X}}$ is replaced with variable $\overline{\mathbf{Y}}$ in the function $\mathbf{F}$ on the left. Similarly, by applying rule $\mathbf{R}$ to the function $\mathbf{F}$ on the right, variable $\overline{\mathbf{X}}$ is replaced with variable $\mathbf{Y}$ in the function $\mathbf{F}$ on the right (Figure A.1f). At this point, since rule $\mathbf{M}_{R\rightarrow L}$ holds with any given two variables, we can safely replace $\mathbf{X}$ and $\mathbf{Y}$ with variables $\mathbf{A}$ and $\mathbf{B}$, respectively (Figure A.1g). Next, we expand function $\mathbf{F}$ (Figure A.1h) and the variables replaced as a result of the previous rule are highlighted in blue. As shown in Figure A.1h, the resulting graph after expanding function $\mathbf{F}$ has multiple node reduction opportunities using rule $\mathbf{M}_{L\rightarrow R}$ and starting from the top of the graph. The nodes that can be eliminated

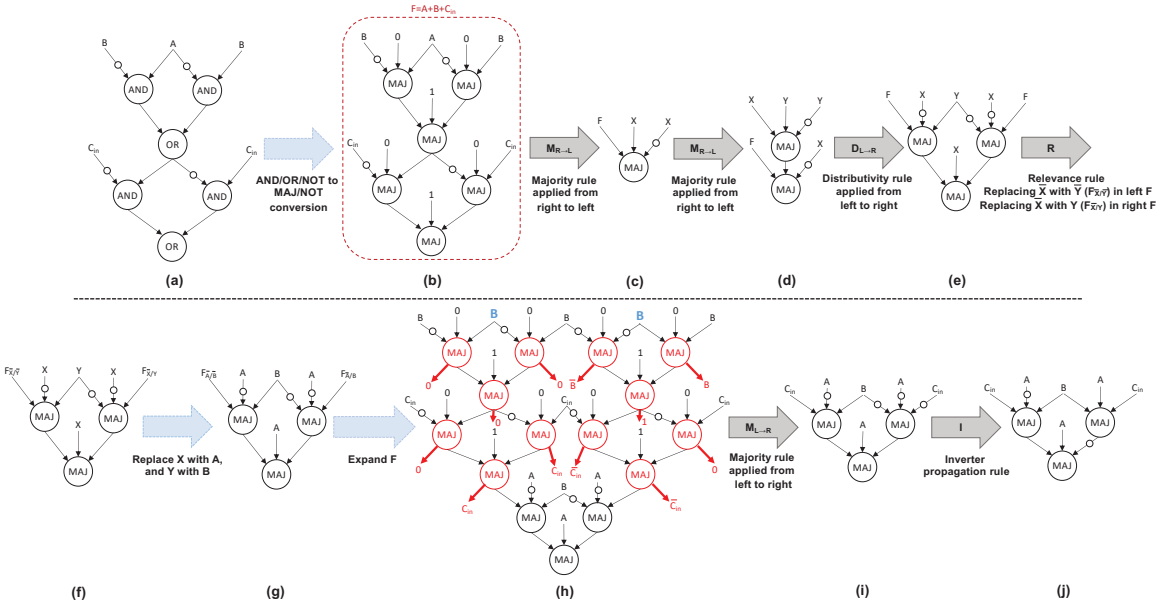


Figure A.1: Synthesizing SIMDRAM circuit for a full addition.

using this rule are marked in red and the replacing value is indicated with a red arrow leaving the node. Figure A.1i shows the same MIG after resolving all the node reductions. We next use rule **I** to remove all three NOT primitives in the rightmost MAJ node. The final optimized MIG that is shown in Figure A.1j requires only 3 MAJ primitives to perform the full addition operation, as opposed to the 6 we started with (in Figure A.1b).

The node reduction step followed by the MIG reshaping step are repeated (for a predefined number of times) until we achieve an optimized MIG that requires minimal number of MAJ operations to perform the desired in-DRAM operation. The process of converting an operation to a MAJ-based implementation can be automated as suggested by prior work [16, 357].

Appendix B

Row-to-Operand Allocation

Algorithm 1 describes SIMDRAM's row-to-operand allocation procedure. To enable in-DRAM computation, our allocation algorithm copies (i.e., maps) input operands for each MAJ node in the MIG from D-group rows (where the operands normally reside) into compute rows. However, due to the limited number of compute rows, the allocation algorithm cannot allocate DRAM rows to all input operands from all MAJ nodes at once. To address this issue, the allocation algorithm divides the allocation process into *phases*. Each phase allocates as many compute rows to operands as possible. For example, because no rows are allocated yet, the initial phase (Phase 0) has all six compute rows available for allocation (i.e., the rows are vacant), and can allocate up to six input operands to the compute rows. A phase is considered finished when either (1) there are not enough vacant compute rows to allocate all input operands for the next logic primitive that needs to be computed, or (2) there are no more MAJ primitives left to process in the MIG. The phase information is used when generating the µProgram for the MIG in Task 2 of Step 2 of SIMDRAM framework (Section 2.3.2), where µOps for all MAJ primitives in phase i are generated prior to the MAJ primitives in phase $i + 1$. Knowing that all the MAJ primitives in phase i are performed before the next phase $i + 1$ starts, the allocation algorithm can safely reuse the compute rows for use in phase $i + 1$, without worrying about the output of a MAJ primitive being overwritten by a new row-to-operand allocation.

We now describe the row-to-operand allocation algorithm in detail, using the MIG for full addition in Figure 2.5a as an example of a MIG being traversed by the algorithm. The allocation algorithm starts at Phase 0. Throughout its execution, the algorithm maintains (1) the list of free compute rows that are available for allocation (*B_rows* and *B_rows_DCC* in Algorithm 1, initialized in lines 3–4); and (2) the list of row-to-operand allocations associated with each MAJ node, tagged with the phase number that the allocations were performed in (*row_operand_allocation* in Algorithm 1). Once a row-to-operand allocation is performed, the algorithm removes the compute row used for the allocation from the list of the free compute rows, and adds the new allocation to the list of row-to-operand allocations generated in that phase for the corresponding MAJ node. The algorithm follows a simple procedure to allocate compute rows to the input operands of the MAJ nodes in the MIG. The algorithm does a topological traversal starting with the leftmost MAJ node in the highest level of the MIG (e.g., Level 0 in Figure 2.5a), and traverses all the MAJ nodes in each level, before moving to the next lower level of the graph.

Algorithm 1 SIMDRAM's Row-to-Operand Allocation Algorithm.

```
1: Input: MIG G = (V, E)                                   ▷ Majority-Inverter Graph G nodes <vertex, edge>
2: Output: row_operand_allocation                          ▷ Allocation map of rows to operands
3: B_rows ← {T0, T1, T2, T3}
4: B_rows_DCC ← {DCC0, DCC1}
5: phase ← 0
6: row_operand_allocation_map ← ∅
7: for each level in G do
8:   for each V in G[level] do
9:     for each input edge in E[V] do
10:      Search for input edge's parent
11:      if input edge has no parents then
12:        if input edge is negated then
13:          Allocate row in B_rows_DCC to input edge
14:          Remove allocated row from B_rows_DCC
15:        else
16:          Allocate row in B_rows to input edge
17:          Remove allocated row from B_rows
18:      else
19:        if  input edge is negated then
20:          Map allocated parent row in B_rows_DCC to input edge
21:        else
22:          Map allocated parent row in B_rows to input edge
23:      if B_rows and B_rows_DCC are empty then
24:        phase ← phase + 1
25:        B_rows ← {T0, T1, T2, T3}
26:        B_rows_DCC ← {DCC0, DCC1}
27:      row_operand_allocation ← (input edge, allocated row, phase)
```

Case 1 (lines 11–17)

Case 2 (lines 18–22)

Case 3 (lines 23–27)

For each of the three input edges (i.e., operands) of any given MAJ node, the algorithm checks for the following three possible cases and performs the allocation accordingly:

Case 1: if the edge is not connected to another MAJ node in a higher level of the graph (line 11 in Algorithm 1), i.e., the edge does not have a parent (e.g., the three edges entering the blue node in Figure 2.5a), and a compute row is available, the input operand associated with the edge is considered to be a source input, and is currently located in the D-group rows of the subarray. As a result, the algorithm copies the input operand associated with the edge from its D-group row to the first available compute row. Note that if the edge *is* complemented, i.e., the input operand is negated (e.g., the edge with operand A for the blue node in Figure 2.5a), the algorithm allocates the first available compute row with dual contact cells (DCC0 or DCC1) to the input operand of the edge (lines 12–14 in Algorithm 1). If the edge is *not* complemented (e.g., the edge with operand B for the blue node in Figure 2.5a), a regular compute row is allocated to the input operand (lines 15–17 in Algorithm 1).

Case 2: if the edge is connected to another MAJ node in a higher level of the graph (line 18 in Algorithm 1), the edge has a parent node and the value of the input operand associated with the edge equals the result of the parent node, which is available in the compute rows that hold the result of the parent MAJ node. As a result, the algorithm maps the input operand of the edge to a compute row that holds the result of its parent node (lines 19–22 in Algorithm 1).

Case 3: if there are no free compute rows available, the algorithm considers the phase as *complete* and continues the allocations in the next phase (lines 23–26 in Algorithm 1).

Once DRAM rows are allocated to all the edges connected to a MAJ node, the algorithm stores the row-to-operand allocation information of the three input operands of the MAJ node in *row_operand_allocation* (line 27 in Algorithm 1) and associates this information with the MAJ node and the phase number that the allocations were performed in. The algorithm finishes once DRAM rows are allocated to *all* the input operands of all the MAJ nodes in the MIG. Figure 2.5b shows these allocations as the output of Task 1 for the full

addition example. The resulting *row_operand_allocation* is then used in Task 2 of Step 2 of the SIMDRAM framework (Section 2.3.2) to generate the series of µOps to compute the operation that the MIG represents.

Appendix C

Scalability of Operations

Table C.1 lists the semantics and the total number of AAP/APs required for each of the 16 SIMDRAM operations that we evaluate in this work (Section 2.6) for input element(s) of size n. Each operation is classified based on how the latency of the operation scales with respect to the element size n. Class 1, 2, and 3 operations scale linearly, logarithmically,and quadratically with n, respectively.

Table C.1: Evaluated SIMDRAM operations (for n-bit data).

Type	**Operation**	**# AAPs/APs**	**Class**	**Semantics**
Arithmetic	abs	$10n - 2$	Linear	$dst = (src > 0)?\ src : -(src)$
	addition	$8n + 1$	Linear	$dst = src_1 + src_2$
	bitcount	$\Omega = 8n - 8\log_2(n+1)$	Linear	$\sum_{i=0}^{n} src(i)$
		$O = 8n$		
	division	$8n^2 + 12n$	Quadratic	$dst = \frac{src_1}{src_2}$
	max	$10n + 2$	Linear	$dst = (src_1 > src_2)?\ src_1 : src_2$
	min	$10n + 2$	Linear	$dst = (src_1 < src_2)?\ src_1 : src_2$
	multiplication	$11n^2 - 5n - 1$	Quadratic	$dst = src_1 \times src_2$
	ReLU	$3n + ((n-1)\ mod\ 2)$	Linear	$dst = (src \geq 0)?\ src : 0$
	subtraction	$8n + 1$	Linear	$dst = src_1 - src_2$
Predication	if_else	$7n$	Linear	$dst = (sel)?\ src_1 : src_2$
Reduction	and_reduction	$5\lfloor\frac{n}{2}\rfloor + 2$	Logarithmic	$Y = src(1) \wedge src(2) \wedge src(3)$
	or_reduction	$5\lfloor\frac{n}{2}\rfloor + 2$	Logarithmic	$Y = src(1) \vee src(2) \vee src(3)$
	xor_reduction	$6\lfloor\frac{n}{2}\rfloor + 1$	Logarithmic	$Y = src(1) \oplus src(2) \oplus src(3)$
Relational	equal	$4n + 3$	Linear	$dst = (src_1 == src_2)$
	greater	$3n + 2$	Linear	$dst = (src_1 > src_2)$
	greater_equal	$3n + 2$	Linear	$dst = (src_1 \geq src_2)$

Appendix D

Evaluated Real-World Applications

Convolutional Neural Networks (CNNs). CNNs [141, 203, 319] are used in many classification tasks such as image and handwriting classification. CNNs are often computationally intensive as they use many general-matrix-multiplication (GEMM) operations using floating-point operations for each convolution. Prior works [141, 235, 319] demonstrate that instead of the costly floating-point multiplication operations, convolutions can be performed using a series of bitcount, addition, shift, and XNOR operations. In this work, we use the XNOR-NET [319] implementations of VGG-13, VGG-16, and LeNET provided by [141], to evaluate the functionality of SIMDRAM. We modify these implementations to make use of SIMDRAM's bitcout, addition, shift, and XNOR operations. We evaluate all three networks for inference using two different datasets: VGG-13 and VGG-16 (using CIFAR-10 [202]), and LeNet-5 (using MNIST [76]).

k-Nearest Neighbor Classifier (kNN). We use a kNN classifier to solve the handwritten digits recognition problem [210]. The kNN classifier finds a group of k objects in the input set using a simple distance algorithms such as Euclidean distance [100]. In our evaluations, we use SIMDRAM to implement the Euclidean distance algorithm entirely in DRAM. We evaluate a kNN algorithm using the MNIST dataset [76] with 3000 training images and 1000 testing images. We quantize the inputs using an 8-bit representation.

Database. We evaluate SIMDRAM using two different database workloads. First, we evaluate a simple table scan query `'select count(*) from T where c1 <= val <= c2'` using the BitWeaving algorithm [233]. Second, we evaluate the performance of the TPC-H [372] scheme using query 01, which executes many arithmetic operations, including addition and multiplication. For our evaluation, we follow the column-based data layout employed in [327] and use a scale factor of 100.

Brightness. We use a simple image brightness algorithm [97] to demonstrate the benefits of the SIMDRAM predication operation. The algorithm evaluates if a given brightness value is larger than 0. If so, it increases the pixel value of the image by the brightness value. Before assigning the new brightness value to the pixel, the algorithm verifies if the new pixel value is between 0 and 255. In our SIMDRAM implementation, we use both addition and predication operations.